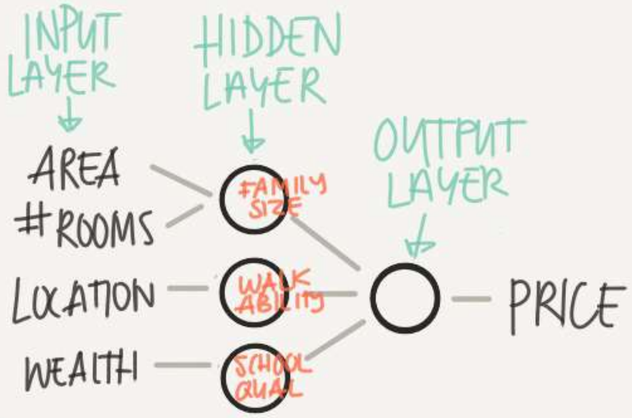
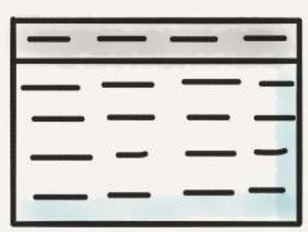
INTRO TO DEEP LEARNING



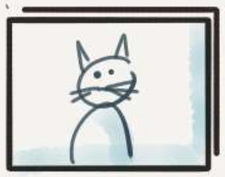
SUPERVISED LEARNING

INPUT: X	OUTPUT: Y	NN TYPE
HOME FEATURES AD + USER INFO	PRICE WILL CLICK ON AD (0/1)	STANDARD NN
IMAGE	OBJECT (1...1000)	CONV. NN (CNN)
AUDIO ENGLISH	TEXT TRANSCRIPT CHINESE	RECURRENT NN (RNN)
IMAGE/RADAR	POS OF OTHER CARS	CUSTOM/HYBRID

NNs CAN DEAL WITH BOTH STRUCTURED & UNSTRUCTURED DATA



STRUCTURED

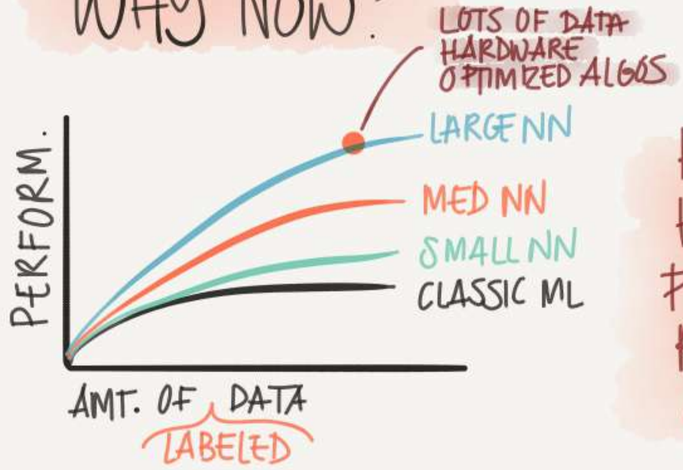


"THE QUICK BROWN FOX"

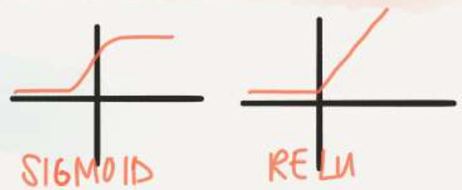
UNSTRUCTURED

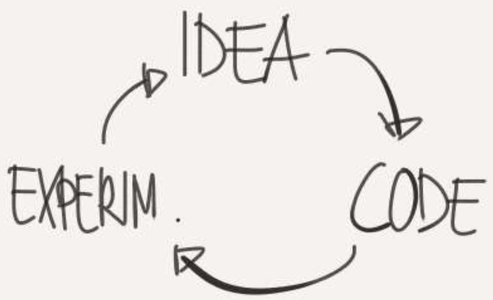
HUMANS ARE GOOD AT THIS

WHY NOW?



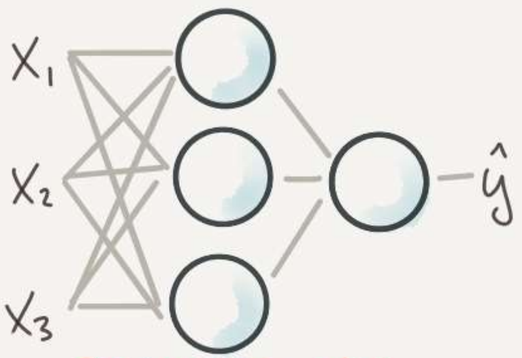
ONE OF THE BIG BREAKTHROUGHS HAS BEEN MOVING FROM SIGMOID TO RELU FOR FASTER GRADIENT DESCENT



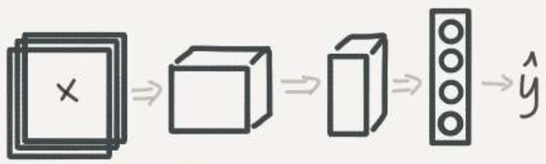


FASTER COMPUTATION IS IMPORTANT TO SPEED UP THE ITERATIVE PROCESS

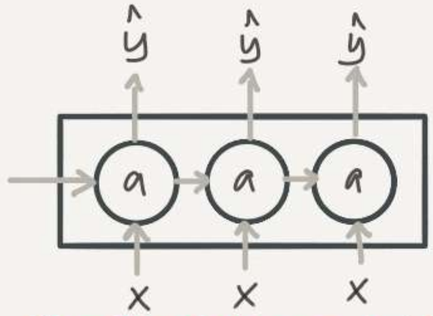
NETWORK ARCHITECTURES



STANDARD NN



CONVOLUTIONAL NN

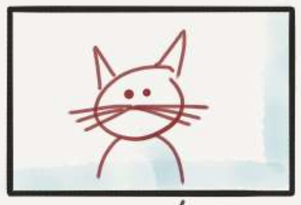


RECURRENT NN

LOGISTIC REGRESSION

AS A NEURAL NET

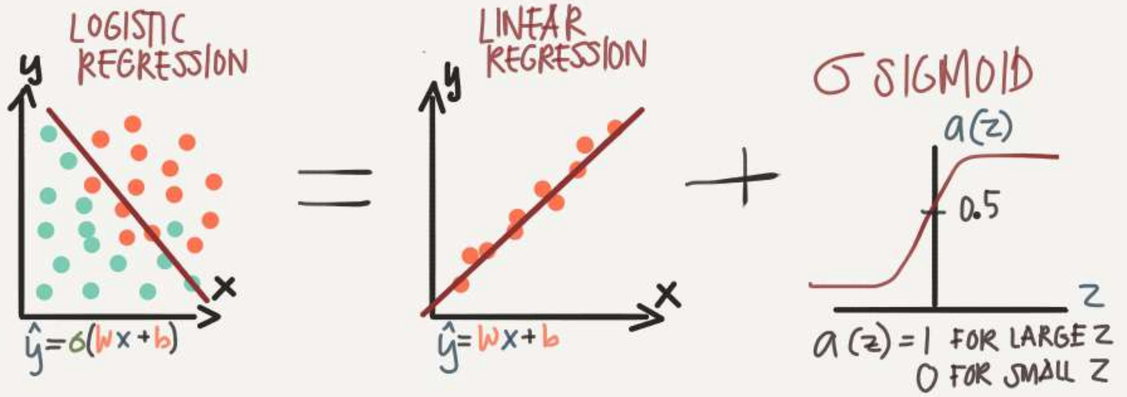
BINARY CLASSIFICATION



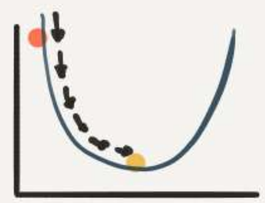
1 : CAT
0 : NOT CAT

X

y



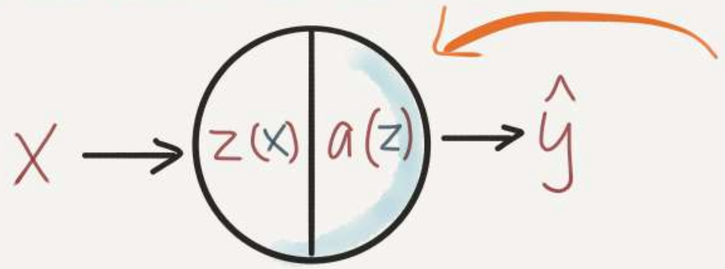
FINDING THE MINIMUM WITH GRADIENT DESCENT



1. FIND THE DOWNHILL DIRECTION (USING DERIVATIVES)
2. WALK (UPDATE W & B) AT A α LEARNING RATE

REPEAT UNTIL YOU REACH BOTTOM (CONVERGE)

PUTTING IT ALL TOGETHER



$$z(x) = wx + b$$

$$\hat{y} = a(z) = \sigma\,\text{SIGMOID}(z)$$

1. FORWARD PROPAGATION • CALCULATE $\hat{y}$
2. BACKWARD PROPAGATION • GRADIENT DESCENT + UPDATE W & b

REPEAT UNTIL IT CONVERGES

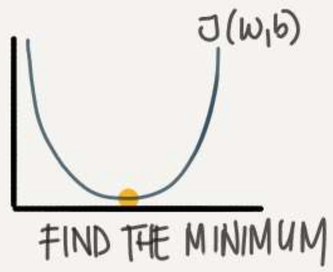
THE TASK IS TO LEARN w & b BUT HOW?

A : OPTIMIZE HOW GOOD THE GUESS IS BY MINIMIZING THE DIFF BETWEEN GUESS ($\hat{y}$) AND TRUTH (y)

$$\text{LOSS} = \mathcal{L}(\hat{y}, y)$$

$$\text{COST} = J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

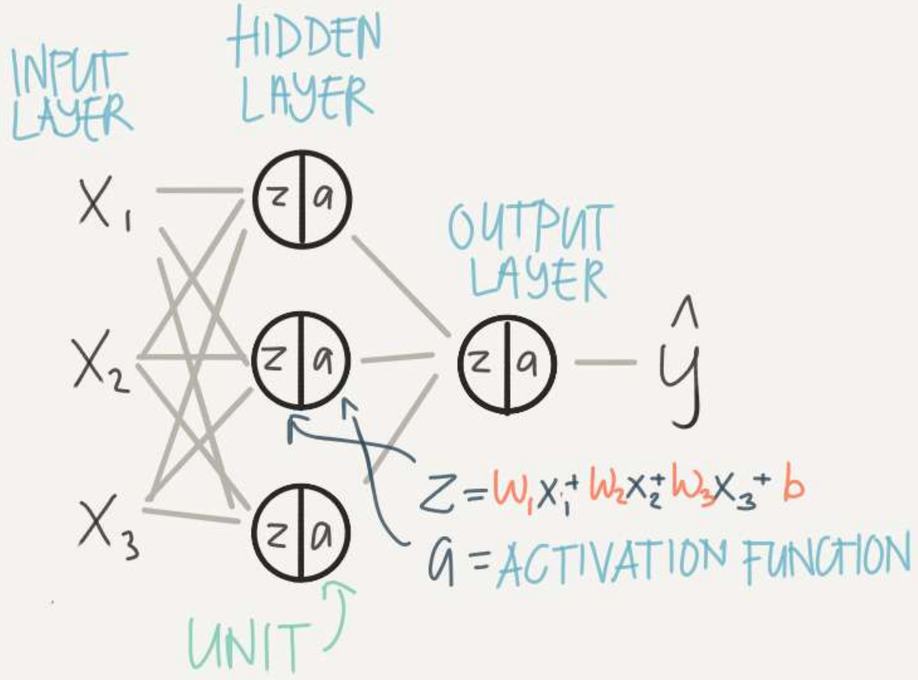
COST = LOSS FOR THE ENTIRE DATASET



NEURAL NETWORKS & DEEP LEARNING • COURSERA • ANDREW NG • WK 2

SHALLOW NEURAL NETS

2 LAYER NEURAL NET



WHY ACTIVATION FUNCTIONS?

EX. WITH NO ACTIVATION – $a = z$

$$a^{[1]} = z^{[1]} = w^{[1]}x + b^{[1]} \quad \text{LAYER 1}$$
$$a^{[2]} = z^{[2]} = w^{[2]}a^{[1]} + b^{[2]} \quad \text{LAYER 2}$$

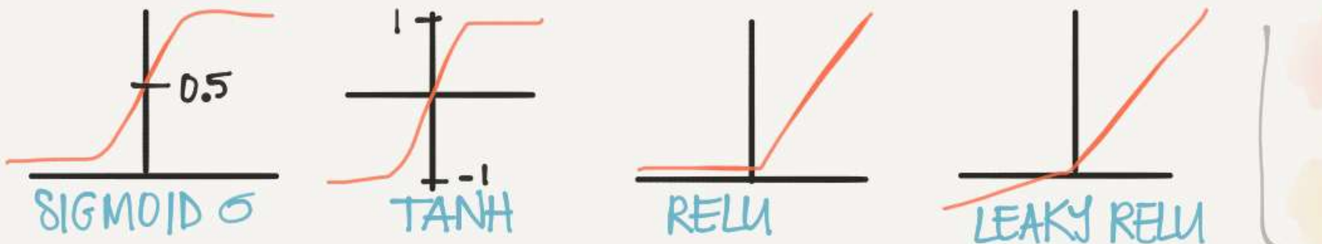
PLUG IN $a^{[1]}$

$$a^{[2]} = w^{[2]}(w^{[1]}x + b^{[1]}) + b^{[2]}$$
$$= w^{[2]}w^{[1]}x + w^{[2]}b^{[1]} + b^{[2]}$$
$$w'x + b'$$

← LINEAR FUNCTION

WE COULD JUST AS WELL HAVE SKIPPED THE WHOLE NEURAL NET & USED LIN. REGR.

ACTIVATION FUNCTIONS



SIGMOID σ
- BINARY CLASSIFIER - ONLY USED FOR OUTPUT LAYER
- SLOW GRAD DESCENT SINCE SLOPE IS SMALL FOR LARGE/SMALL VAL

TANH
- NORMALIZED ⇒ GRADIENT DESCENT IS FASTER

RELU
- DEFAULT CHOICE FOR ACTIVATION
- SLOPE = 1/0

LEAKY RELU
- AVOIDS UNDEF SLOPE AT 0 BUT RARELY USED IN PRACTICE

INITIALIZING w+b

WHAT IF: INIT TO 0

THIS WILL CAUSE ALL THE UNITS TO BE THE SAME AND LEARN EXACTLY THE SAME FEATURES

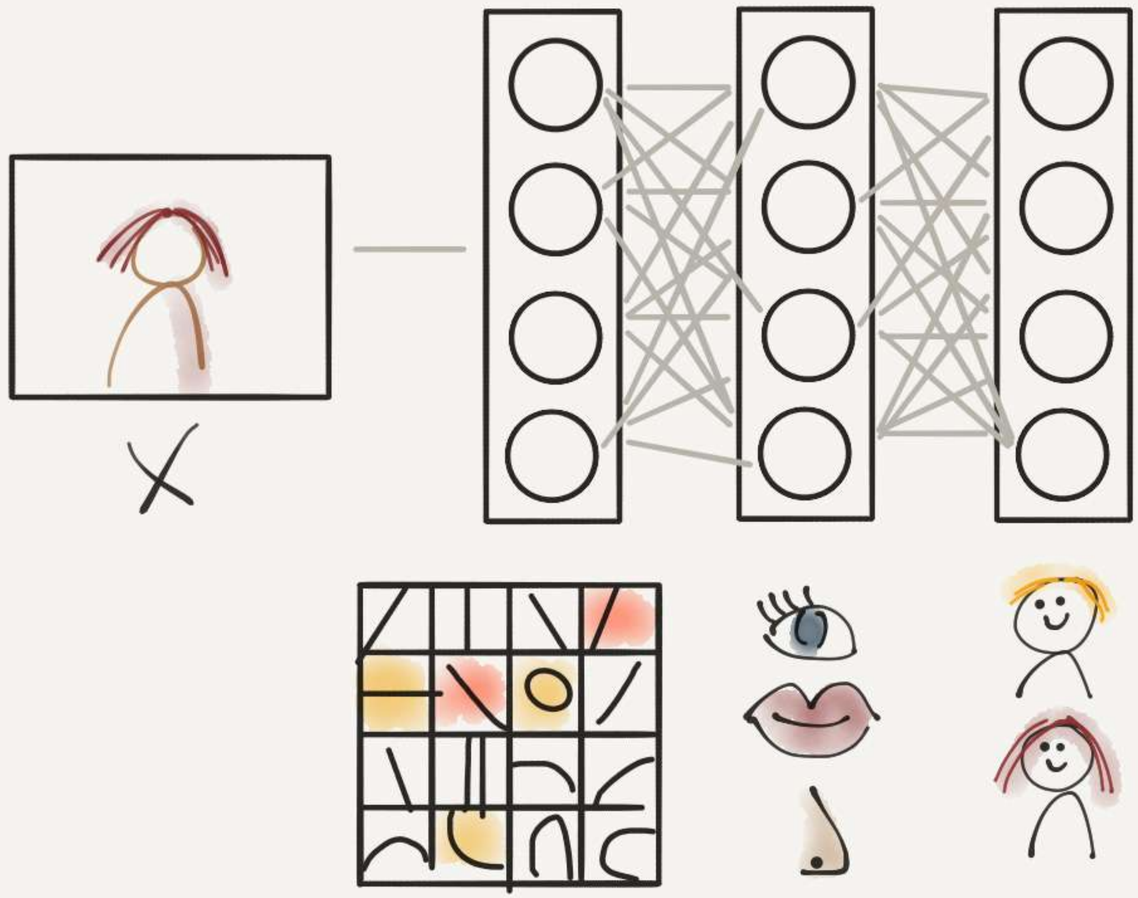
SOLUTION: RANDOM INIT

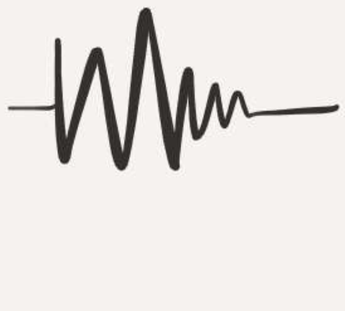
BUT ALSO WANT THEM SMALL SO RAND * 0.01 ← HYPERPARAM

DEEP NEURAL NETS

THERE ARE FUNCTIONS A SMALL DEEP NET CAN COMPUTE THAT SHALLOW NETS NEED EXP. MORE UNITS TO COMP.

WHY DEEP NEURAL NETS?





LOW LEVEL AUDIO WAVE FEATURES
↗↘ PITCH

— PHONEMES
C A T

— WORDS — SENTENCES

VERY DATA HUNGRY

NEED LOTS OF COMPUTER POWER

ALWAYS VECTORIZE
VECTOR MULT · CHEAPER THAN FOR LOOPS

COMPUTE ON GPUs

LOTS OF HYPERPARAMS

LEARNING RATE α
#ITERATIONS
HIDDEN LAYERS
HIDDEN UNITS
CHOICE OF ACTIVATION
MOMENTUM
MINI-BATCH SIZE
REGULARIZATION

SETTING UP
YOUR ML APP

CLASSIC ML

100 - 10000 SAMPLES

TRAIN	DEV	TEST
60%	20%	20%

DEEP LEARNING

1M SAMPLES

TRAIN	D	T
98%	1%	1%

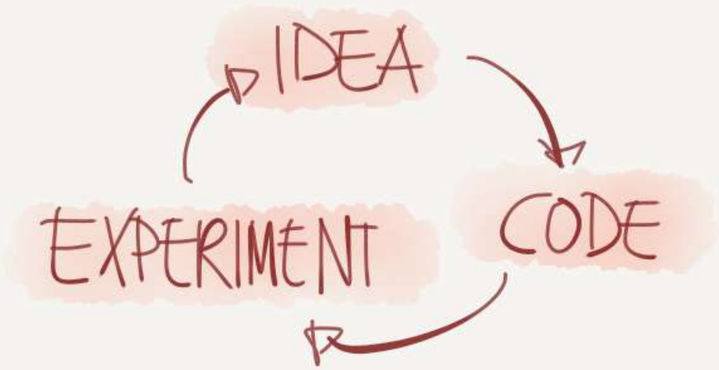
ALL FROM SAME PLACE → DISTRIBUTION

EX: TRAIN  PRO CAT PICS FROM INTERNET

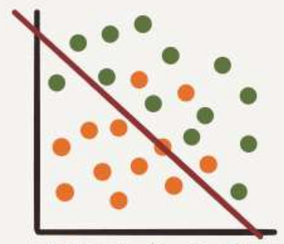
DEV/TEST  BLURRY CAT PICS FROM APP

TIP

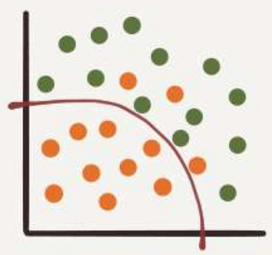
DEV & TEST SHOULD COME FROM SAME DISTRIBUTION



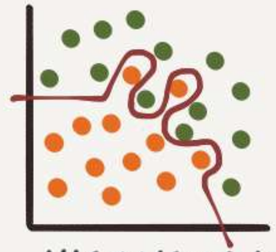
BIAS/VARIANCE



HIGH BIAS "UNDERFIT"



JUST RIGHT

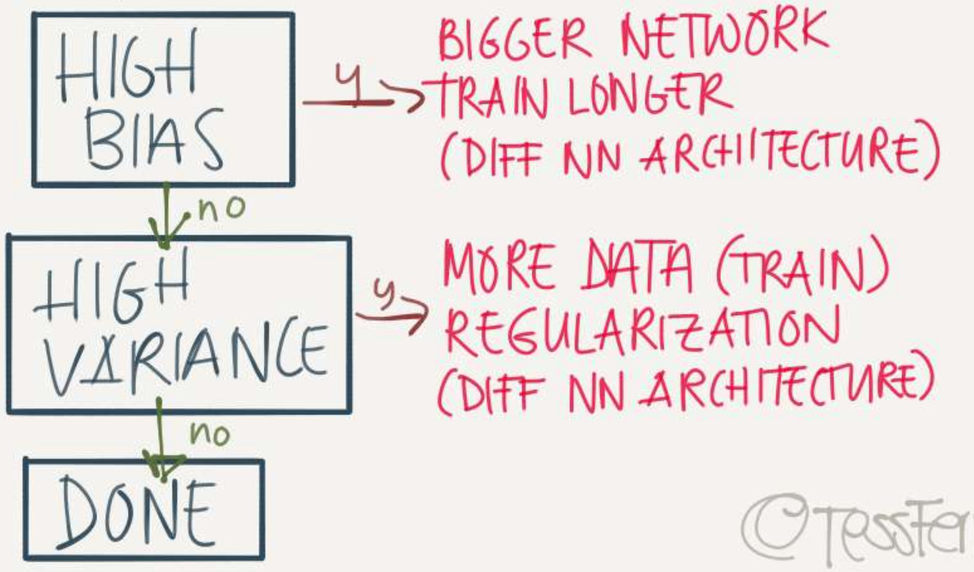


HIGH VARIANCE "OVERFIT"

	ERROR			
TRAIN	1%	15%	15%	0.5%
TEST	11%	16%	30%	1%
	HIGH VARIANCE	HIGH BIAS	HIGH BIAS & VARIANCE	LOW BIAS & VARIANCE

ASSUMING HUMANS GET 0% ERROR

THE ML RECIPE



REGULARIZATION
PREVENTING OVERFITTING

L2 REGULARIZATION

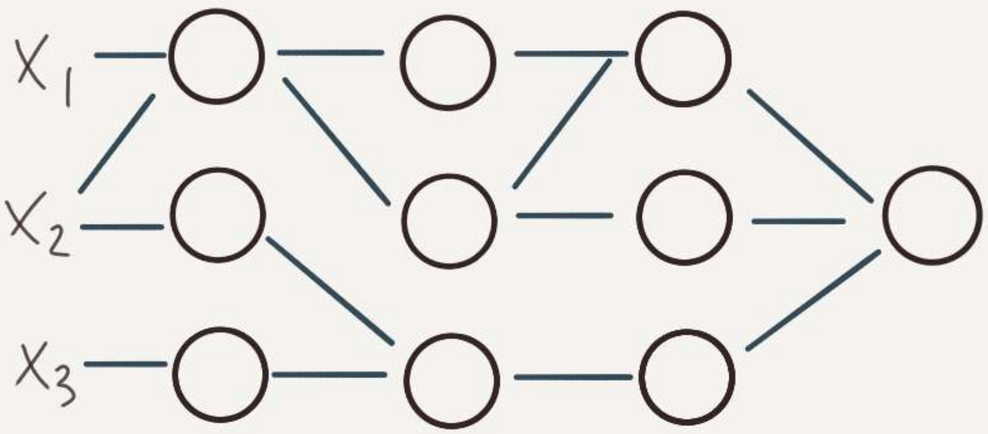
$$\text{COST: } J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y},y) + \frac{\lambda}{2m}\|w\|_2^2$$

← EUCLIDEAN NORM

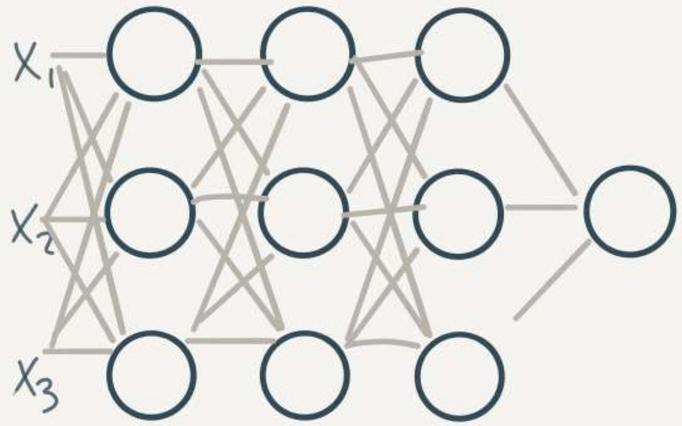
L1 REGULARIZATION

$$\text{COST: } J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y},y) + \frac{\lambda}{m}\|w\|_1$$

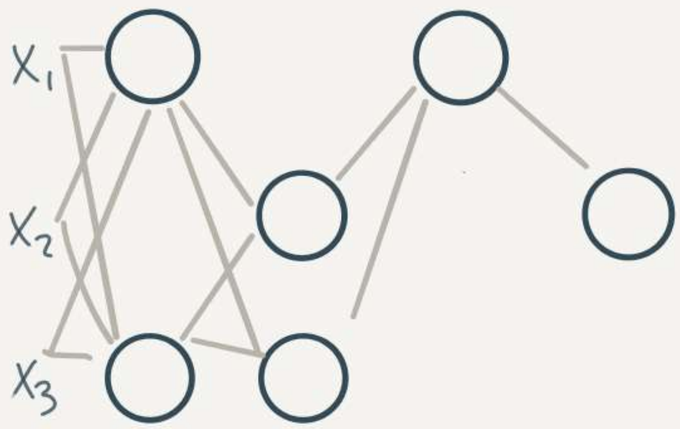
BOTH PENALIZE LARGE WEIGHTS ⇒
SOME WILL BE CLOSE TO Ø ⇒
SIMPLER NETWORKS



DROPOUT



FOR EACH ITERATION & SAMPLE SOME NODES ARE RANDOMLY DROPPED (BASED ON KEEP-PROB)

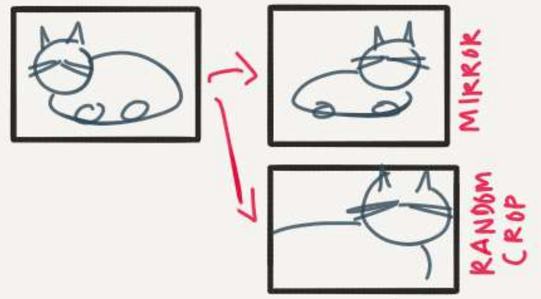


WE GET SIMPLER NWs & LESS CHANCE TO RELY ON SINGLE FEATURES

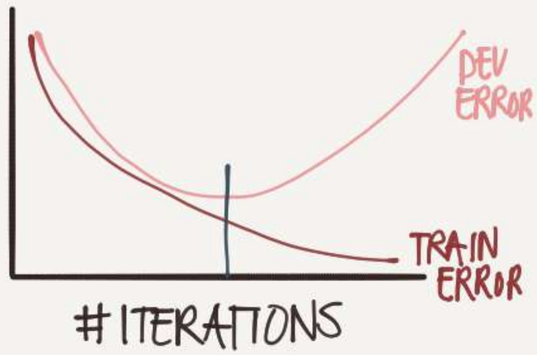
OTHER REGULARIZATION TECHNIQUES

DATA AUGMENTATION

GENERATE NEW PICS FROM EXISTING



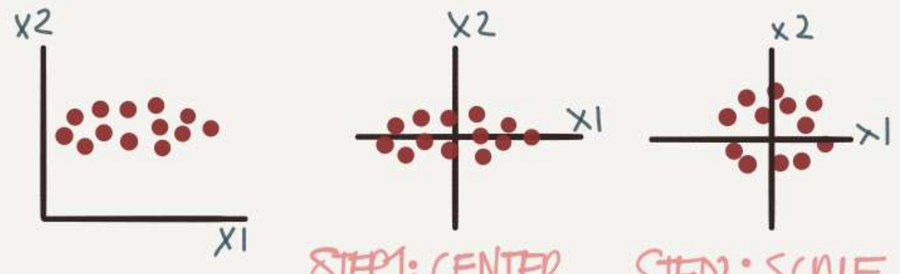
EARLY STOPPING



PROBLEM: AFFECTS BOTH BIAS & VARIANCE

OPTIMIZING TRAINING

NORMALIZING INPUTS



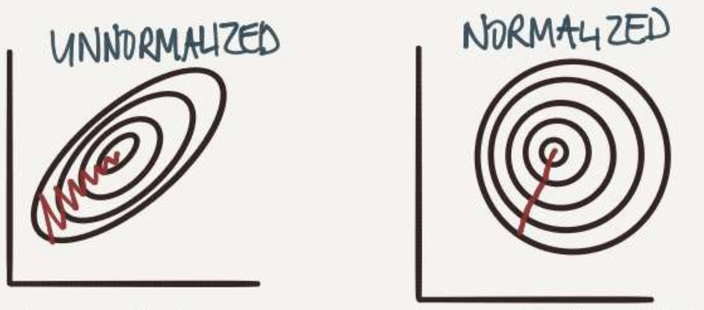
STEP1: CENTER AROUND 0,0

STEP2: SCALE SO VARIANCE IS SAME CX $-1 \rightarrow 1$

TIP

USE SAME AVG/VAR TO NORMALIZE DEV/TEST

WHY DO WE DO THIS?



IF WE NORMALIZE, WE CAN USE A MUCH LARGER LEARNING RATE α

DEALING WITH VANISHING/EXPLODING GRADIENTS

EX: DEEP NW (L LAYERS)

$\hat{y} = W^{[L-1]} W^{[L-2]} \cdots W^{[1]} X + b$

IF $W = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix} \Rightarrow 0.5^{L-1} \Rightarrow$ VANISHING

OR $W = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Rightarrow 1.5^{L-1} \Rightarrow$ EXPLODING

IN BOTH CASES GRADIENT DESCENT TAKES A VERY LONG TIME

PARTIAL SOLUTION: CHOOSE INITIAL VALUES CAREFULLY

$W^{[l]} = rand * \sqrt{\frac{2}{n^{[l-1]}}}$ (FOR RELU)

$\#inputs$

XAVIER $\sqrt{\frac{1}{n^{[l-1]}}}$ (FOR TANH)

SETS THE VARIANCE

GRADIENT CHECKING

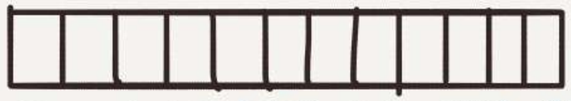
IF YOUR COST DOES NOT DECREASE ON EACH ITER YOU MAY HAVE A BACKPROP BUG.

GRADIENT CHECKING APPROXIMATES THE GRADIENTS SO YOU CAN VERIFY CALC.

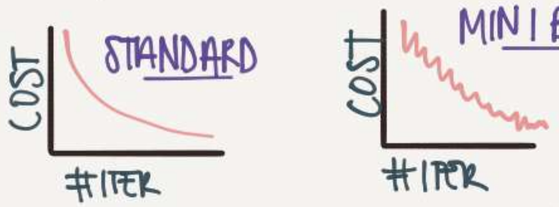
NOTE ONLY USE WHEN DEBUGGING SINCE IT'S SLOW

OPTIMIZATION ALGORITHMS

MINI-BATCH GRAD. DESCENT

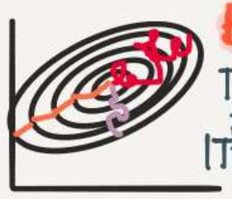


SPLIT YOUR DATA INTO MINI-BATCHES & DO GRAD DESCENT AFTER EACH BATCH
THIS WAY YOU CAN PROGRESS AFTER JUST A SHORT WHILE



CHOOSING THE MINIBATCH SIZE

SIZE = m → BATCH GRAD DESC.
SIZE = 1 → STOCHASTIC GRAD DESC



BATCH — TOO LONG PER ITERATION
MIDDLE
STOCHASTIC — LOOSE ALMOST ALL SPEED FROM VECTORIZATION

- SHORT ITER.
- USES VECT.

TIP
IF YOU HAVE < 2000 SAMPLES
USE SIZE = 2000
OTHERWISE, USE 64, 128, 256...
SO X+Y FITS IN CPU/GPU CACHE

GRADIENT DESCENT W. MOMENTUM

WE WANT TO REDUCE OSCILLATION ↕ SO WE GET TO THE GOAL FASTER

SOLUTION: SMOOTH OUT THE CURVE BY TAKING AN EXPONENTIALLY WEIGHTED AVERAGE OF THE DERIVATIVES (i.e. LAST ONE HAS MORE IMPORTANCE)

RMSProp - ROOT MEAN SQUARED

NORMALIZE GRADIENT USING A MOVING AVG.

$$S_{dw} = \beta S_{dw} + (1-\beta)\, dw^2$$
$$S_{db} = \beta S_{db} + (1-\beta)\, db^2$$
$$w = w - \alpha \frac{dw}{\sqrt{S_{dw}}} \qquad b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$

ADAM OPTIMIZATION

COMBO OF GD W MOMENTUM & RMSProp

LEARNING RATE DECAY

IDEA: USE A LARGE α IN THE BEGINNING. THEN DECREASE AS WE GET CLOSER TO GOAL

OPTION 1: $\alpha = \frac{1}{1 + \text{DECAYRATE} \cdot \text{EPOCH}}\,\alpha_0$

EXPONENTIAL: $\alpha = 0.95^{\text{EPOCH}}\,\alpha_0$

OPTION 3: $\alpha = \frac{k}{\sqrt{\text{EPOCH}}}\,\alpha_0$

OPTION 4: $\alpha = \frac{k}{\sqrt{t}}\,\alpha_0$

OPTION 5: DISCRETE STAIRCASE

OPTION 6: MANUAL

EPOCH = 1 PASS THROUGH THE DATA

IMPROVING DEEP NEURAL NETS · COURSERA

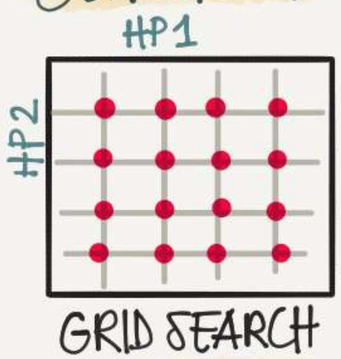
HYPERPARAM TUNING

WHICH HYPERPARAMS ARE MOST IMPORTANT?

- α LEARNING RATE
- # HIDDEN UNITS
- MINIBATCH SIZE
- β MOMENTUM TERM = 0.9
- # LAYERS
- LEARNING RATE DECAY
- $\beta_1 = 0.9$ $\beta_2 = 0.999$ $\varepsilon = 10^{-8}$ (ADAM)

TESTING VALUES

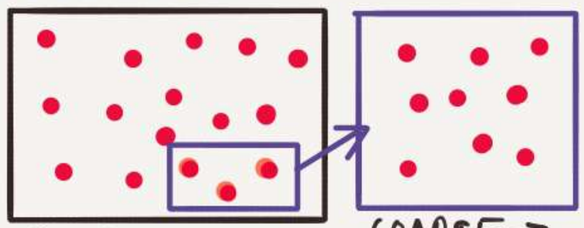
CLASSIC ML



GRID SEARCH

PROBLEM: ONE ITERATION TAKES A LONG TIME & IN 16 GO'S WE HAVE ONLY TRIED 4α - BUT 4 DIFF ε ← NOT AS IMPORTANT

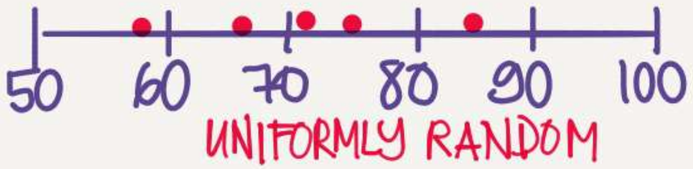
SOLUTION



RANDOM SEARCH + COARSE → DENSE

USE AN APPROPRIATE SCALE

HIDDEN UNITS



UNIFORMLY RANDOM

α LEARNING RATE

$r = -4 \cdot \text{rand}\ [-4, 0]$

$\alpha = 10^r$

β EXP WEIGHT AVE

$r = -3 \cdot \text{rand}\ [-3, -1]$

$\beta = (1 - 10^r)$

TIP

RE-EVALUATE YOUR HYP-PARAMS EVERY FEW MONTHS

PANDA VS CAVIAR

MY PANDA IS ACTUALLY A MISCLASSIFIED CAT BECAUSE I CAN'T DRAW PANDAS

BABYSIT ONE MODEL & TUNE

SPAWN LOTS OF MODELS W DIFF HP

GOOD IF YOU HAVE LOTS OF SPARE COMP POWER

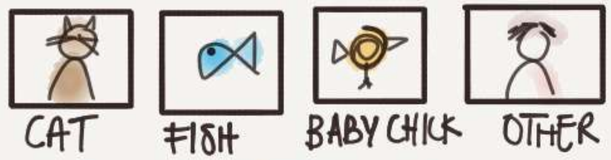
MISC. EXTRAS

BATCH NORMALIZATION

NORMALIZE LAYER OUTPUT

- SPEEDS UP TRAINING
- MAKES WEIGHTS DEEPER IN NW MORE ROBUST (COVARIATE SHIFT)
- SLIGHT REGULARIZING EFFECT

MULTICLASS CLASSIFIC.

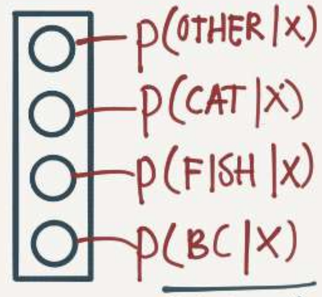


C = # CLASSES = 4

SOFTMAX ACTIVATION

$$t = e^{(z^{[L]})}$$

$$a^{[L]} = \frac{t}{\sum t_i}$$



SUM: 1

EX: $z^{[L]} = \begin{bmatrix} 5 \\ 2 \\ -1 \\ 3 \end{bmatrix}$ $t = \begin{bmatrix} e^5 \\ e^2 \\ e^{-1} \\ e^3 \end{bmatrix} = \begin{bmatrix} 148.4 \\ 7.4 \\ 0.4 \\ 20.1 \end{bmatrix}$ 176.3

$\Rightarrow a^{[L]} = \frac{t}{176.3} = \begin{bmatrix} 0.842 \\ 0.042 \\ 0.02 \\ 0.114 \end{bmatrix}$ 11.4% PROB IT'S A BABY CHICK

STRUCTURING YOUR ML PROJECTS

SETTING YOUR GOAL

★ GOAL SHOULD BE A SINGLE #

	PRECISION	RECALL
A	95%	90%
B	98%	85%

↳ IS A OR B BEST?

	PRECISION	RECALL	F1
A	95%	90%	92.4%
B	98%	85%	91%

↳ A IS BEST

F1 = HARMONIC MEAN BETW. RECALL & PRECISION

★ DEFINE OPTIMIZING VS SATISFICING METRICS

	ACCURACY	RUNTIME
A	90%	80ms
B	92%	95ms
C	~~95%~~	~~1500ms~~

MAXIMIZE ACC.
GIVEN TIME ≤ 100ms

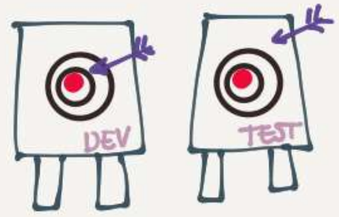
ACCURACY = OPTIMIZING
RUNTIME = SATISFICING

SELECTING YOUR DEV/TEST SETS

DATA

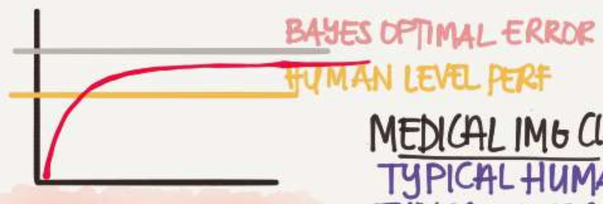
US
UK
~~EUROPE~~
S.AM
INDIA
CHINA
AUST.

OPTION 1:
DEV = UK, US, EUR
TEST = REST



IF DEV & TEST ARE DIFF & WE OPTIMIZE FOR DEV WE WILL MISS THE TEST-TARGET

HUMAN LEVEL PERF



WHY DOES ACC SLOW DOWN WHEN WE SURPASS HUMAN LEVEL PERF?

MEDICAL IMG CLASS
TYPICAL HUMAN 3%
TYPICAL DOCTOR 1%
EXPERIENCED DR. 0.7%
TEAM OF EXP DRs. 0.5%
↑ HUMAN LEV PERF (PROXY FOR BAYES)

1. OFTEN CLOSE TO BAYES
2. A HUMAN CAN NO LONGER HELP IMPROVE (INSIGHTS)
3. DIFFICULT TO ANALYSE BIAS/VARIANCE

CAT CLASSIFICATION

	A	B
HUMAN	1%	7.5%
TRAIN ERR	8%	8%
DEV ERR	10%	10%
	FOCUS ON BIAS	FOCUS ON VARIANCE

↳ BLURRY
← AVOIDABLE BIAS
← VARIANCE

HUMAN
| AVOIDABLE BIAS
TRAIN
| VARIANCE
DEV

- TRAIN BIGGER NETW.
- TRAIN LONGER/BETTER OPT. ALGOS (RMSPROP & ADAM)
- CHANGE NN ARCH OR HYPERPARAMS
- MORE DATA (TRAIN)
- REGULARIZATION
- NN ARCHITECTURE

	A	B
HUMAN	0.5	0.5
TRAIN ERR	0.6	0.3
DEV ERR	0.8	0.4
AVOID. BIAS	0.1	?

← AVOIDABLE BIAS
← VARIANCE

DON'T KNOW IF WE OVERFIT OR IF WE'RE CLOSE TO BAYES

OPTIONS TO PROCEED ARE UNCLEAR

STRUCTURING ML PROJECTS • COURSERA

ERROR ANALYSIS

YOU HAVE 10% ERRORS, SOME ARE DOGS MIS-CLASSIFIED AS CATS. SHOULD YOU TRAIN ON MORE DOG PICS?

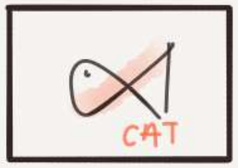
1. PICK 100 MIS-LABLED
2. COUNT ERROR REASONS

	DOG	BLURRY	INSTA FILTER	BIG CAT	...
1	1		1		
2				1	
3		1			
...					
100			1		
	5	...			

5% OF ALL ERRORS

FOCUSING ON DOGS. THE BEST WE CAN HOPE FOR IS 9.5% ERROR

YOU FIND SOME INCORR. LABLED DATA IN THE DEV SET. SHOULD YOU FIX IT?



DL ALGORITHMS ARE PRETTY ROBUST TO RANDOM ERRORS.

BUT NOT TO SYSTEMATIC ERR.

(EX. ALL WHITE CATS INCORR LABLED AS MICE)

ADD EXTRA COL. IN ERROR ANALYSIS AND USE SAME CRITERIA

NOTE IF YOU FIX DEV YOU SHOULD FIX TEST AS WELL.

FOR NEW PROJ. BUILD 1st SYSTEM QUICK & ITERATE

EX: SPEECH RECOGNITION

WHAT SHOULD YOU FOCUS ON?

NOISE
ACCENTS
FAR FROM MIKE

1. START QUICKLY
   DEV/TEST METRICS
2. GET TRAIN-SET
3. TRAIN
4. BIAS/VARIANCE ANAL
5. ERROR ANALYSIS
6. PRIORITIZE NEXT STEP

TRAIN vs DEV/TEST MISMATCH

AVAILABLE DATA

200k PRO CAT PICS FROM INTERNET

10k BLURRY CAT PICS FROM APP (WHAT WE CARE ABT)

HOW DO WE SPLIT → TRAIN/DEV/TEST?

OPTION 1: SHUFFLE ALL

PROBLEM: DEV/TEST IS NOW MOSTLY WEB IMG (NOT REPRES. OF END SCENARIO)

SOLUTION: LET DEV/TEST COME FROM APP · THEN SHUFFLE 5k OF APP PICS W WEB FOR TRAIN

BIAS & VARIANCE W MISMATCHED TRAIN/DEV

HUMANS	~0%
TRAIN	1%
DEV ERR	10%

IS THIS DIFF DUE TO THE MODEL NOT GENERALIZING OR IS DEV DATA MUCH HARDER

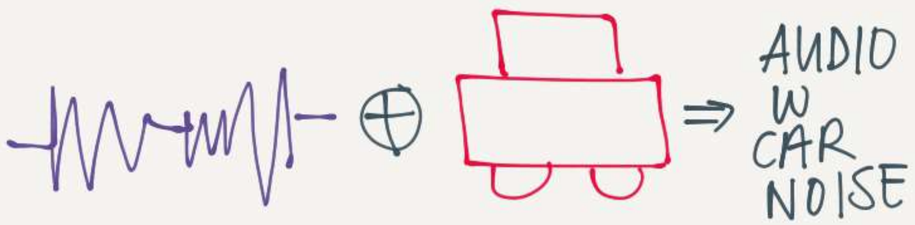
A: CREATE A TRAIN-DEV SET THAT WE DON'T TRAIN ON

	A	B	C	D
TRAIN	1%	1%	10%	10%
TRAIN-DEV	9%	15%	11%	11%
DEV	10%	10%	12%	20%
	VARIANCE	TRAIN/DEV MISMATCH	BIAS	BIAS + DATA MISMATCH

ADDRESSING DATA MISMATCH

EX. CAR GPS · TRAINING DATA IS 10,000H OF GENERAL SPEECH DATA

1. CARRY OUT MANUAL ERROR ANALYSIS TO UNDERSTAND THE DIFFERENCE (EX NOISE, STREET NUMBERS)
2. TRY TO MAKE TRAIN MORE SIMILAR TO DEV OR GATHER MORE DEV-LIKE TRAIN-DATA



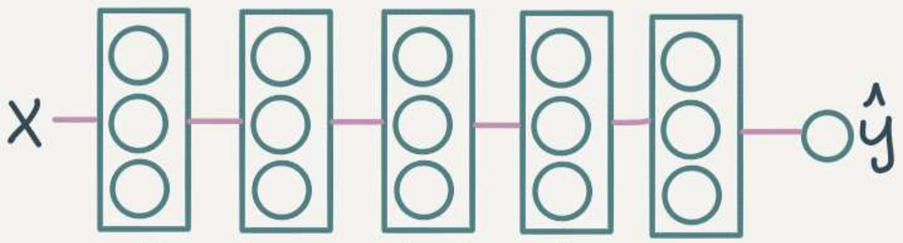
NOTE BE CAREFUL · IF YOU ONLY HAVE 1 HR OF CAR NOISE & APPLY IT TO 10K HR SPEECH YOU MAY OVERFIT TO THE CAR NOISE

STRUCTURING ML PROJECTS · COURSERA

EXTENDED LEARNING

TRANSFER LEARNING

PROBLEM: YOU WANT TO CLASSIFY SOME MEDICAL IMG. YOU HAVE AN NN THAT CLASSIFIES CATS



NN TO CLASSIFY CATS/DOGS/CARS

OPTION 1: YOU ONLY HAVE A FEW RADIOLOGY IMAGES

SOLUTION: INIT W. WEIGHTS FROM CAT NN
ONLY RETRAIN LAST LAYER(S) ON RADIOLOGY IMAGES

OPTION 2 YOU HAVE LOTS OF RADIOLOGY IMG.

SOLUTION: INIT WITH WEIGHTS FROM CAT NN
RETRAIN ALL LAYERS

TESS NOTES

THIS IS MICROSOFT CUSTOM VISION

MULTI TASK LEARNING

TRAINING ON MULT. TASKS AT ONCE

DETECT
CAR
STOPSIGN
PEDESTR.
TRAFFIC LIGHT

$\hat{y}$

$$\begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

UNLIKE SOFMAX · MANY THINGS CAN BE TRUE

$$\text{COST: } J(w,b) = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} \mathcal{L}(\hat{y}_j^{(i)}, y_j^{(i)})$$

SUMMING OVER ALL OUTP OPTIONS

WE COULD HAVE JUST TRAINED 4 NNS INSTEAD BUT.. MT LEARNING MAKES SENSE WHEN

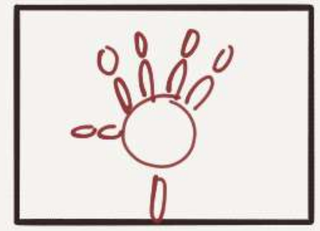
A. THE LEARNING DATA YOU HAVE FOR THE DIFF TASKS IS QUITE SIMILAR – & THE AMOUNTS (EG. 1K CARS, 1K STOP SIGNS)

B. THE SUM OF THE DATA ALLOWS YOU TO TRAIN A BIG ENOUGH NN TO DO WELL ON ALL TASKS

IN REALITY TRANSFER LEARNING IS USED MORE OFTEN

END-TO-END LEARNING

FROM X-RAY OF CHILDS HAND TELL ME THE AGE OF THE CHILD



TYPICAL SOLN:

1. LOCATE BONES TO FIND LENGTHS USING ML
2. TRAIN MODEL TO PREDICT AGE BASED ON BONELENGTH

END-TO-END

RADIOLOGY IMG → CHILD AGE

PROS:

- LET'S THE DATA SPEAK (MAYBE IT FINDS RELATIONS WE'RE UNAWARE OF)
- LESS HAND-DESIGNING OF COMPONENTS NEEDED

CONS:

- NEEDS LARGE AMTS OF LABLED DATA (X→Y)
- EXCLUDES POTENTIALLY USEFUL HAND-MADE COMPONENTS

CONVOLUTION
FUNDAMENTALS

COMPUTER VISION

PROBLEM: IMAGES CAN BE BIG

1000 × 1000 × 3 (RGB) = 3M

WITH 1000 HIDDEN UNITS WE NEED 3M * 1000 = 3B PARAMS

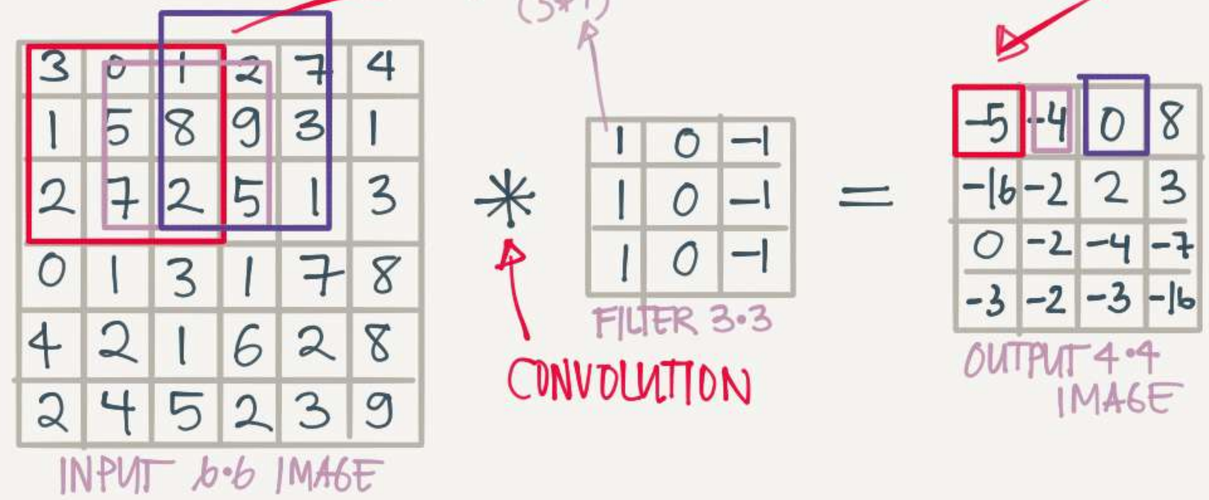
SOLUTION: USE CONVOLUTIONS

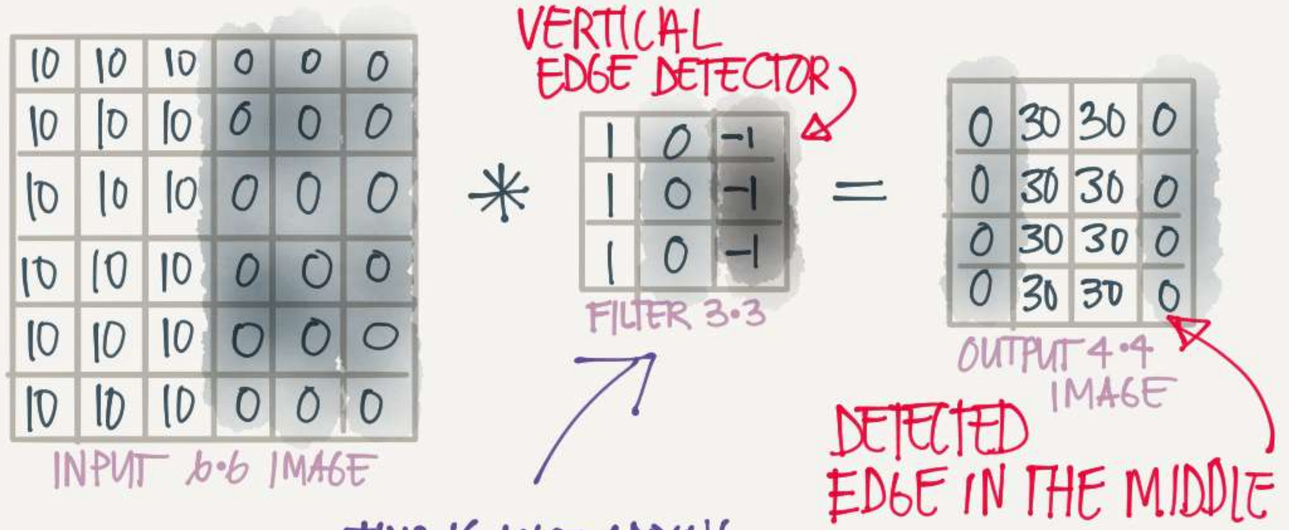
IT'S LIKE SCANNING OVER YOUR IMG WITH A MAGNIFYING GLASS OR FILTER

TESS NOTE

ALSO SOLVES THE PROBLEM THAT THE CAT IS NOT ALWAYS IN THE SAME LOCATION IN THE IMG

CONVOLUTION





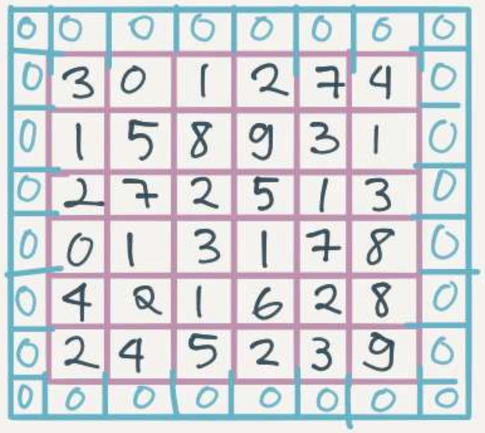
WE COULD HARD-CODE FILTERS · JUST LIKE WE CAN HARD-CODE HEURISTIC RULES.... BUT.... A MUCH BETTER WAY IS TO TREAT THE FILTER # AS PARAMS TO BE LEARNED

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

PADDING

PROBLEM: IMAGES SHRINK
6x6 → 3x3 → 4x4
PROBLEM: EDGES GET LESS 'LOVE'

SOLUTION: PAD W. A BORDER OF 0s BEFORE CONVOLVING



TWO COMMONLY USED PADDING OPTIONS
(HOW MUCH TO PAD)

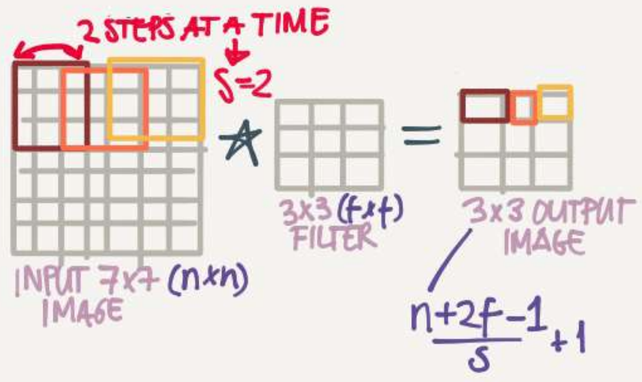
'VALID' ⇒ P = 0 NO PADDING

'SAME' ⇒ P = $\frac{f-1}{2}$ OUTPUT SIZE = INPUT SIZE

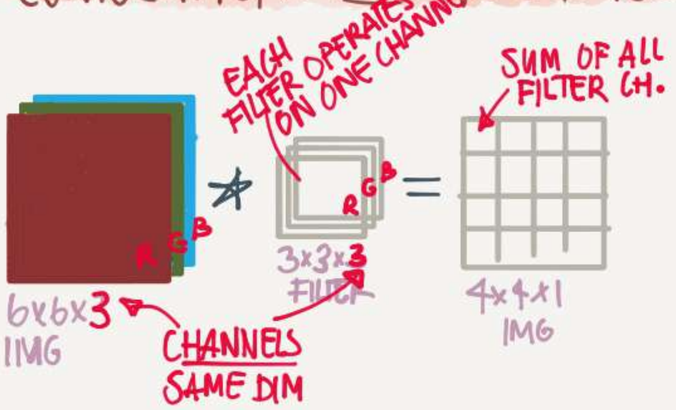
FILTER SIZE

STRIDE

WHAT PACE YOU SCAN WITH



CONVOLUTIONS OVER VOLUMES

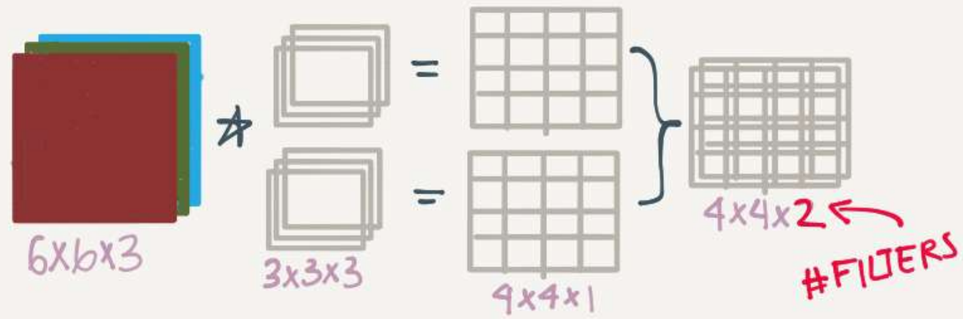


THIS ALLOWS US TO DETECT FEATURES IN COLOR IMAGES FOR EXAMPLE

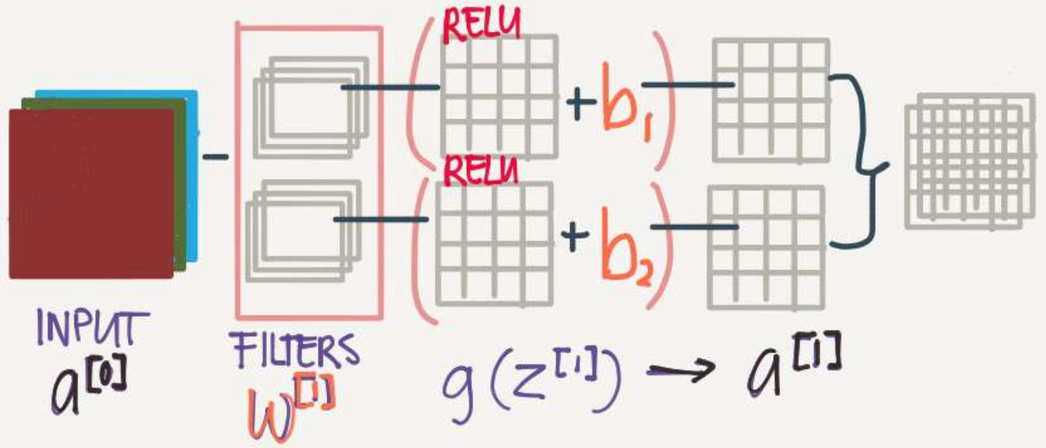
MAYBE WE WANT TO FIND ALL EDGES OR MAYBE ORANGE BLOBS

MULTIPLE FILTERS

DETECTING MULTIPLE FEATURES AT A TIME



ONE CONV. NET LAYER



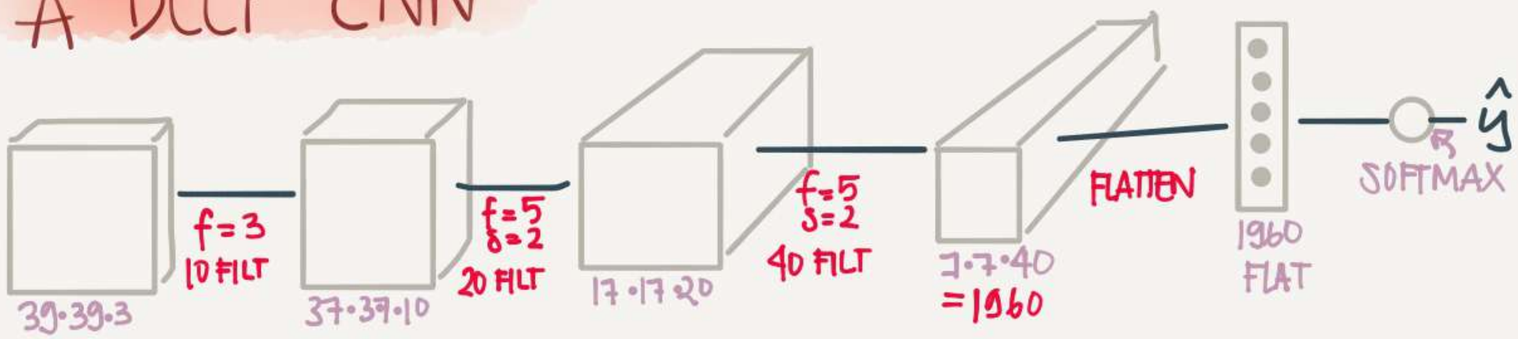
NOTE IT DOESN'T MATTER HOW BIG THE INPUT IS - THE LEARNABLE PARAMS W & b ONLY DEPEND ON THE # OF FILTERS AND THEIR SIZES.

$W = 3 \cdot 3 \cdot 3 \cdot 2 = 54$
$b = 2$
} 56 PARAMS TO LEARN

NOTE: ALL CONVOLUTION IDEAS CAN BE APPLIED TO 1D AS WELL LIKE EKG SIGNALS · AND 3D LIKE CT-SCANS

CONVOLUTIONAL NEURAL NETS · COURSERA

A DEEP CNN



A LOT OF THE WORK IS FIGURING OUT HYPERPARAMS = #FILTERS, STRIDE, PADDING ETC

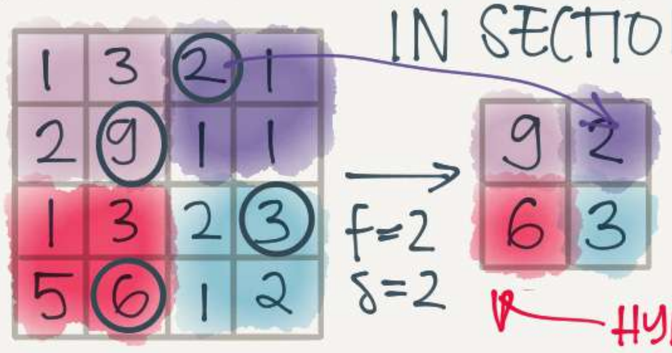
TYPICALLY SIZE → TREND DOWN
FILTERS → TREND UP

TYPICAL CONV.NET LAYERS

CONVOLUTION
POOLING
FULLY CONNECTED

POOLING (MAX)

FIND MAX VAL IN SECTION

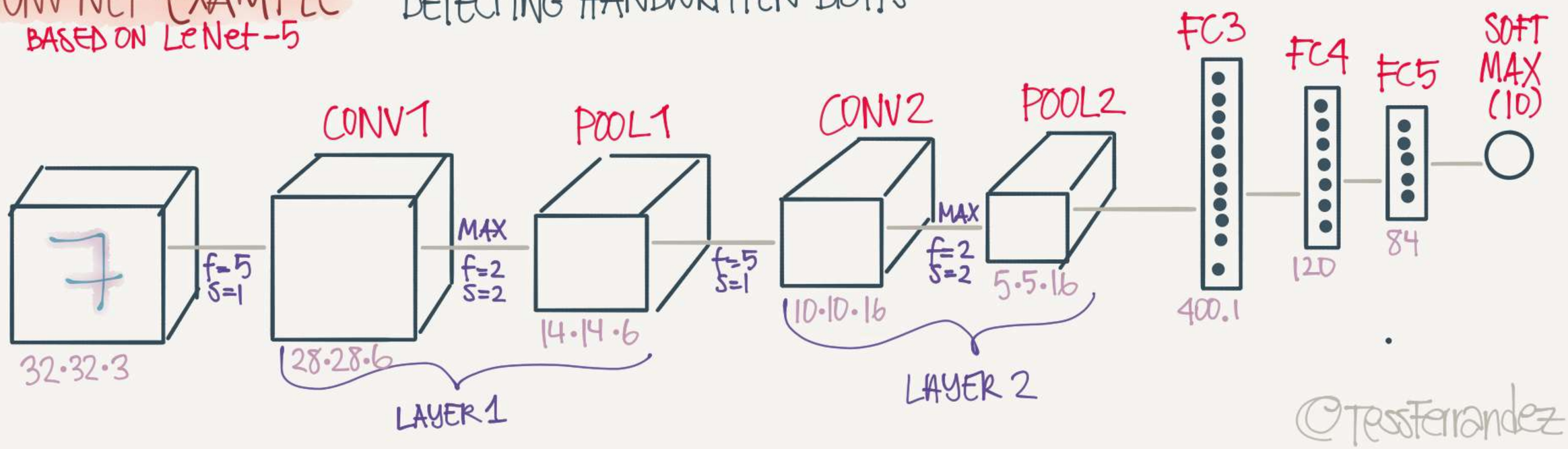


- REDUCES SIZE OF REPRES.
- SPEEDS UP COMPUTATION
- MAKES SOME OF THE DETECTED FEAT. MORE ROBUST

CONV NET EXAMPLE

BASED ON LeNet-5

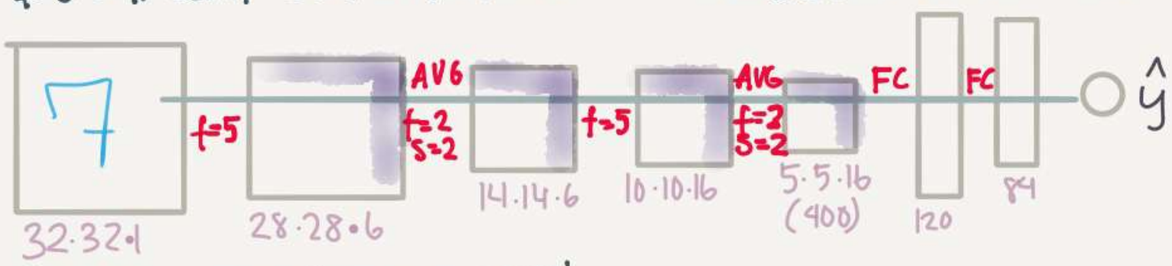
DETECTING HANDWRITTEN DIGITS



CLASSIC CONV. NETS

LeNet-5

DOCUMENT CLASSIFICATION ≈60k PARAMETERS



TRENDS: HEIGHT/WIDTH GO DOWN
CHANNELS GO UP

COMMON PATTERN: A COUPLE OF CONV(1+)/POOL LAYERS FOLLOWED BY A FEW FC

OLD STUFF: USED AVG POOLING INST. OF MAX
PADDING WAS NOT VERY COMMON
IT USED SIGMOID/TANH INST OF RELU

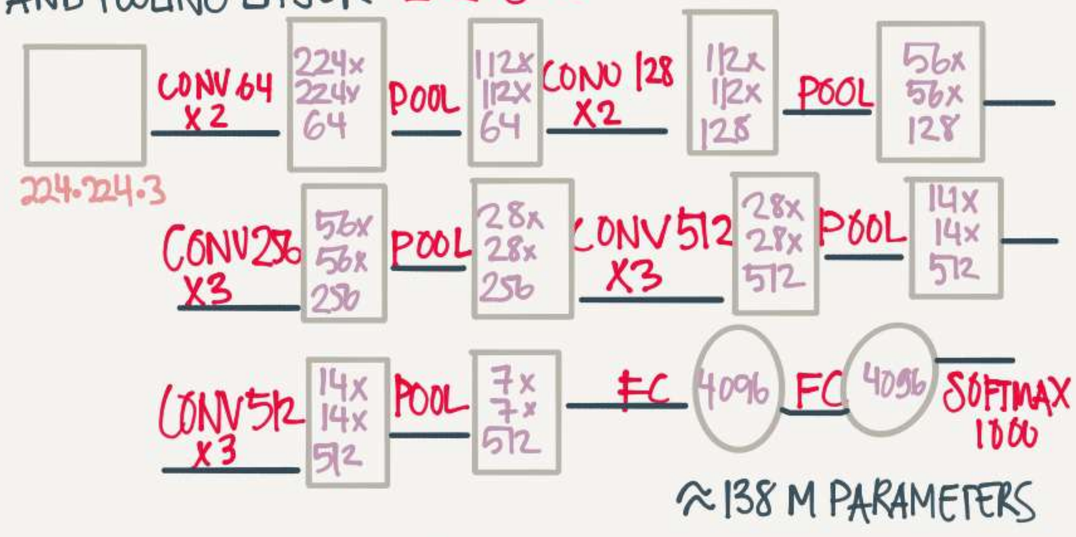
AlexNet

IMAGE CLASSIFICATION ≈60M PARAMETERS

- SIMILAR TO LeNet BUT MUCH BIGGER
- USES RELU
- THE NN THAT GOT RESEARCHERS INTERESTED IN VISION AGAIN

VGG-16

ALL CONV. LAYERS HAVE SAME #PARAMS
f=3x3 S=1 P=SAME
AND POOLING LAYER 2x2 S=2



≈138 M PARAMETERS

- VERY DEEP
- EASY ARCHITECTURE
- #FILTERS DOUBLE 64,128,256,512

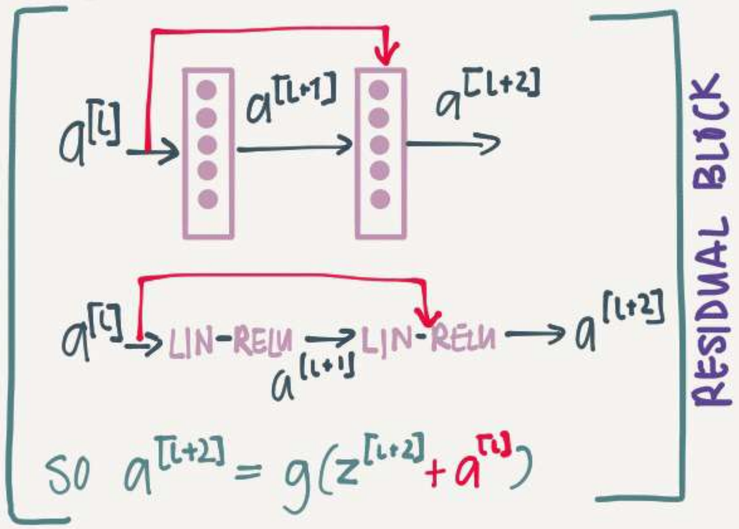
CONVOLUTIONAL NEURAL NETS · COURSERA

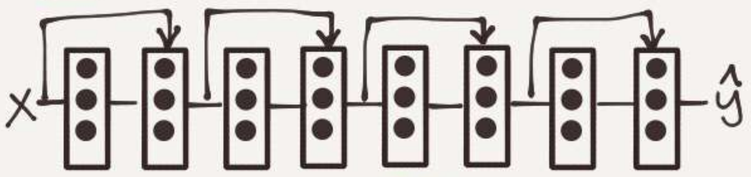
SPECIAL NETWORKS

ResNets

PROBLEM: DEEP NN OFTEN SUFFER PROBLEMS W VANISHING OR EXPLODING GRADIENTS

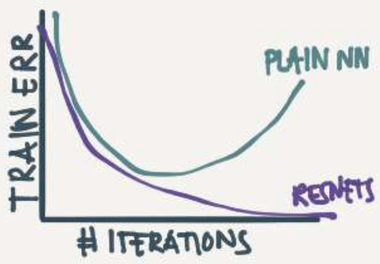
SOLUTION RESIDUAL NETS





VERY DEEP NN

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = (w^{[l+2]} a^{[l+1]} + b^{[l+1]} + a^{[l]})$$



WORST CASE: W & b HAVE VANISHED TO 0 ⇒

$a^{[l+2]} = g(a^{[l]}) = a^{[l]}$

SO EXTRA LAYERS LEARNED NOTHING BUT DIDN'T HURT

BEST CASE: LAYERS LEARNED SOMETHING

CONVOLUTIONAL NEURAL NETS · COURSERA

NETWORK IN NETWORK (1×1 CONVOLUTION)

6	5	3	2
4	1	9	5
5	8	2	4
0	3	6	1

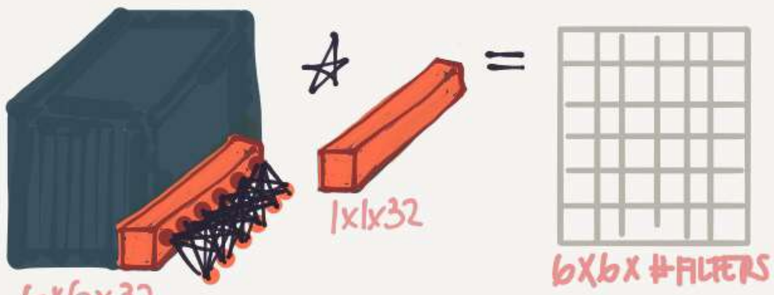
* [2] =

12	10	6	4
8	2	18	10
10	16	4	8
0	6	12	2

1×1 CONVOLUTION

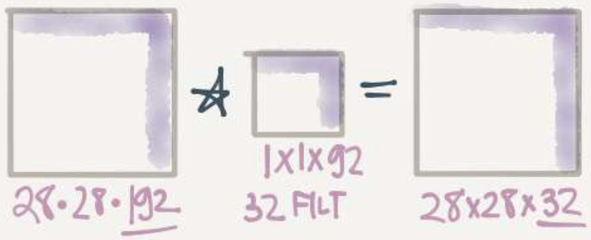
IT SEEMS PRETTY USELESS, BUT IT ACTUALLY SERVES 2 PURPOSES

1. NETWORK IN A NETWORK



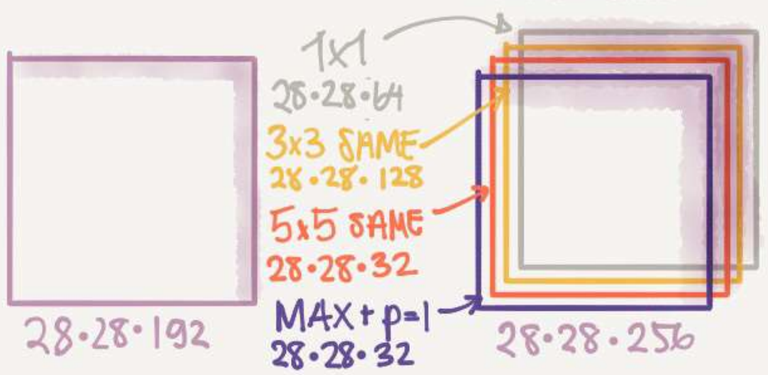
LEARNS COMPLEX, NON-LINEAR RELATIONSHIPS ABOUT A SLICE OF A VOLUME

2. REDUCING # CHANNELS



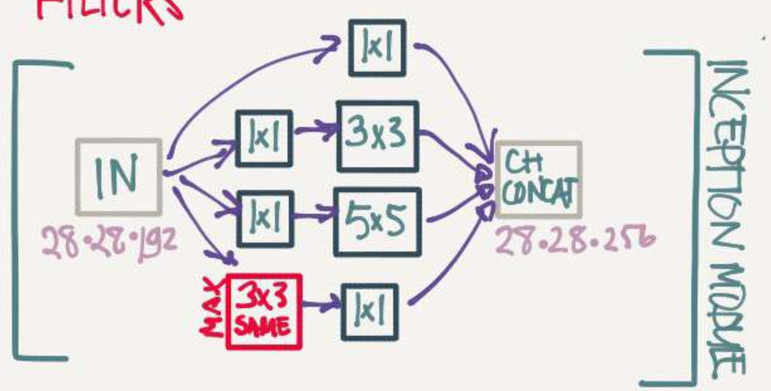
INCEPTION NETWORKS

INSTEAD OF CHOOSING A 1×1, 3×3, 5×5 OR A POOLING LAYER - CHOOSE ALL



PROBLEM: VERY EXPENSIVE TO COMPUTE

SOLUTION: SHRINK THE #CHANNELS W A 1×1 CONV BEFORE APPLYING ALL THE FILTERS



TO BUILD AN INCEPTION NETWORK YOU MAINLY STACK A BUNCH OF INCEPTION MODULES

INCEPTION THE MOVIE

PRACTICAL ADVICE

USE OPEN SOURCE IMPLEMENTATIONS

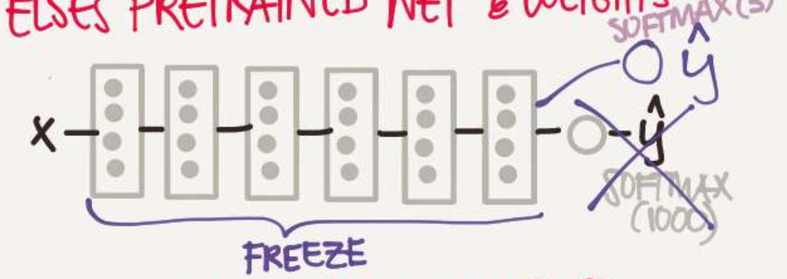
SOME OF THE PAPERS ARE HARD TO IMPLEMENT FROM SCRATCH - USING OS YOU CAN REUSE OTHER PPLS WORK

DON'T FORGET TO CONTRIBUTE

TRANSFER LEARNING

WANT TO TRAIN A CLASSIFIER FOR YOUR CATS BUT DON'T HAVE ENOUGH PICTURES

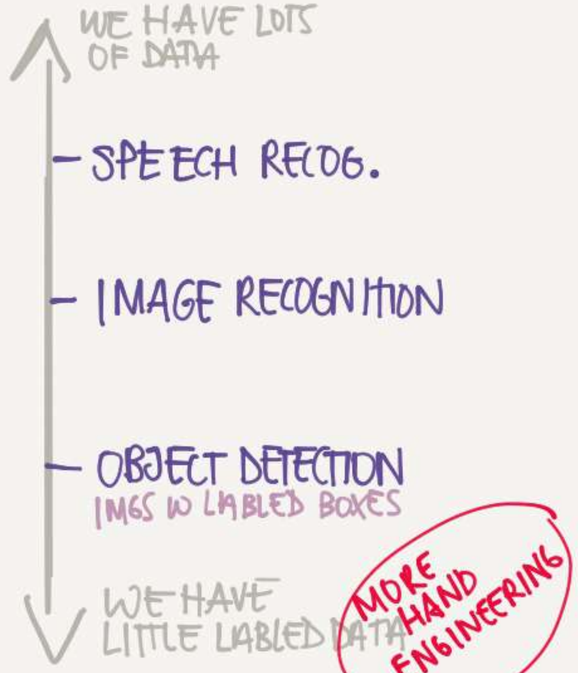
SOLUTION DOWNLOAD SOMEONE ELSES PRETRAINED NET & WEIGHTS



FREEZE THE PARAMS, AND JUST REPLACE THE SOFTMAX LAYER WITH YOUR OWN & TRAIN

IF YOU HAVE MORE PICS • RETRAIN A MORE OF THE LATER LAYERS (MAYBE INITIALIZING WITH THE PRETRAINED WEIGHTS)

STATE OF COMPUTER VISION



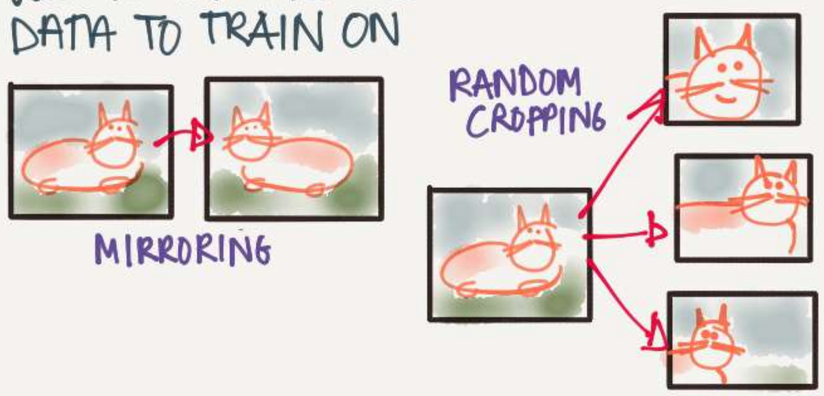
TIPS FOR DOING WELL ON BENCHMARKS/COMPETITIONS

- ENSEMBLING
  AVG OUTPUTS FROM MULT NN
- MULTI-CROP AT TEST TIME
  AVG OUTPUTS FROM MULTIPLE CROPS OF THE IMAGE

IN PRACTICE THEY ARE NOT USED IN PRODUCTION BECAUSE THEY ARE COMPUTE & MEM EXPENSIVE

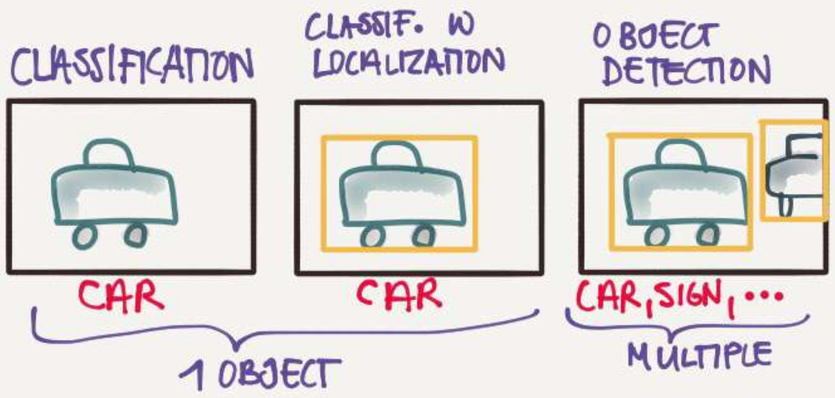
DATA AUGMENTATION

WE ALMOST ALWAYS NEED MORE DATA TO TRAIN ON

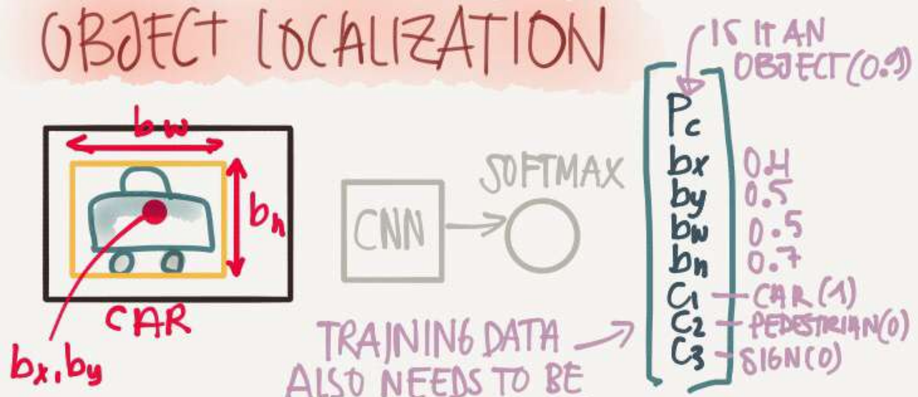


CONVOLUTIONAL NEURAL NETS · COURSERA

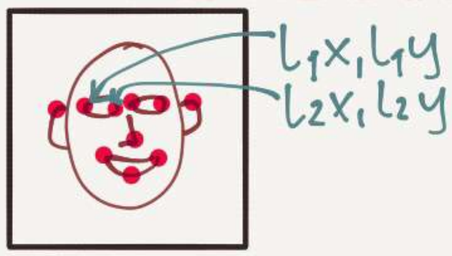
DETECTION ALGORITHMS



OBJECT LOCALIZATION



LANDMARK DETECTION



$$\hat{y} = \begin{bmatrix} \text{FACE} \\ L_1x \\ L_1y \\ L_2x \\ L_2y \\ \vdots \end{bmatrix}$$

TO DETECT LANDMARKS IN THE FACE (CORNER OF MOUTH ETC) LABEL THE X,Y COORDS OF THE LANDMARK

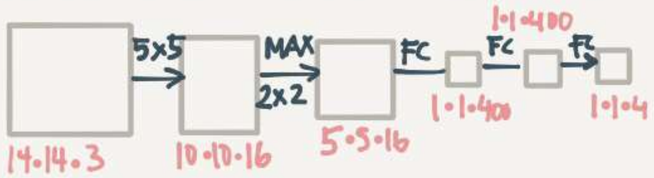
USED FOR SENTIMENT ANALYSIS & FOR EFFECTS LIKE PLACING CROWN ON HEAD ETC.

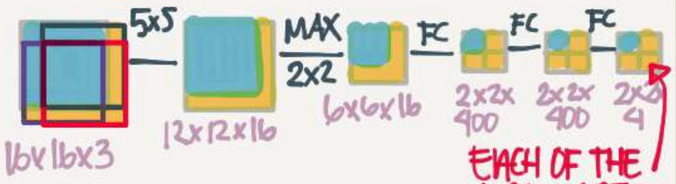
SLIDING WINDOWS DETECTION

1. CREATE TIGHTLY CROPPED IMG OF CARS (LOTS)
2. SLIDE A WINDOW OVER THE IMG. & CLASSIFY THIS WINDOW CAR(1/0) AGAINST YOUR OTHER CARS
3. REPEAT WITH SLIGHTLY LARGER WINDOW SIZE

PROBLEM: VERY EXPENSIVE (TO COMPUTE)

SINCE ADJ WINDOWS SHARE A LOT OF THE COMPUTATIONS WE CAN DO THIS MUCH CHEAPER W CONVOLUTIONS

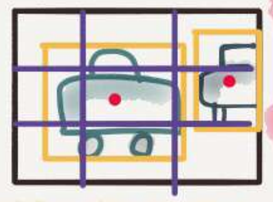




EACH OF THE 4 VALS ARE RESULTS FOR EACH OF THE 4 WINDOWS

NOW WE JUST PASS THROUGH ONCE AND CALC ALL AT THE SAME TIME

YOLO: You Only Look Once

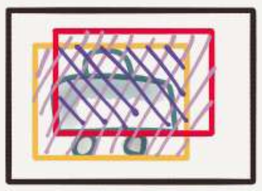


1. SPLIT IMG INTO X(9) GRID CELLS
2. FOR EACH CELL, SAY IF IT CONTAINS CAR + BOUNDING BOX (IF CELL CONTAINS THE MID POINT)
   SAME AS OBJECT LOCALIZATION

IN PRACTICE WE MIGHT HAVE A 19x19 GRID

$$3 \times 3 \times 8$$

HOW DO YOU KNOW HOW GOOD IT IS?



HOW GOOD IS THE RED SQUARE?

$$IOU = \frac{\text{SIZE OF}}{\text{SIZE OF}}$$

INTERSECTION OVER UNION

GENERALLY - IF $IOU \geq 0.5$ IT IS REGARDED AS CORRECT

WHAT IF MULTIPLE SQUARES CLAIM THE SAME CAR?

NON-MAX SUPRESSION

IF TWO BOUNDING BOXES HAVE A HIGH IOU - PICK THE ONE W HIGHEST P_c - GET RID OF THE REST.

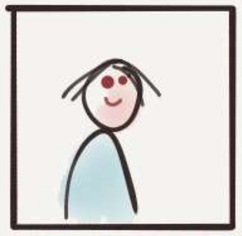
ANCHOR BOXES

ANCHOR BOXES LET YOU ENCODE MULTIPLE OBJECTS IN THE SAME SQUARE

CONVOLUTIONAL NEURAL NETS - COURSERA

FACE RECOGNITION

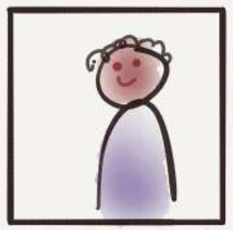
FACE VERIFICATION



IS THIS PETE?

99% ACC ⇒ PRETTY GOOD

FACE RECOGNITION



WHO IS THIS? (OUT OF K PERSONS)

IF K = 100 NEED MUCH HIGHER THAN 99%

ONE-SHOT LEARNING

NEED TO BE ABLE TO RECOGNIZE A PERSON EVEN THOUGH YOU ONLY HAVE ONE SAMPLE IN YOUR DB.

YOU CAN'T TRAIN A CNN WITH A SOFTMAX (EACH PERSON) BECAUSE

(A) YOU DON'T HAVE ENOUGH SAMPLES

(B) IF A NEW PERSON JOINS YOU NEED TO RETRAIN THE NETWORK

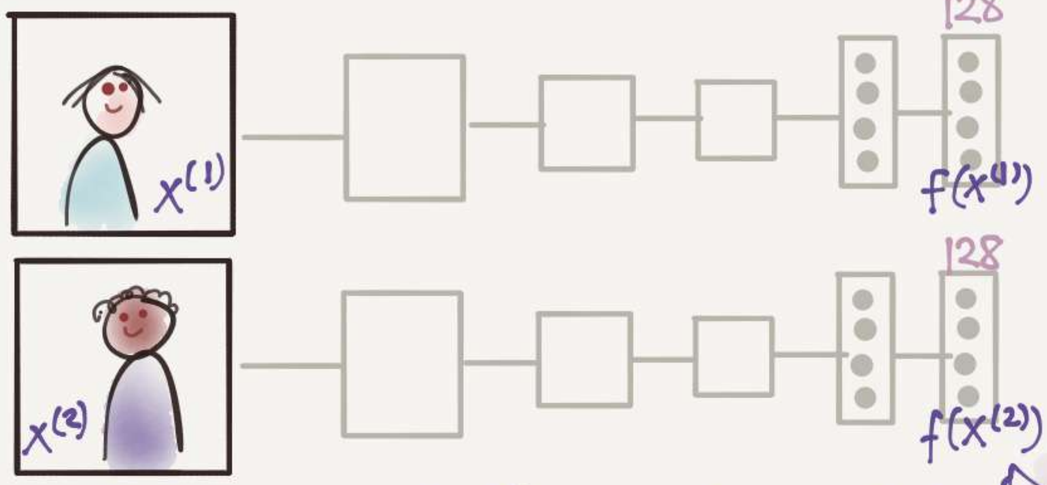
SOLUTION LEARN A SIMILARITY FUNCTION

$d(\text{img1}, \text{img2}) = \text{degree of difference}$

BUT HOW DO YOU LEARN THIS?

SIAMESE NETWORK

DeepFace



$$d(x^{(1)}, x^{(2)}) = \| f(x^{(1)}) - f(x^{(2)}) \|_2^2$$

LEARN THE PARAMS OF THE NN SUCH THAT

- IF $x^{(i)}, x^{(j)}$ ARE THE SAME PERSON · $d(x^i, x^j) \Rightarrow$ SMALL
- IF $x^{(i)}, x^{(j)}$ ARE DIFFERENT PEOPLE · $d(x^i, x^j) \Rightarrow$ LARGE

WE CAN ACCOMPLISH THIS WITH THE TRIPLET LOSS FUNCTION

TRIPLET LOSS

FaceNet

WANT $\underbrace{\|f(A)-f(P)\|^2}_{d(A,P)} \leq \underbrace{\|f(A)-f(N)\|^2}_{d(A,N)} \Rightarrow d(A,P) - d(A,N) \leq 0$

BUT WE WANT A GOOD MARGIN, SO...

$d(A,P) - d(A,N) + \alpha \leq 0$

TIP

PRECOMPUTE ENCODINGS FOR PPL IN YOUR DB, SO YOU DONT HAVE TO SAVE IMAGES & COMPUTE ENCODINGS AT RUN. TIME

HOW DO WE CHOOSE TRIPLETS TO TRAIN ON?

- IF A/P ARE VERY SIMILAR, & A/N ARE VERY DIFFERENT TRAINING IS VERY EASY.

SELECT A/N THAT ARE PRETTY SIMILAR TO TRAIN A GOOD NET

SOME BIG COMPANIES HAVE ALREADY TRAINED NETWORKS ON LARGE AMTS OF PHOTOS SO YOU MAY JUST WANT TO REUSE THEIR WEIGHTS

CONVOLUTIONAL NEURAL NETS · COURSERA

NEURAL STYLE TRANSFER

WE CAN VISUALIZE WHAT A NETWORK LEARNS BY LOOKING AT WHAT IMAGES (PARTS) ACTIVATED EACH UNIT MOST

BUT HOW DOES THIS HELP US GENERATE AN IMAGE IN THE STYLE OF ANOTHER?

IDEA:

1. GENERATE A RANDOM IMG
2. OPTIMIZE THE COST FUNCTION

$$J(G) = \alpha J_{CONTENT}(C,G) + \beta J_{STYLE}(S,G)$$

$J_{CONTENT}(C,G)$: HOW SIMILAR ARE C & G

$J_{STYLE}(S,G)$: HOW SIMILAR ARE S & G

3. UPDATE EACH PIXEL

CONTENT COST FUNCTION

- USE A PRE-TRAINED CONVNET (EX VGG)
- SELECT A HIDDEN LAYER SOMEWHERE IN THE MIDDLE
  LATER ⇒ COPIES LARGER FEATURES
- LET $a^{[l](C)}$ & $a^{[l](G)}$ BE THE ACTIVATIONS
- IF $a^{[l](C)}$ & $a^{[l](G)}$ ARE SIMILAR THEY HAVE SIMILAR CONTENT
  BECAUSE THEY BOTH TRIGGER THE SAME HIDDEN UNITS

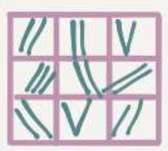
HOW DO WE TELL IF THEY ARE SIMILAR?

$$J_{CONTENT}(C,G) = \frac{1}{2} \| a^{[l](C)} - a^{[l](G)} \|^2$$

CAPTURING THE STYLE

USING THE STYLE IMG AND THE ACTIVATIONS IN A LAYER.
LOOK THROUGH THE ACTIVATIONS IN THE DIFFERENT CHANNELS TO SEE HOW CORRELATED THEY ARE

WHEN WE SEE PATTERNS LIKE THIS



DO WE USUALLY SEE IT WITH PATCHES LIKE THESE?

STYLE MATRIX

CREATE A MATRIX OF HOW CORRELATED THE ACTIVATIONS ARE, FOR EACH POS (x,y) & CHANNEL PAIR (K, K') FOR THE STYLE IMG & GENERATED

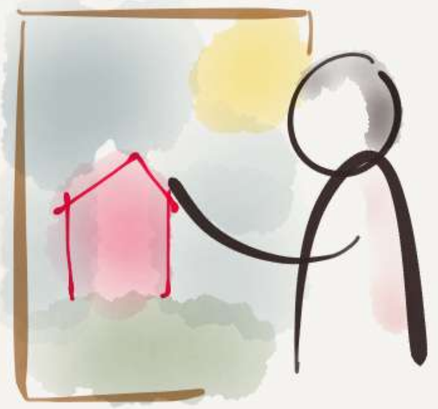
$$G_{KK'} = \sum_{i=1}^{n_H} \sum_{j=1}^{n_W} a_{ijk} \cdot a_{ijk'}$$

THE STYLE COST FUNCTION

$$J(S,G) = \| G^{(S)} - G^{(G)} \|_F^2$$

FROBENIUS NORM

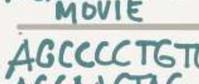
TO GET MORE VISUALLY PLEASING IMAGES IF YOU CALC J(S,G) OVER MULTIPLE LAYERS



RECURRENT
NEURAL NETWORKS

SEQUENCE MODELS · COURSERA

SEQUENCE PROBLEMS

IN	OUT	PURPOSE
	THE QUICK BROWN FOX JUMPED...	SPEECH RECOGNITION
∅		MUSIC GENERATION
THERE IS NOTHING TO LIKE IN THIS MOVIE	★ ★ ★ ★ ★	SENTIMENT CLASSIFICATION
AGCCCCTGTG AGGAACTAG	AGCCCCTGTG AGGAACTAG	DNA SEQUENCE ANALYSIS
Voulez-vous chanter avec moi?	Do you want to sing with me?	MACHINE TRANSLATION
	RUNNING	VIDEO ACTIVITY RECOGNITION
Yesterday Harry Potter met Hermoine Granger	Yesterday Harry Potter met Hermoine Granger	NAME ENTITY RECOGNITION

NAME ENTITY RECOGNITION

X = HARRY POTTER AND HERMOINE $T_x = 9$ (9 WORDS)
$x^{<1>}$ $x^{<2>}$ ...
GRANGER INVENTED A NEW SPELL

y = 1 1 0 1 $T_y = T_x$
$y^{<1>}$ $y^{<2>}$ ...
1 0 0 0 0

EXAMPLE OF A PROBLEM WHERE EVERY $x^{<i>}$ HAS AN OUTPUT $y^{<i>}$

HOW DO WE REPRESENT WORDS?

CREATE A VOCABULARY (EG 10K MOST COMMON WORDS IN YOUR TEXTS · OR DOWNLOAD EXISTING)

a	1
aaron	2
and	367
harry	4075
potter	6830
zulu	10000

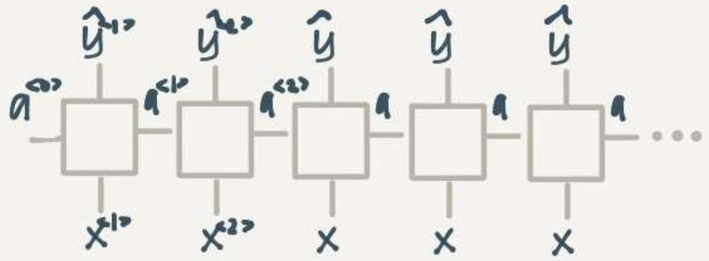
EACH WORD IS A ONE-HOT VECTOR

HARRY = [0 0 ... 1 ... 0] (1 at 4075)

WE COULD USE A STANDARD NETWORK BUT...

(A) INPUT & OUTPUTS CAN HAVE DIFFERENT LENGTHS IN DIFF EXAMPLES
(B) WE DON'T SHARE FEATURES LEARNED ACROSS DIFFERENT POSITIONS

RECURRENT NEURAL NET (RNN)



PREVIOUS RESULTS ARE PASSED IN AS INPUTS SO WE GET CONTEXT.

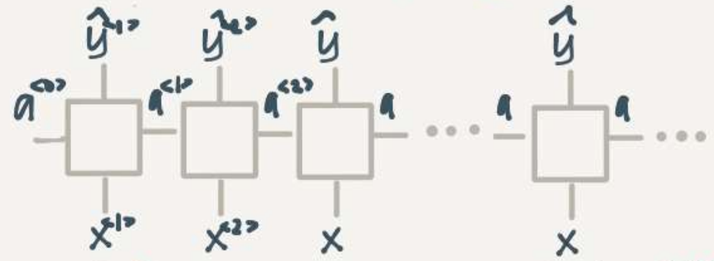
$a^{<1>} = g_1(W_a[a^{<0>}, x^{<1>}] + b_a)$ TANH/RELU
$\hat{y}^{<1>} = g_2(W_{ya} a^{<1>} + b_y)$ SIGMOID

THE SAME W & b ARE USED IN ALL TIME STEPS

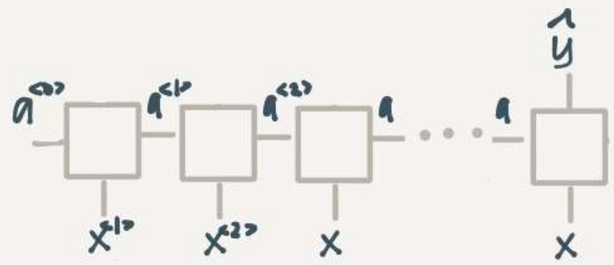
THE LOSS WE OPTIMIZE IS THE SUM OF $\mathcal{L}(\hat{y}, y)$ FROM 1–T

DIFFERENT TYPES OF RNN

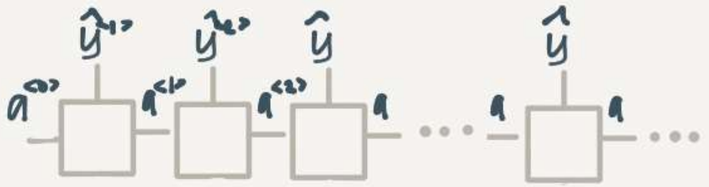
MANY-TO-MANY $T_x = T_y$



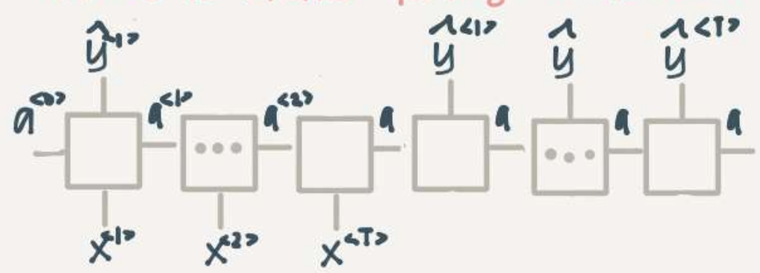
MANY-TO-ONE EX. SENTIMENT ANALYSIS



ONE-TO-MANY · MUSIC GENERATION



MANY-TO-MANY $T_x \neq T_y$ TRANSLATION



MORE ON RNNs

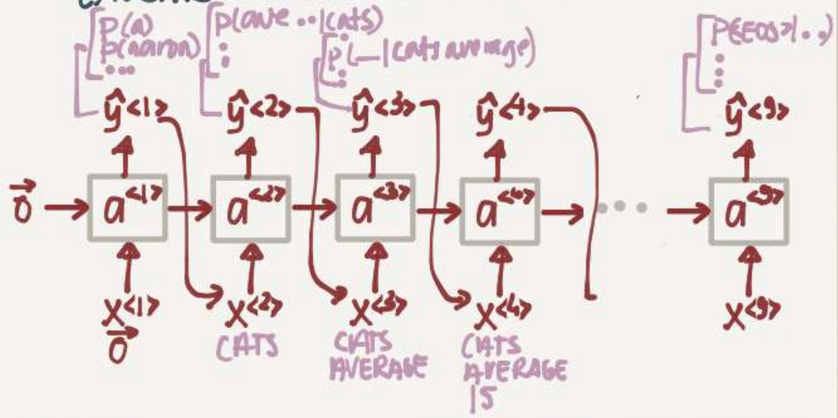
SEQUENCE MODELS · COURSERA

LANGUAGE MODELLING

HOW DO YOU KNOW IF SOMEONE SAID
THE APPLE AND PAIR SALAD OR
THE APPLE AND PEAR SALAD?

THE PURPOSE OF A LANG. MODEL IS TO CALCULATE THE PROBABILITIES

EX. CATS AVERAGE 15 HOURS OF SLEEP A DAY



SO GIVEN: CATS AVERAGE 15 WHAT IS THE PROB. THE NEXT WORD IS HOURS?

SAMPLING SENTENCES

1. TRAIN ON ALL HARRY POTTER BOOKS.
2. RANDOMLY SELECT A WORD (ONE OF THE TOP WORDS) (EX. THE)
3. PASS THIS INTO THE NEXT TIMESTAMP AND SAMPLE A NEW WORD
4. REPEAT UNTIL X WORDS OR YOU REACHED <EOS>

CAN DO AT CHARACTER LEVEL AS WELL

YAY! YOU ARE NOW YOUR OWN J.K. ROWLING

VANISHING GRADIENTS

THE CAT, WHO ALREADY ATE APPLES AND ORANGES AND A FEW MORE THINGS BLA BLA WAS FULL
THE CATS, WHO ALREADY ATE ... WERE FULL

NEED TO REMEMBER SING/PLURAL FOR A LONG TIME

SINCE LONG SENTENCE ⇒ DEEP RNN
WE GET THE VANISHING GRADIENTS PROB WE HAVE IN STANDARD NNs – I.E THE GRADIENTS FOR CAT/CATS HAVE LITTLE OR NO EFFECT ON WAS/WERE.

NOTE SOMETIMES YOU SEE EXPLODING GRAD (AS OVERFLOW NaN) BUT THIS IS EASILY FIXED WITH GRADIENT CLIPPING

GATED RECURRENT UNIT GRU

HELPS RECALL IF CAT WAS SING. OR PLURAL

GRU

THE GRU ACTS AS A MEMORY — AT EVERY TIMESTEP IT CALCULATES A NEW $\tilde{c}$ TO STORE AND A GATE Γ_u DECIDES TO UPDATE c TO $\tilde{c}$ OR NOT

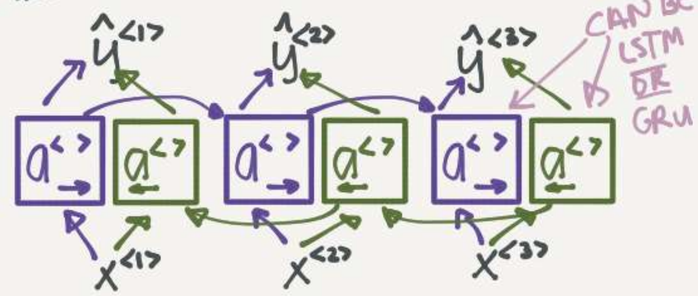
LONG SHORT TERM MEMORY (LSTM)

THE LSTM IS A VARIATION ON THE SAME THEME AS GRU BUT WITH AN ADDITIONAL Γ_F FORGET GATE

BI-DIRECTIONAL RNNs (BRNN)

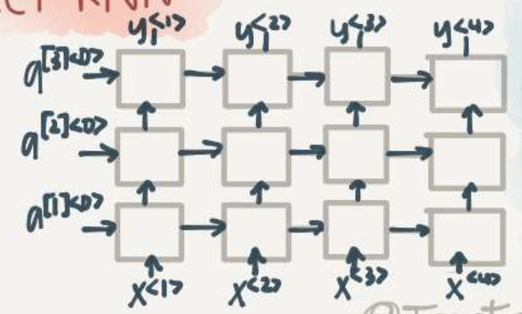
HE SAID, 'TEDDY BEARS ARE ON SALE'
HE SAID, 'TEDDY ROOSEVELT WAS A GREAT PRESIDENT'

PROBLEM: WITHOUT LOOKING FORWARD WE CANT SAY IF TEDDY IS A TOY OR A NAME



ONE DISADVANTAGE IS THAT YOU NEED THE FULL SENTENCE BEFORE YOU BEGIN - SO NOT SUITABLE FOR LIVE SPEECH RECO

DEEP RNN



SINCE THEY ARE ALREADY TEMORALLY DEEP THEY USUALLY DON'T HAVE A LOT OF LAYERS

SEQUENCE MODELS · COURSERA

NLP & WORD EMBEDDINGS

MAN IS TO WOMAN AS KING IS TO QUEEN

PROBLEM: THE ONE-HOT REPR O_{4914} OF APPLE HAS NO INFO ABOUT ITS RELATIONSHIP TO O_{6257} ORANGE

I WANT A GLASS OF ORANGE ____
I WANT A GLASS OF APPLE ____

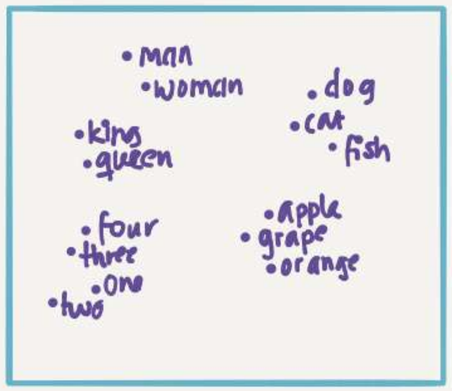
SOLUTION: CREATE A MATRIX OF FEATURES TO DESCRIBE THE WORDS

WORD EMBEDDINGS

	MAN 5391	WOMAN 9853	KING 4914	QUEEN 7157	APPLE 456	ORANGE 6257
GENDER	-1	1	-0.95	0.97	0.00	0.01
ROYAL	0.01	0.02	0.93	0.95	-0.01	0.00
AGE	0.03	0.02	0.7	0.69	0.03	-0.02
FOOD	0.04	0.01	0.02	0.01	0.95	0.97
⋮						

e_{5391}

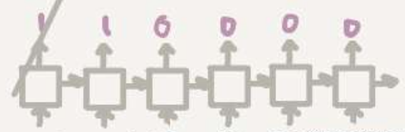
IN REALITY · THE FEATURES ARE LEARNED & NOT AS STRAIGHTFWD AS GENDER/AGE



t-SNE
VISUAL REPRESENT OF 300D WORD EMBEDDINGS

USING WORD EMBEDDINGS

EX. NAME/ENTITY RECOGN



SALLY SMITH IS AN ORANGE FARMER
ROBERT LIN IS AN APPLE FARMER
BOB JONES IS A DURIAN CULTIVATOR

WITH WORD EMBEDDINGS WE UNDERSTAND THAT AN ORANGE FARMER IS A PERSON ⇒ SALLY SMITH = NAME

- APPLE ~ ORANGE ⇒ APPLE FARMER = PERSON
- USING WORD EMBEDDINGS TRAINED ON LOTS OF TEXT WE ALSO GET EMB. FOR MORE UNCOMMON WORDS (DURIAN, CULTIVATOR)

EX. MAN IS TO WOMAN AS KING IS TO ?

E = EMBEDDING MATRIX

	MAN 5391	WOMAN 9853	KING 4914	QUEEN 7157
GENDER	-1	1	-0.95	0.97
ROYAL	0.01	0.02	0.93	0.95
AGE	0.03	0.02	0.7	0.69
FOOD	0.04	0.01	0.02	0.01

e_{man}

$e_{man} - e_{woman}$ $e_{king} - e_{queen}$

$\begin{bmatrix}-2\\0\\0\\0\end{bmatrix}$ VERY SIMILAR $\begin{bmatrix}-2\\0\\0\\0\end{bmatrix}$

FIND MAN→WOMAN
KING→ ?

FIND(w):
$ARG.MAX\ SIM(e_w, e_{KING} - e_{MAN} + e_{WOMAN})$

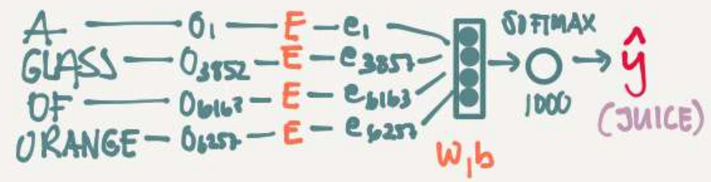
$SIM(u,v) = \frac{u^T v}{\|u\|_2 \|v\|_2}$ COSINE SIMILARITY

LEARNING WORD EMBEDDINGS

HOW DO WE LEARN THE EMBEDDING MATRIX E?

IDEA 1: USING A NEURAL LANG MODEL

I WANT A GLASS OF ORANGE $\hat{y}$



WE CAN USE DIFFERENT CONTEXTS THAN THE LAST 4 WORDS

- LAST 4 WORDS
- 4 WORDS LEFT+RIGHT
- LAST 1 WORD
- NEARBY 1 WORD ← SKIPGRAM RANDOM WITHIN EX 5 WORDS

IDEA 2: SKIP-GRAMS WORD2VEC

I WANT A GLASS OF ORANGE JUICE TO GO ALONG WITH MY CEREAL

PICK RANDOM CONTEXT/TARGET PAIRS (WITHIN EX 5 WORDS)

CONTEXT	TARGET
ORANGE	JUICE
ORANGE	GLASS
ORANGE	MY
...	...

$O_c \to E \to e_c \to$ SOFTMAX $\to \hat{y}(O_T)$ 1000

NOTE: WHILE THIS SIMPLE NN PREDICTS O_T OUR REAL GOAL IS TO LEARN E

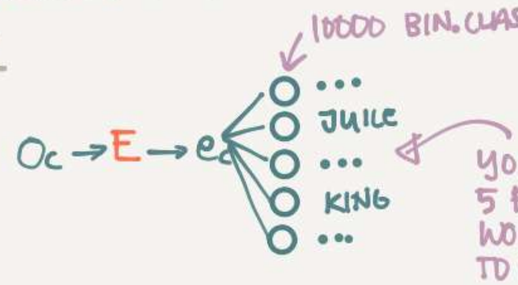
THIS IS VERY COMPUTATIONALLY EXPENSIVE BUT WE CAN OPTIMIZE BY USING A HIERARCHICAL SOFTMAX CLASSIFIER

IDEA: NEGATIVE SAMPLING

1. PICK A CONTEXT/TARGET PAIR AS A POSITIVE EXAMPLE
2. PICK A FEW NEG EXAMPLES CONTEXT + RANDOM

CONTEXT	WORD	TARGET
ORANGE	JUICE	1
ORANGE	KING	0
ORANGE	BOOK	0
ORANGE	THE	0
ORANGE	OF	0

NOTE: SOMETIMES BY CHANCE YOU PICK A POS PAIR · BUT IT DOESN'T MATTER



SEQUENCE MODELS · COURSERA

WORD EMBEDDINGS CONTINUED...

GloVe WORD VECTORS

X_{ij} = #TIMES WORD i (TARGET t) APPEARS IN THE CONTEXT OF j (CONTEXT c)
(HOW RELATED THEY ARE)

$$\text{MINIMIZE} \sum_{i=1}^{10k} \sum_{j=1}^{10k} f(X_{ij})(\theta_i^T e_j + b_i + b_j' - \log X_{ij})^2$$

$f(X_{ij})$: 0 IF NO CONTEXT
(ALSO HELPS WEIGHING VERY FREQ WORDS (THE, OF...) & VERY INFREQUENT (DURIAN))

EVERYTHING LED UP TO THIS VERY SIMPLE ALGORITHM

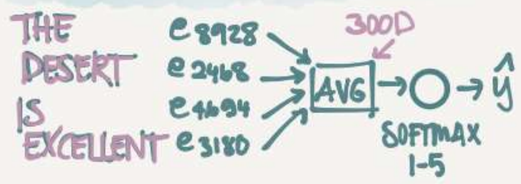
SENTIMENT CLASSIFICATION

X	y
THE DESSERT IS EXCELLENT	★★★★
SERVICE WAS QUITE SLOW	★★
GOOD FOR A QUICK MEAL BUT NOTHING SPECIAL	★★★
COMPLETELY LACKING IN GOOD TASTE GOOD SERVICE AND GOOD AMBIENCE	★

PROBLEM: YOU MAY NOT HAVE A LARGE DATASET

BUT YOU CAN USE AN EMBEDDING MATRIX E THAT IS ALREADY PRE-TRAINED

IDEA: SIMPLE CLASSIFICATION

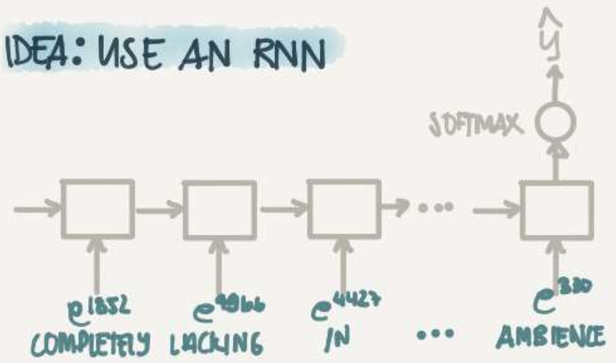


WORKS WELL FOR SHORT SENTENCES BUT DOESN'T TAKE ORDER INTO ACCOUNT

"COMPLETELY LACKING IN GOOD TASTE, GOOD SERVICE AND GOOD AMBIENCE"

THIS MAY BE SEEN AS A +++ REVIEW

IDEA: USE AN RNN



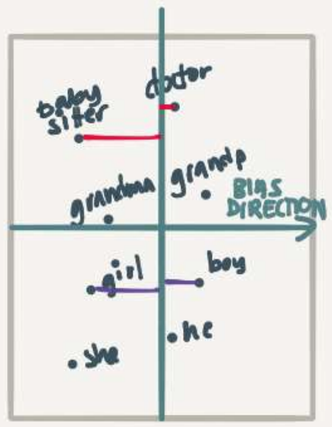
THIS CAN NOW TAKE INTO ACCOUNT THAT COMPLETELY LACKING NEGATES THE WORD GOOD

ELIMINATING BIAS IN WORD EMBEDDINGS

MAN IS TO COMPUTER PROGRAMMER AS WOMAN IS TO HOME MAKER

SOMETIMES THE TEXT CONTAINS & ALGOS LEARN A GENDER, RACE, AGE... BIAS WE DON'T WANT OUR MODELS TO HAVE · EX. HIRING BASED ON GENDER, SENTENCING BASED ON RACE ETC.

ADDRESSING BIAS



1. IDENTIFY BIAS DIRECTION
   $\{e_{he} \rightarrow e_{she}$
   $e_{male} \rightarrow e_{female}$
2. NEUTRALIZE
   FOR EVERY WORD THAT IS NOT DEFINITIONAL (GIRL, BOY, HE, SHE...) PROJECT TO GET RID OF BIAS
3. EQUALIZE PAIRS
   THE ONLY DIFF BETWEEN EX GIRL/BOY SHOULD BE GENDER

HOW DO YOU KNOW WHICH WORDS TO NEUTRALIZE?

DOCTOR, BEARD, SEWING MACHINE?

A: BY TRAINING A CLASSIFIER TO FIND OUT IF A WORD IS DEFINITIONAL

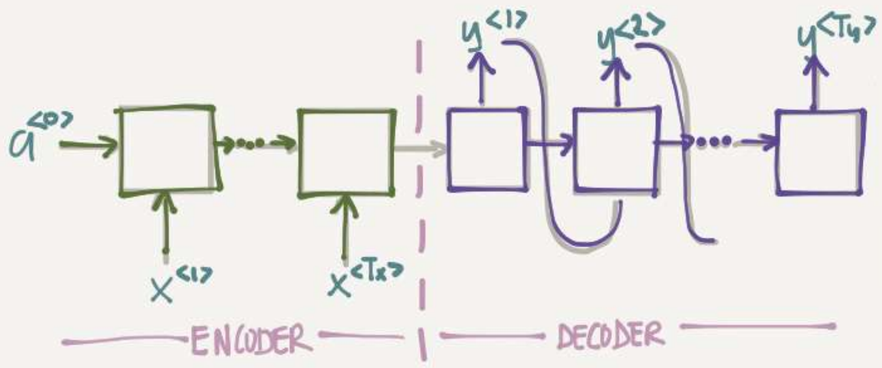
TURNS OUT THE # OF PAIRS IS FAIRLY SMALL SO YOU CAN EVE HAND PICK THEM

SEQUENCE TO SEQUENCE

SEQUENCE MODELS · COURSERA

BASIC MODELS

JANE VISITE L'AFRICE EN SEPTEMBRE → JANE IS VISITING AFRICA IN SEPTEMBER



 → THIS IS A CAT ON A CHAIR

CNN → RNN

HOW DO YOU PICK THE MOST LIKELY SENTENCE?

$$P(y^{\langle 1\rangle}, \dots, y^{\langle T_y\rangle} | x)$$
(ENGLISH, FRENCH)

WE DON'T WANT A RANDOMLY GENERATED SENTENCE
(WE WOULD SOMETIMES GET A GOOD, SOMETIMES BAD)
INSTEAD WE WANT TO MAXIMIZE

$$\text{ARG MAX } P(y^{\langle 1\rangle}, \dots, y^{\langle T_y\rangle} | x)$$

IDEA: USE GREEDY SEARCH

1. PICK THE WORD WITH THE BEST PROBABILITY
2. REPEAT UNTIL <EOS>

WITH THIS WE COULD GET
- JANE IS GOING TO BE VISITING AFRICA THIS SEPTEMBER

INSTEAD OF
- JANE IS VISITING AFRICA THIS SEPTEMBER

SOLUTION: OPTIMISE THE PROB OF THE WHOLE SENTENCE INSTEAD

BEAM SEARCH

1. PICK THE FIRST WORD
   PICK THE B (EX 3) BEST ALTERNATIVES
   (IN, JANE, SEPTEMBER)
2. FOR EACH B WORDS PICK THE NEXT WORD AND EVALUATE THE PAIRS TO END UP W B PAIRS
   $P(y^{\langle 1\rangle}, y^{\langle 2\rangle} | x) = P(y^{\langle 1\rangle} | x)\, P(y^{\langle 2\rangle} | x, y^{\langle 1\rangle})$

(IN SEPTEMBER, JANE IS, JANE VISITS)

NOTE: KEEP TRACK OF THE P FOR THE SENTENCES OF EACH LENGTH - AFTER X ITER (X MAX WORDS) PICK THE BEST

3. REPEAT TIL DONE

$$\text{ARG MAX} \prod_{t=1}^{T_y} P(y^{\langle t\rangle} | x, y^{\langle 1\rangle}, \dots y^{\langle t-1\rangle})$$

PROBLEM: MULTIPLYING PROBABILITIES (0<P<1) RESULTS IN A VERY SMALL NUMBER → OVERFLOWS

PROBLEM II: IF WE OPTIMIZE FOR THE MULT WE WILL PREFER SHORT SENTENCES. SINCE EACH WORD WILL REDUCE PROB

INSTEAD WE CAN OPTIMIZE FOR THIS

$$\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{\langle t\rangle} | x, y^{\langle 1\rangle}, \dots, y^{\langle t-1\rangle})$$

HOW DO WE PICK B?

LARGE B: BETTER RESULT, SLOWER
SMALL B: WORSE RESULT, BETTER

IN PROD YOU MIGHT SE B=10.
100 IS PROBABLY A BIT TOO HIGH - BUT ITS DOMAIN DEPENDENT

ERROR ANALYSIS IN BEAM S.

HUMAN: JANE VISITS AFRICA IN SEPT... y^*
ALGO: JANE VISITED AFRICA LAST SEPTEMBER $\hat{y}$

HOW DO WE KNOW IF ITS OUR RNN OR OUR BEAM SEARCH WE SHOULD WORK ON?

LET THE RNN GIVE $Py^* = P(y^*, x)$ & $P\hat{y} = P(\hat{y}, x)$

IF $Py^* > P\hat{y}$:
BEAM PICKED THE WRONG ONE
TRY A HIGHER B

ELSE:
THE RNN PICKED THE WRONG PROBS · SO FOCUS ON THE RNN

@Tessferrandez

SEQUENCE TO SEQUENCE

FRENCH: LE CHAT EST SUR LE TAPIS
HUMAN1: THE CAT IS ON THE MAT
HUMAN2: THERE IS A CAT ON THE MAT

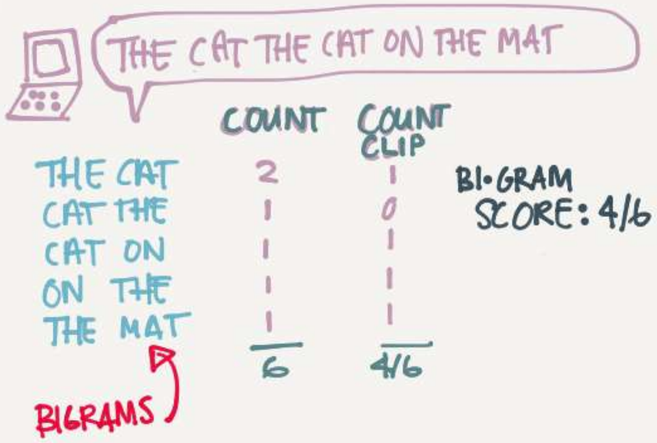
HOW DO YOU EVALUATE THE MACHINE TRANSLATION WHEN MULTIPLE ARE RIGHT?

BLEU SCORE

IDEA: CHECK IF THE WORDS (MT) APPEAR IN THE REAL TRANSLATION

IDEA: ONLY GIVE CREDIT FOR A WORD THE MAX # TIMES IT APPEARS IN A TARGET SENTENCE (COUNT CLIP)

SCORE: 2/7



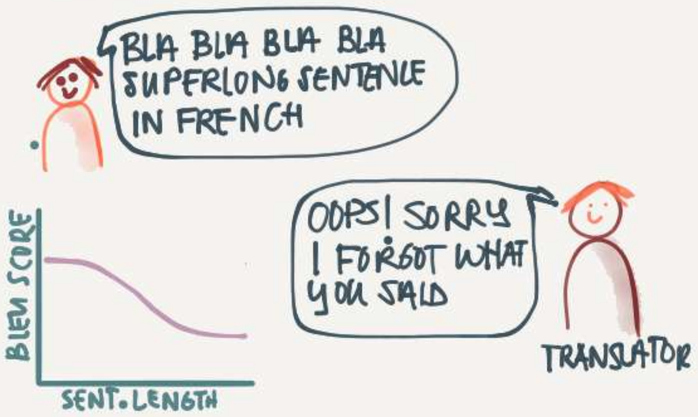
COMBINED BLEU SCORE

$BP \cdot \exp\left(\frac{1}{4}\sum_{n=1}^{4} P_n\right)$

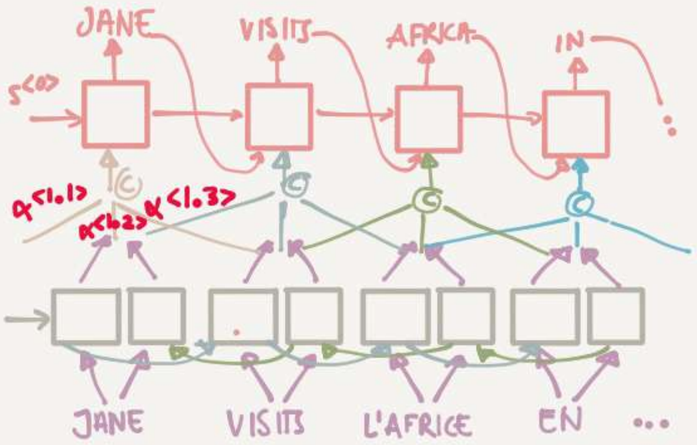
P_1 = SCORE SINGLE WORD
P_2 = SORE BIGRAMS
...
BP = BREVITY PENALTY (PENALIZES SENTENCES SHORTER THAN THE TARGET)

A USEFUL SINGLE NUMBER EVAL METRIC

ATTENTION MODEL



SOLUTION: TRANSLATE A LITTLE AT A TIME USING ONLY PARTS OF THE SENTENCE AS CONTEXT



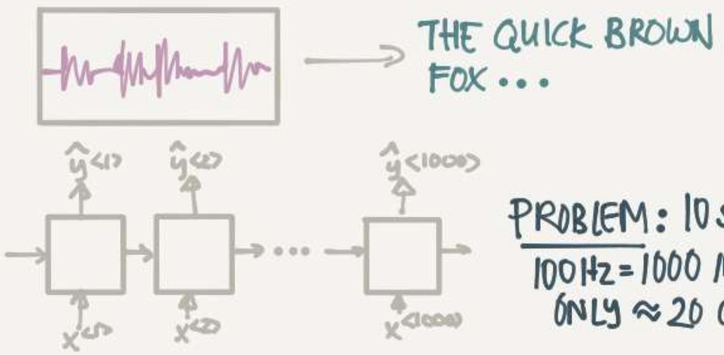
$\alpha^{<t,t'>}$ = HOW MUCH ATTENTION $y^{<t>}$ SHOULD PAY TO $a^{<t'>}$

$$c^{<2>} = \sum_{t'} \alpha^{<2,t'>} \cdot a^{<t'>} \quad \Big| \quad \sum_{t'} \alpha^{<1,t'>} = 1$$

α IS CALCULATED USING A SMALL NEURAL NETWORK

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t'=1}^{T_x} \exp(e^{<t,t'>})}$$

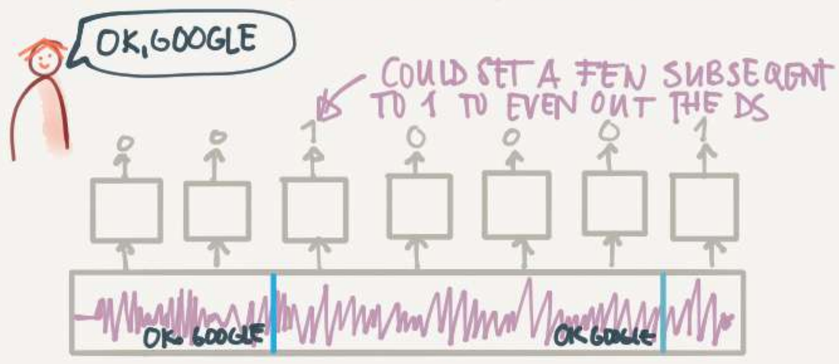
SPEECH RECOGNITION



SOLUTION: USE CTC COST (CONNECTION TEMPORAL CLASSIFICATION)

ttt_h_eee___u___qqq__ → COLLAPSE REPEATED CHARS NOT SEP BY BLANK

TRIGGER WORD DETECTION



SEQUENCE MODELS · COURSERA

A Selective Overview of Deep Learning

Jianqing Fan* Cong Ma‡ Yiqiao Zhong*

April 16, 2019

Abstract

Deep learning has arguably achieved tremendous success in recent years. In simple words, deep learning uses the composition of many nonlinear functions to model the complex dependency between input features and labels. While neural networks have a long history, recent advances have greatly improved their performance in computer vision, natural language processing, etc. From the statistical and scientific perspective, it is natural to ask: What is deep learning? What are the new characteristics of deep learning, compared with classical methods? What are the theoretical foundations of deep learning?

To answer these questions, we introduce common neural network models (e.g., convolutional neural nets, recurrent neural nets, generative adversarial nets) and training techniques (e.g., stochastic gradient descent, dropout, batch normalization) from a statistical point of view. Along the way, we highlight new characteristics of deep learning (including depth and over-parametrization) and explain their practical and theoretical benefits. We also sample recent results on theories of deep learning, many of which are only suggestive. While a complete understanding of deep learning remains elusive, we hope that our perspectives and discussions serve as a stimulus for new statistical research.

Keywords: neural networks, over-parametrization, stochastic gradient descent, approximation theory, generalization error.

Contents

Author names are sorted alphabetically.

*Department of Operations Research and Financial Engineering, Princeton University, Princeton, NJ 08544, USA; Email: {jqfan, congm, yiqiaoz}@princeton.edu.

1 Introduction

Modern machine learning and statistics deal with the problem of *learning from data*: given a training dataset $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ where $\boldsymbol{x}_i \in \mathbb{R}^d$ is the input and $y_i \in \mathbb{R}$ is the output[1], one seeks a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ from a certain function class $\mathcal{F}$ that has good prediction performance on test data. This problem is of fundamental significance and finds applications in numerous scenarios. For instance, in image recognition, the input $\boldsymbol{x}$ (reps. the output y) corresponds to the raw image (reps. its category) and the goal is to find a mapping $f(\cdot)$ that can classify future images accurately. Decades of research efforts in statistical machine learning have been devoted to developing methods to find $f(\cdot)$ efficiently with provable guarantees. Prominent examples include linear classifiers (e.g., linear / logistic regression, linear discriminant analysis), kernel methods (e.g., support vector machines), tree-based methods (e.g., decision trees, random forests), nonparametric regression (e.g., nearest neighbors, local kernel smoothing), etc. Roughly speaking, each aforementioned method corresponds to a different function class $\mathcal{F}$ from which the final classifier $f(\cdot)$ is chosen.

Deep learning [70], in its simplest form, proposes the following *compositional* function class:

$$\left\{f(\boldsymbol{x};\boldsymbol{\theta}) = \mathbf{W}_L\boldsymbol{\sigma}_L(\mathbf{W}_{L-1}\cdots\boldsymbol{\sigma}_2(\mathbf{W}_2\boldsymbol{\sigma}_1(\mathbf{W}_1\boldsymbol{x}))) \;\middle|\; \boldsymbol{\theta} = \{\mathbf{W}_1,\ldots,\mathbf{W}_L\}\right\}. \tag{1}$$

Here, for each $1 \le l \le L$, $\boldsymbol{\sigma}_\ell(\cdot)$ is some nonlinear function, and $\boldsymbol{\theta} = \{\mathbf{W}_1,\ldots,\mathbf{W}_L\}$ consists of matrices with appropriate sizes. Though simple, deep learning has made significant progress towards addressing the problem of learning from data over the past decade. Specifically, it has performed close to or better than humans in various important tasks in artificial intelligence, including image recognition [50], game playing [114], and machine translation [132]. Owing to its great promise, the impact of deep learning is also growing rapidly in areas beyond artificial intelligence; examples include statistics [15, 111, 76, 104, 41], applied mathematics [130, 22], clinical research [28], etc.

Table 1: Winning models for ILSVRC image classification challenge.

Model	Year	# Layers	# Params	Top-5 error
Shallow	< 2012	—	—	$> 25\%$
AlexNet	2012	8	61M	16.4%
VGG19	2014	19	144M	7.3%
GoogleNet	2014	22	7M	6.7%
ResNet-152	2015	152	60M	3.6%

To get a better idea of the success of deep learning, let us take the ImageNet Challenge [107] (also known as ILSVRC) as an example. In the classification task, one is given a training dataset consisting of 1.2 million color images with 1000 categories, and the goal is to classify images based on the input pixels. The performance of a classifier is then evaluated on a test dataset of 100 thousand images, and in the end the top-5 error[2] is reported. Table 1 highlights a few popular models and their corresponding performance. As

[1] When the label y is given, this problem is often known as *supervised learning*. We mainly focus on this paradigm throughout this paper and remark sparingly on its counterpart, *unsupervised learning*, where y is not given.

[2] The algorithm makes an error if the true label is not contained in the 5 predictions made by the algorithm.

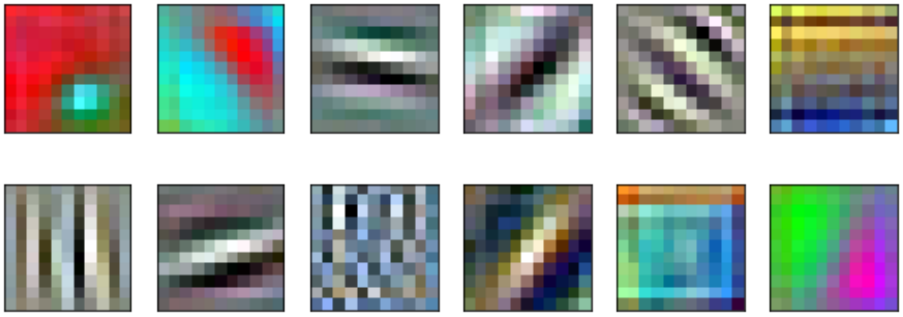


Figure 1: Visualization of trained filters in the first layer of AlexNet. The model is pre-trained on ImageNet and is downloadable via PyTorch package `torchvision.models`. Each filter contains $11 \times 11 \times 3$ parameters and is shown as an RGB color map of size 11×11.

can be seen, deep learning models (the second to the last rows) have a clear edge over shallow models (the first row) that fit linear models / tree-based models on handcrafted features. This significant improvement raises a foundational question:

Why is deep learning better than classical methods on tasks like image recognition?

1.1 Intriguing new characteristics of deep learning

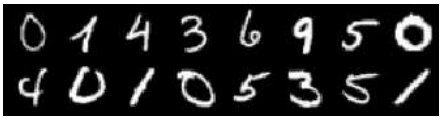
It is widely acknowledged that two indispensable factors contribute to the success of deep learning, namely (1) huge datasets that often contain millions of samples and (2) immense computing power resulting from clusters of graphics processing units (GPUs). Admittedly, these resources are only recently available: the latter allows to train larger neural networks which reduces biases and the former enables variance reduction. However, these two alone are not sufficient to explain the mystery of deep learning due to some of its "dreadful" characteristics: (1) *over-parametrization*: the number of parameters in state-of-the-art deep learning models is often much larger than the sample size (see Table 1), which gives them the potential to overfit the training data, and (2) *nonconvexity*: even with the help of GPUs, training deep learning models is still NP-hard [8] in the worst case due to the highly nonconvex loss function to minimize. In reality, these characteristics are far from nightmares. This sharp difference motivates us to take a closer look at the salient features of deep learning, which we single out a few below.

1.1.1 Depth

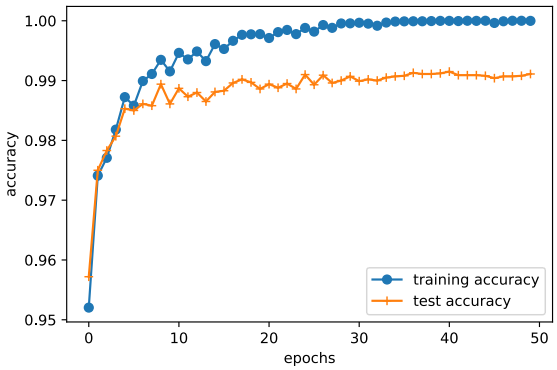
Deep learning expresses complicated nonlinearity through composing many nonlinear functions; see (1). The rationale for this multilayer structure is that, in many real-world datasets such as images, there are different levels of features and lower-level features are building blocks of higher-level ones. See [134] for a visualization of trained features of convolutional neural nets; here in Figure 1, we sample and visualize weights from a pre-trained AlexNet model. This intuition is also supported by empirical results from physiology and neuroscience [56, 2]. The use of function composition marks a sharp difference from traditional statistical methods such as projection pursuit models [38] and multi-index models [73, 27]. It is often observed that depth helps efficiently extract features that are representative of a dataset. In comparison, increasing width (e.g., number of basis functions) in a shallow model leads to less improvement. This suggests that deep learning models excel at representing a very different function space that is suitable for complex datasets.

1.1.2 Algorithmic regularization

The statistical performance of neural networks (e.g., test accuracy) depends heavily on the particular optimization algorithms used for training [131]. This is very different from many classical statistical problems, where the related optimization problems are less complicated. For instance, when the associated optimization



(a) MNIST images



(b) training and test accuracies

Figure 2: (a) shows the images in the public dataset MNIST; and (b) depicts the training and test accuracies along the training dynamics. Note that the training accuracy is approaching 100% and the test accuracy is still high (no overfitting).

problem has a relatively simple structure (e.g., convex objective functions, linear constraints), the solution to the optimization problem can often be unambiguously computed and analyzed. However, in deep neural networks, due to over-parametrization, there are usually many local minima with different statistical performance [72]. Nevertheless, common practice runs stochastic gradient descent with random initialization and finds model parameters with very good prediction accuracy.

1.1.3 Implicit prior learning

It is well observed that deep neural networks trained with only the raw inputs (e.g., pixels of images) can provide a useful representation of the data. This means that after training, the units of deep neural networks can represent features such as edges, corners, wheels, eyes, etc.; see [134]. Importantly, the training process is automatic in the sense that no human knowledge is involved (other than hyper-parameter tuning). This is very different from traditional methods, where algorithms are designed after structural assumptions are posited. It is likely that training an over-parametrized model efficiently learns and incorporates the prior distribution $p(\boldsymbol{x})$ of the input, even though deep learning models are themselves discriminative models. With automatic representation of the prior distribution, deep learning typically performs well on similar datasets (but not very different ones) via transfer learning.

1.2 Towards theory of deep learning

Despite the empirical success, theoretical support for deep learning is still in its infancy. Setting the stage, for any classifier f, denote by $\mathbb{E}(f)$ the expected risk on fresh sample (a.k.a. test error, prediction error or generalization error), and by $\mathbb{E}_n(f)$ the empirical risk / training error averaged over a training dataset. Arguably, the key theoretical question in deep learning is

why is $\mathbb{E}(\hat{f}_n)$ small, where $\hat{f}_n$ is the classifier returned by the training algorithm?

We follow the conventional approximation-estimation decomposition (sometimes, also bias-variance trade-off) to decompose the term $\mathbb{E}(\hat{f}_n)$ into two parts. Let $\mathcal{F}$ be the function space expressible by a family of neural nets. Define $f^* = \operatorname{argmin}_f \mathbb{E}(f)$ to be the best possible classifier and $f^*_{\mathcal{F}} = \operatorname{argmin}_{f\in\mathcal{F}} \mathbb{E}(f)$ to be the best classifier in $\mathcal{F}$. Then, we can decompose the excess error $\mathcal{E} \triangleq \mathbb{E}(\hat{f}_n) - \mathbb{E}(f^*)$ into two parts:

$$\mathcal{E} = \underbrace{\mathbb{E}(f^*_{\mathcal{F}}) - \mathbb{E}(f^*)}_{\text{approximation error}} + \underbrace{\mathbb{E}(\hat{f}_n) - \mathbb{E}(f^*_{\mathcal{F}})}_{\text{estimation error}}. \tag{2}$$

Both errors can be small for deep learning (cf. Figure 2), which we explain below.

- The *approximation error* is determined by the function class $\mathcal{F}$. Intuitively, the larger the class, the smaller the approximation error. Deep learning models use many layers of nonlinear functions (Figure 3)that can drive this error small. Indeed, in Section 5, we provide recent theoretical progress of its representation power. For example, deep models allow efficient representation of interactions among variable while shallow models cannot.

- The *estimation error* reflects the generalization power, which is influenced by both the complexity of the function class $\mathcal{F}$ and the properties of the training algorithms. Interestingly, for *over-parametrized* deep neural nets, stochastic gradient descent typically results in a near-zero training error (i.e., $\mathbb{E}_n(\hat{f}_n) \approx 0$; see e.g. left panel of Figure 2). Moreover, its generalization error $\mathbb{E}(\hat{f}_n)$ remains small or moderate. This "counterintuitive" behavior suggests that for over-parametrized models, gradient-based algorithms enjoy benign statistical properties; we shall see in Section 7 that gradient descent enjoys *implicit regularization* in the over-parametrized regime even without explicit regularization (e.g., ℓ_2 regularization).

The above two points lead to the following heuristic explanation of the success of deep learning models. The large depth of deep neural nets and heavy over-parametrization lead to small or zero training errors, even when running simple algorithms with moderate number of iterations. In addition, these simple algorithms with moderate number of steps do not explore the entire function space and thus have limited complexities, which results in small generalization error with a large sample size. Thus, by combining the two aspects, it explains heuristically that the test error is also small.

1.3 Roadmap of the paper

We first introduce basic deep learning models in Sections 2–4, and then examine their representation power via the lens of approximation theory in Section 5. Section 6 is devoted to training algorithms and their ability of driving the training error small. Then we sample recent theoretical progress towards demystifying the generalization power of deep learning in Section 7. Along the way, we provide our own perspectives, and at the end we identify a few interesting questions for future research in Section 8. The goal of this paper is to present suggestive methods and results, rather than giving conclusive arguments (which is currently unlikely) or a comprehensive survey. We hope that our discussion serves as a stimulus for new statistics research.

2 Feed-forward neural networks

Before introducing the vanilla feed-forward neural nets, let us set up necessary notations for the rest of this section. We focus primarily on classification problems, as regression problems can be addressed similarly. Given the training dataset $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ where $y_i \in [K] \triangleq \{1, 2, \ldots, K\}$ and $\boldsymbol{x}_i \in \mathbb{R}^d$ are independent across $i \in [n]$, supervised learning aims at finding a (possibly random) function $\hat{f}(\boldsymbol{x})$ that predicts the outcome y for a new input $\boldsymbol{x}$, assuming $(y, \boldsymbol{x})$ follows the same distribution as $(y_i, \boldsymbol{x}_i)$. In the terminology of machine learning, the input $\boldsymbol{x}_i$ is often called the *feature*, the output y_i called the *label*, and the pair $(y_i, \boldsymbol{x}_i)$ is an *example*. The function $\hat{f}$ is called the *classifier*, and estimation of $\hat{f}$ is *training* or *learning*. The performance of $\hat{f}$ is evaluated through the prediction error $\mathbb{P}(y \neq \hat{f}(\boldsymbol{x}))$, which can be often estimated from a separate test dataset.

As with classical statistical estimation, for each $k \in [K]$, a classifier approximates the conditional probability $\mathbb{P}(y = k|\boldsymbol{x})$ using a function $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)$ parametrized by $\boldsymbol{\theta}_k$. Then the category with the highest probability is predicted. Thus, learning is essentially estimating the parameters $\boldsymbol{\theta}_k$. In statistics, one of the most popular methods is (multinomial) logistic regression, which stipulates a specific form for the functions $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)$: let $z_k = \boldsymbol{x}^\top \boldsymbol{\beta}_k + \alpha_k$ and $f_k(\boldsymbol{x}; \boldsymbol{\theta}_k) = Z^{-1} \exp(z_k)$ where $Z = \sum_{k=1}^K \exp(z_k)$ is a normalization factor to make $\{f_k(\boldsymbol{x}; \boldsymbol{\theta}_k)\}_{1\le k\le K}$ a valid probability distribution. It is clear that logistic regression induces linear decision boundaries in $\mathbb{R}^d$, and hence it is restrictive in modeling nonlinear dependency between y and $\boldsymbol{x}$. The deep neural networks we introduce below provide a flexible framework for modeling nonlinearity in a fairly general way.

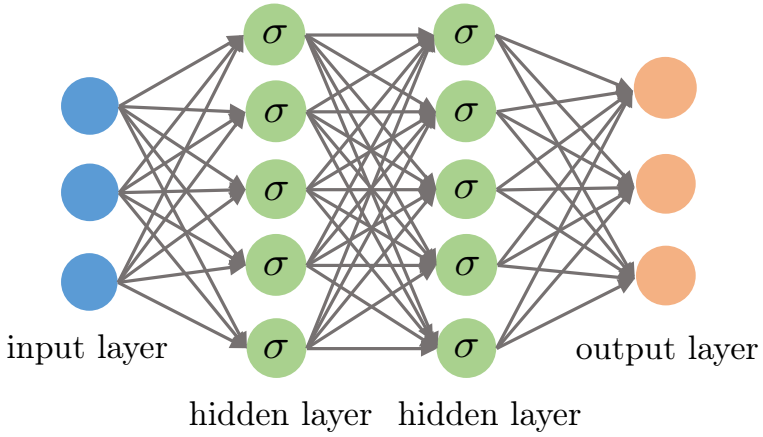


Figure 3: A feed-forward neural network with an input layer, two hidden layers and an output layer. The input layer represents raw features $\{\boldsymbol{x}_i\}_{1\le i\le n}$. Both hidden layers compute an affine transform (a.k.s. indices) of the input and then apply an element-wise activation function $\boldsymbol{\sigma}(\cdot)$. Finally, the output returns a linear transform followed by the softmax activation (resp. simply a linear transform) of the hidden layers for the classification (resp. regression) problem.

2.1 Model setup

From the high level, deep neural networks (DNNs) use composition of a series of simple nonlinear functions to model nonlinearity

$$\boldsymbol{h}^{(L)} = \mathbf{g}^{(L)} \circ \mathbf{g}^{(L-1)} \circ \ldots \circ \mathbf{g}^{(1)}(\boldsymbol{x}),$$

where $\circ$ denotes composition of two functions and L is the number of hidden layers, and is usually called *depth* of a NN model. Letting $\boldsymbol{h}^{(0)} \triangleq \boldsymbol{x}$, one can recursively define $\boldsymbol{h}^{(l)} = \mathbf{g}^{(l)}\big(\boldsymbol{h}^{(l-1)}\big)$ for all $\ell = 1, 2, \ldots, L$. The *feed-forward neural networks*, also called the *multilayer perceptrons* (MLPs), are neural nets with a specific choice of $\mathbf{g}^{(l)}$: for $\ell = 1, \ldots, L$, define

$$\boldsymbol{h}^{(\ell)} = \mathbf{g}^{(l)}\big(\boldsymbol{h}^{(l-1)}\big) \triangleq \boldsymbol{\sigma}\big(\mathbf{W}^{(\ell)}\boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}\big), \tag{3}$$

where $\mathbf{W}^{(l)}$ and $\boldsymbol{b}^{(l)}$ are the weight matrix and the bias / intercept, respectively, associated with the l-th layer, and $\boldsymbol{\sigma}(\cdot)$ is usually a simple given (known) nonlinear function called the *activation function*. In words, in each layer ℓ, the input vector $\boldsymbol{h}^{(\ell-1)}$ goes through an affine transformation first and then passes through a fixed nonlinear function $\boldsymbol{\sigma}(\cdot)$. See Figure 3 for an illustration of a simple MLP with two hidden layers. The activation function $\boldsymbol{\sigma}(\cdot)$ is usually applied element-wise, and a popular choice is the ReLU (Rectified Linear Unit) function:

$$[\boldsymbol{\sigma}(\boldsymbol{z})]_j = \max\{z_j, 0\}. \tag{4}$$

Other choices of activation functions include leaky ReLU, tanh function [79] and the classical sigmoid function $(1 + e^{-z})^{-1}$, which is less used now.

Given an output $\boldsymbol{h}^{(L)}$ from the final hidden layer and a label y, we can define a loss function to minimize. A common loss function for classification problems is the multinomial logistic loss. Using the terminology of deep learning, we say that $\boldsymbol{h}^{(L)}$ goes through an affine transformation and then the *soft-max* function:

$$f_k(\boldsymbol{x}; \boldsymbol{\theta}) \triangleq \frac{\exp(z_k)}{\sum_k \exp(z_k)}, \quad \forall\, k \in [K], \qquad \text{where } \boldsymbol{z} = \mathbf{W}^{(L+1)}\boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)} \in \mathbb{R}^K.$$

Then the loss is defined to be the cross-entropy between the label y (in the form of an indicator vector) and the score vector $(f_1(\boldsymbol{x}; \boldsymbol{\theta}), \ldots, f_K(\boldsymbol{x}; \boldsymbol{\theta}))^\top$, which is exactly the negative log-likelihood of the multinomial logistic regression model:

$$\mathcal{L}(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}), y) = -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log p_k, \tag{5}$$

where $\boldsymbol{\theta} \triangleq \{\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)} : 1 \le \ell \le L+1\}$. As a final remark, the number of parameters scales with both the depth L and the width (i.e., the dimensionality of $\mathbf{W}^{(\ell)}$), and hence it can be quite large for deep neural nets.

2.2 Back-propagation in computational graphs

Training neural networks follows the *empirical risk minimization* paradigm that minimizes the loss (e.g., (5)) over all the training data. This minimization is usually done via *stochastic gradient descent* (SGD). In a way similar to gradient descent, SGD starts from a certain initial value $\boldsymbol{\theta}^0$ and then iteratively updates the parameters $\boldsymbol{\theta}^t$ by moving it in the direction of the negative gradient. The difference is that, in each update, a small subsample $\mathcal{B} \subset [n]$ called a *mini-batch*—which is typically of size 32–512—is randomly drawn and the gradient calculation is only on $\mathcal{B}$ instead of the full batch $[n]$. This saves considerably the computational cost in calculation of gradient. By the law of large numbers, this stochastic gradient should be close to the full sample one, albeit with some random fluctuations. A pass of the whole training set is called an *epoch*. Usually, after several or tens of epochs, the error on a validation set levels off and training is complete. See Section 6 for more details and variants on training algorithms.

The key to the above training procedure, namely SGD, is the calculation of the gradient $\nabla \ell_{\mathcal{B}}(\boldsymbol{\theta})$, where

$$\ell_{\mathcal{B}}(\boldsymbol{\theta}) \triangleq |\mathcal{B}|^{-1} \sum_{i \in \mathcal{B}} \mathcal{L}(\boldsymbol{f}(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i). \tag{6}$$

Gradient computation, however, is in general nontrivial for complex models, and it is susceptible to numerical instability for a model with large depth. Here, we introduce an efficient approach, namely *back-propagation*, for computing gradients in neural networks.

Back-propagation [106] is a direct application of the chain rule in networks. As the name suggests, the calculation is performed in a backward fashion: one first computes $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(L)}$, then $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(L-1)}$, ..., and finally $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(1)}$. For example, in the case of the ReLU activation function[3], we have the following recursive / backward relation

$$\frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell-1)}} = \frac{\partial \boldsymbol{h}^{(\ell)}}{\partial \boldsymbol{h}^{(\ell-1)}} \cdot \frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell)}} = (\mathbf{W}^{(\ell)})^\top \mathsf{diag}\left(\mathbb{1}\{\mathbf{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)} \ge \mathbf{0}\} \right) \frac{\partial \ell_{\mathcal{B}}}{\partial \boldsymbol{h}^{(\ell)}} \tag{7}$$

where $\mathsf{diag}(\cdot)$ denotes a diagonal matrix with elements given by the argument. Note that the calculation of $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(\ell-1)}$ depends on $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(\ell)}$, which is the partial derivatives from the next layer. In this way, the derivatives are "back-propagated" from the last layer to the first layer. These derivatives $\{\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}^{(\ell)}\}$ are then used to update the parameters. For instance, the gradient update for $\mathbf{W}^{(\ell)}$ is given by

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \frac{\partial \ell_{\mathcal{B}}}{\partial \mathbf{W}^{(\ell)}}, \quad \text{where} \quad \frac{\partial \ell_{\mathcal{B}}}{\partial W^{(\ell)}_{jm}} = \frac{\partial \ell_{\mathcal{B}}}{\partial h^{(\ell)}_j} \cdot \sigma' \cdot h^{(\ell-1)}_m, \tag{8}$$

where $\sigma' = 1$ if the j-th element of $\mathbf{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$ is nonnegative, and $\sigma' = 0$ otherwise. The step size $\eta > 0$, also called the *learning rate*, controls how much parameters are changed in a single update.

A more general way to think about neural network models and training is to consider *computational graphs*. Computational graphs are directed acyclic graphs that represent functional relations between variables. They are very convenient and flexible to represent function composition, and moreover, they also allow an efficient way of computing gradients. Consider an MLP with a single hidden layer and an ℓ_2 regularization:

$$\ell^{\lambda}_{\mathcal{B}}(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + r_\lambda(\boldsymbol{\theta}) = \ell_{\mathcal{B}}(\boldsymbol{\theta}) + \lambda \Big(\sum_{j,j'} \big(W^{(1)}_{j,j'}\big)^2 + \sum_{j,j'} \big(W^{(2)}_{j,j'}\big)^2 \Big), \tag{9}$$

where $\ell_{\mathcal{B}}(\boldsymbol{\theta})$ is the same as (6), and $\lambda \ge 0$ is a tuning parameter. A similar example is considered in [45]. The corresponding computational graph is shown in Figure 4. Each node represents a function (inside a circle), which is associated with an output of that function (outside a circle). For example, we view the term $\ell_{\mathcal{B}}(\boldsymbol{\theta})$ as a result of 4 compositions: first the input data $\boldsymbol{x}$ multiplies the weight matrix $\mathbf{W}^{(1)}$ resulting in $\boldsymbol{u}^{(1)}$,

[3] The issue of non-differentiability at the origin is often ignored in implementation.

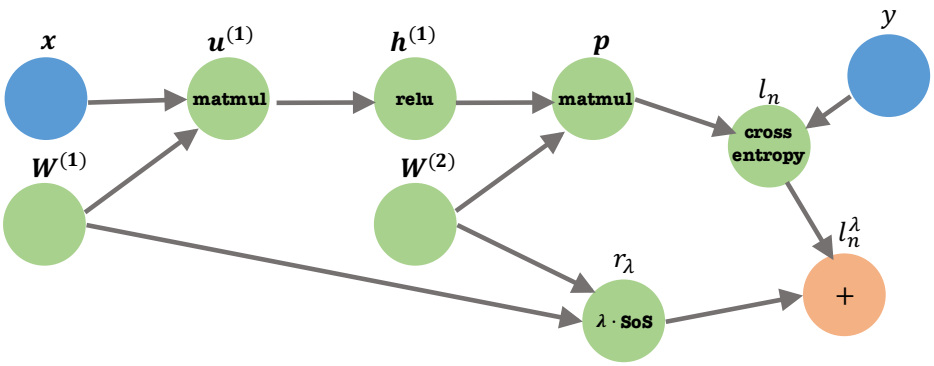


Figure 4: The computational graph illustrates the loss (9). For simplicity, we omit the bias terms. Symbols inside nodes represent functions, and symbols outside nodes represent function outputs (vectors/scalars). `matmul` is matrix multiplication, `relu` is the ReLU activation, `cross entropy` is the cross entropy loss, and `SoS` is the sum of squares.

then it goes through the ReLU activation function `relu` resulting in $\boldsymbol{h}^{(1)}$, then it multiplies another weight matrix $\mathbf{W}^{(2)}$ leading to $\boldsymbol{p}$, and finally it produces the cross-entropy with label y as in (5). The regularization term is incorporated in the graph similarly.

A forward pass is complete when all nodes are evaluated starting from the input $\boldsymbol{x}$. A backward pass then calculates the gradients of $\ell_{\mathcal{B}}^{\lambda}$ with respect to all other nodes in the reverse direction. Due to the chain rule, the gradient calculation for a variable (say, $\partial \ell_{\mathcal{B}} / \partial \boldsymbol{u}^{(1)}$) is simple: it only depends on the gradient value of the variables ($\partial \ell_{\mathcal{B}} / \partial \boldsymbol{h}$) the current node points to, and the function derivative evaluated at the current variable value ($\boldsymbol{\sigma}'(\boldsymbol{u}^{(1)})$). Thus, in each iteration, a computation graph only needs to (1) calculate and store the function evaluations at each node in the forward pass, and then (2) calculate all derivatives in the backward pass.

Back-propagation in computational graphs forms the foundations of popular deep learning programming softwares, including TensorFlow [1] and PyTorch [92], which allows more efficient building and training of complex neural net models.

3 Popular models

Moving beyond vanilla feed-forward neural networks, we introduce two other popular deep learning models, namely, the convolutional neural networks (CNNs) and the recurrent neural networks (RNNs). One important characteristic shared by the two models is *weight sharing*, that is some model parameters are identical across locations in CNNs or across time in RNNs. This is related to the notion of translational invariance in CNNs and stationarity in RNNs. At the end of this section, we introduce a modular thinking for constructing more flexible neural nets.

3.1 Convolutional neural networks

The convolutional neural network (CNN) [71, 40] is a special type of feed-forward neural networks that is tailored for image processing. More generally, it is suitable for analyzing data with salient spatial structures. In this subsection, we focus on image classification using CNNs, where the raw input (image pixels) and features of each hidden layer are represented by a 3D tensor $\boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$. Here, the first two dimensions d_1, d_2 of $\boldsymbol{X}$ indicate spatial coordinates of an image while the third d_3 indicates the number of channels. For instance, d_3 is 3 for the raw inputs due to the red, green and blue channels, and d_3 can be much larger (say, 256) for hidden layers. Each channel is also called a *feature map*, because each feature map is specialized to detect the same feature at different locations of the input, which we will soon explain. We now introduce two building blocks of CNNs, namely the convolutional layer and the pooling layer.

1. *Convolutional layer (CONV).* A convolutional layer has the same functionality as described in (3), where

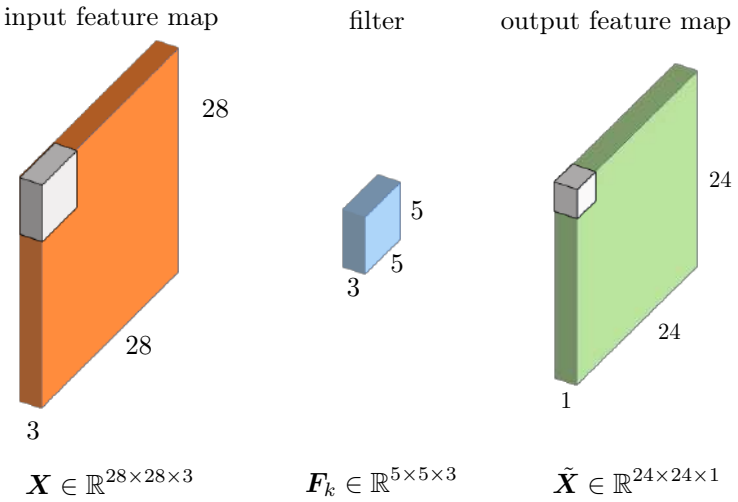


Figure 5: $\boldsymbol{X} \in \mathbb{R}^{28\times28\times3}$ represents the input feature consisting of 28×28 spatial coordinates in a total number of 3 channels / feature maps. $\boldsymbol{F}_k \in \mathbb{R}^{5\times5\times3}$ denotes the k-th filter with size 5×5. The third dimension 3 of the filter automatically matches the number 3 of channels in the previous input. Every 3D patch of $\boldsymbol{X}$ gets convolved with the filter $\boldsymbol{F}_k$ and this as a whole results in a single output feature map $\tilde{X}_{:,:,k}$ with size $24 \times 24 \times 1$. Stacking the outputs of all the filters $\{\boldsymbol{F}_k\}_{1\le k\le K}$ will lead to the output feature with size $24 \times 24 \times K$.

the input feature $\boldsymbol{X} \in \mathbb{R}^{d_1\times d_2\times d_3}$ goes through an affine transformation first and then an element-wise nonlinear activation. The difference lies in the specific form of the affine transformation. A convolutional layer uses a number of *filters* to extract local features from the previous input. More precisely, each filter is represented by a 3D tensor $\boldsymbol{F}_k \in \mathbb{R}^{w\times w\times d_3}$ $(1 \le k \le \tilde{d}_3)$, where w is the size of the filter (typically 3 or 5) and $\tilde{d}_3$ denotes the total number of filters. Note that the third dimension d_3 of $\boldsymbol{F}_k$ is equal to that of the input feature $\boldsymbol{X}$. For this reason, one usually says that the filter has size $w \times w$, while suppressing the third dimension d_3. Each filter $\boldsymbol{F}_k$ then convolves with the input feature $\boldsymbol{X}$ to obtain one single feature map $\boldsymbol{O}^k \in \mathbb{R}^{(d_1-w+1)\times(d_1-w+1)}$, where[4]

$$O^k_{ij} = \left\langle [\boldsymbol{X}]_{ij}, \boldsymbol{F}_k \right\rangle = \sum_{i'=1}^{w}\sum_{j'=1}^{w}\sum_{l=1}^{d_3}[\boldsymbol{X}]_{i+i'-1,j+j'-1,l}[\boldsymbol{F}_k]_{i',j',l}. \tag{10}$$

Here $[\boldsymbol{X}]_{ij} \in \mathbb{R}^{w\times w\times d_3}$ is a small "patch" of $\boldsymbol{X}$ starting at location (i, j). See Figure 5 for an illustration of the convolution operation. If we view the 3D tensors $[\boldsymbol{X}]_{ij}$ and $\boldsymbol{F}_k$ as vectors, then each filter essentially computes their inner product with a part of $\boldsymbol{X}$ indexed by i, j (which can be also viewed as convolution, as its name suggests). One then pack the resulted feature maps $\{\boldsymbol{O}^k\}$ into a 3D tensor $\boldsymbol{O}$ with size $(d_1 - w + 1) \times (d_1 - w + 1) \times \tilde{d}_3$, where

$$[\boldsymbol{O}]_{ijk} = [\boldsymbol{O}^k]_{ij}. \tag{11}$$

The outputs of convolutional layers are then followed by nonlinear activation functions. In the ReLU case, we have

$$\tilde{X}_{ijk} = \sigma(O_{ijk}), \qquad \forall\, i \in [d_1 - w + 1], j \in [d_2 - w + 1], k \in [\tilde{d}_3]. \tag{12}$$

The convolution operation (10) and the ReLU activation (12) work together to extract features $\tilde{\boldsymbol{X}}$ from the input $\boldsymbol{X}$. Different from feed-forward neural nets, the filters $\boldsymbol{F}_k$ are shared across all locations (i, j). A patch $[\boldsymbol{X}]_{ij}$ of an input responds strongly (that is, producing a large value) to a filter $\boldsymbol{F}_k$ if they are positively correlated. Therefore intuitively, each filter $\boldsymbol{F}_k$ serves to extract features similar to $\boldsymbol{F}_k$.

As a side note, after the convolution (10), the spatial size $d_1 \times d_2$ of the input $\boldsymbol{X}$ shrinks to $(d_1 - w + 1) \times (d_2 - w + 1)$ of $\tilde{\boldsymbol{X}}$. However one may want the spatial size unchanged. This can be achieved via *padding*, where one

[4]To simplify notation, we omit the bias/intercept term associated with each filter.

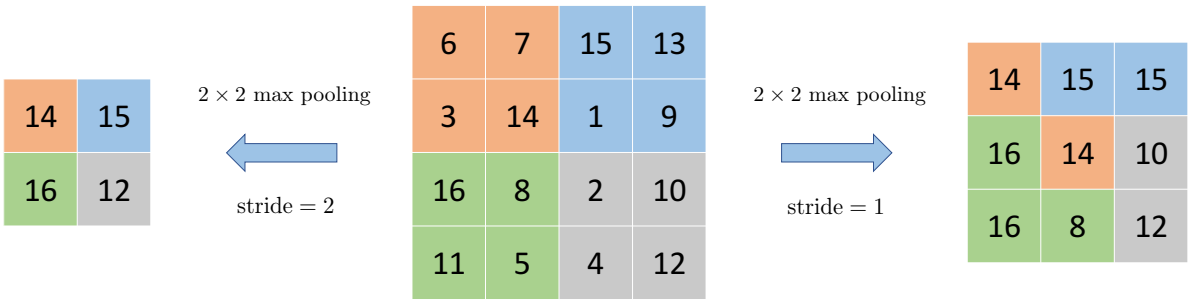


Figure 6: A 2×2 max pooling layer extracts the maximum of 2 by 2 neighboring pixels / features across the spatial dimension.

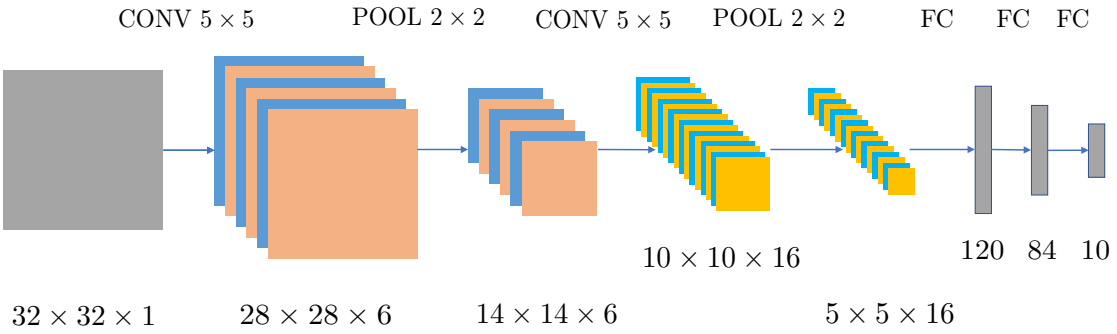


Figure 7: LeNet is composed of an input layer, two convolutional layers, two pooling layers and three fully-connected layers. Both convolutions are valid and use filters with size 5×5. In addition, the two pooling layers use 2×2 average pooling.

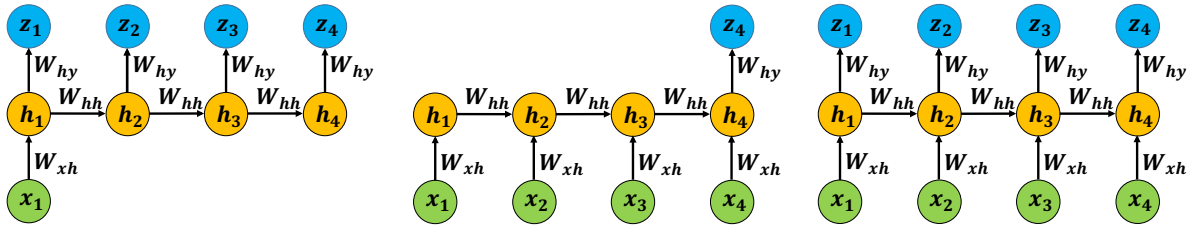
appends zeros to the margins of the input $\boldsymbol{X}$ to enlarge the spatial size to $(d_1 + w - 1) \times (d_2 + w - 1)$. In addition, a *stride* in the convolutional layer determines the gap $i' - i$ and $j' - j$ between two patches $\boldsymbol{X}_{ij}$ and $\boldsymbol{X}_{i'j'}$: in (10) the stride is 1, and a larger stride would lead to feature maps with smaller sizes.

2. *Pooling layer (POOL).* A pooling layer aggregates the information of nearby features into a single one. This downsampling operation reduces the size of the features for subsequent layers and saves computation. One common form of the pooling layer is composed of the 2×2 max-pooling filter. It computes $\max\{X_{i,j,k}, X_{i+1,j,k}, X_{i,j+1,k}, X_{i+1,j+1,k}\}$, that is, the maximum of the 2×2 neighborhood in the spatial coordinates; see Figure 6 for an illustration. Note that the pooling operation is done separately for each feature map k. As a consequence, a 2×2 max-pooling filter acting on $\boldsymbol{X} \in \mathbb{R}^{d_1 \times d_2 \times d_3}$ will result in an output of size $d_1/2 \times d_2/2 \times d_3$. In addition, the pooling layer does not involve any parameters to optimize. Pooling layers serve to reduce redundancy since a small neighborhood around a location (i, j) in a feature map is likely to contain the same information.

In addition, we also use fully-connected layers as building blocks, which we have already seen in Section 2. Each fully-connected layer treats input tensor $\boldsymbol{X}$ as a vector $\mathrm{Vec}(\boldsymbol{X})$, and computes $\tilde{\boldsymbol{X}} = \boldsymbol{\sigma}(\mathbf{W}\mathrm{Vec}(\boldsymbol{X}))$. A fully-connected layer does not use weight sharing and is often used in the last few layers of a CNN. As an example, Figure 7 depicts the well-known LeNet 5 [71], which is composed of two sets of CONV-POOL layers and three fully-connected layers.

3.2 Recurrent neural networks

Recurrent neural nets (RNNs) are another family of powerful models, which are designed to process time series data and other sequence data. RNNs have successful applications in speech recognition [108], machine translation [132], genome sequencing [21], etc. The structure of an RNN naturally forms a computational graph, and can be easily combined with other structures such as CNNs to build large computational graph



(a) One-to-many (b) Many-to-one (c) Many-to-many

Figure 8: Vanilla RNNs with different inputs/outputs settings. (a) has one input but multiple outputs; (b) has multiple inputs but one output; (c) has multiple inputs and outputs. Note that the parameters are shared across time steps.

models for complex tasks. Here we introduce vanilla RNNs and improved variants such as long short-term memory (LSTM).

3.2.1 Vanilla RNNs

Suppose we have general time series inputs $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_T$. A vanilla RNN models the "hidden state" at time t by a vector $\boldsymbol{h}_t$, which is subject to the recursive formula

$$\boldsymbol{h}_t = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t). \tag{13}$$

Here, $f_{\boldsymbol{\theta}}$ is generally a nonlinear function parametrized by $\boldsymbol{\theta}$. Concretely, a vanilla RNN with one hidden layer has the following form[5]

$$\begin{aligned} \boldsymbol{h}_t &= \mathbf{tanh}\left(\mathbf{W}_{hh}\boldsymbol{h}_{t-1} + \mathbf{W}_{xh}\boldsymbol{x}_t + \boldsymbol{b}_{\boldsymbol{h}}\right), \qquad \text{where } \tanh(a) = \tfrac{e^{2a}-1}{e^{2a}+1}, \\ \boldsymbol{z}_t &= \boldsymbol{\sigma}\left(\mathbf{W}_{hy}\boldsymbol{h}_t + \boldsymbol{b}_{\boldsymbol{z}}\right), \end{aligned}$$

where $\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}$ are trainable weight matrices, $\boldsymbol{b}_{\boldsymbol{h}}, \boldsymbol{b}_{\boldsymbol{z}}$ are trainable bias vectors, and $\boldsymbol{z}_t$ is the output at time t. Like many classical time series models, those parameters are shared across time. Note that in different applications, we may have different input/output settings (cf. Figure 8). Examples include

- **One-to-many:** a single input with multiple outputs; see Figure 8(a). A typical application is image captioning, where the input is an image and outputs are a series of words.
- **Many-to-one:** multiple inputs with a single output; see Figure 8(b). One application is text sentiment classification, where the input is a series of words in a sentence and the output is a label (e.g., positive vs. negative).
- **Many-to-many:** multiple inputs and outputs; see Figure 8(c). This is adopted in machine translation, where inputs are words of a source language (say Chinese) and outputs are words of a target language (say English).

As the case with feed-forward neural nets, we minimize a loss function using back-propagation, where the loss is typically

$$\ell_{\mathcal{T}}(\boldsymbol{\theta}) = \sum_{t\in\mathcal{T}} \mathcal{L}(y_t, \boldsymbol{z}_t) = -\sum_{t\in\mathcal{T}}\sum_{k=1}^{K} \mathbb{1}\{y_t = k\} \log\left(\frac{\exp([\boldsymbol{z}_t]_k)}{\sum_k \exp([\boldsymbol{z}_t]_k)}\right),$$

where K is the number of categories for classification (e.g., size of the vocabulary in machine translation), and $\mathcal{T} \subset [T]$ is the length of the output sequence. During the training, the gradients $\partial\ell_{\mathcal{T}}/\partial\boldsymbol{h}_t$ are computed in the reverse time order (from T to t). For this reason, the training process is often called *back-propagation through time*.

[5]Similar to the activation function $\boldsymbol{\sigma}(\cdot)$, the function $\mathbf{tanh}(\cdot)$ means element-wise operations.

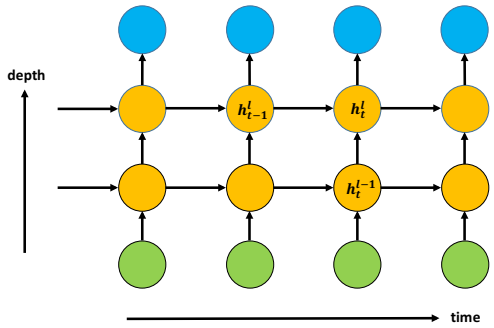


Figure 9: A vanilla RNN with two hidden layers. Higher-level hidden states $\boldsymbol{h}_t^\ell$ are determined by the old states $\boldsymbol{h}_{t-1}^\ell$ and lower-level hidden states $\boldsymbol{h}_t^{\ell-1}$. Multilayer RNNs generalize both feed-forward neural nets and one-hidden-layer RNNs.

One notable drawback of vanilla RNNs is that, they have difficulty in capturing long-range dependencies in sequence data when the length of the sequence is large. This is sometimes due to the phenomenon of *exploding / vanishing gradients*. Take Figure 8(c) as an example. Computing $\partial \ell_{\mathcal{T}} / \partial \boldsymbol{h}_1$ involves the product $\prod_{t=1}^{3} (\partial \boldsymbol{h}_{t+1} / \partial \boldsymbol{h}_t)$ by the chain rule. However, if the sequence is long, the product will be the multiplication of many Jacobian matrices, which usually results in exponentially large or small singular values. To alleviate this issue, in practice, the forward pass and backward pass are implemented in a shorter sliding window $\{t_1, t_1 + 1, \ldots, t_2\}$, instead of the full sequence $\{1, 2, \ldots, T\}$. Though effective in some cases, this technique alone does not fully address the issue of long-term dependency.

3.2.2 GRUs and LSTM

There are two improved variants that alleviate the above issue: gated recurrent units (GRUs) [26] and long short-term memory (LSTM) [54].

- A **GRU** refines the recursive formula (13) by introducing *gates*, which are vectors of the same length as $\boldsymbol{h}_t$. The gates, which take values in $[0, 1]$ elementwise, multiply with $\boldsymbol{h}_{t-1}$ elementwise and determine how much they keep the old hidden states.
- An **LSTM** similarly uses gates in the recursive formula. In addition to $\boldsymbol{h}_t$, an LSTM maintains a *cell state*, which takes values in $\mathbb{R}$ elementwise and are analogous to counters.

Here we only discuss LSTM in detail. Denote by $\odot$ the element-wise multiplication. We have a recursive formula in replace of (13):

$$\begin{pmatrix} \boldsymbol{i}_t \\ \boldsymbol{f}_t \\ \boldsymbol{o}_t \\ \boldsymbol{g}_t \end{pmatrix} = \begin{pmatrix} \boldsymbol{\sigma} \\ \boldsymbol{\sigma} \\ \boldsymbol{\sigma} \\ \mathbf{tanh} \end{pmatrix} \mathbf{W} \begin{pmatrix} \boldsymbol{h}_{t-1} \\ \boldsymbol{x}_t \\ 1 \end{pmatrix},$$
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \boldsymbol{g}_t,$$
$$\boldsymbol{h}_t = \boldsymbol{o}_t \odot \mathbf{tanh}(\boldsymbol{c}_t),$$

where $\mathbf{W}$ is a big weight matrix with appropriate dimensions. The cell state vector $\boldsymbol{c}_t$ carries information of the sequence (e.g., singular/plural form in a sentence). The forget gate $\boldsymbol{f}_t$ determines how much the values of $\boldsymbol{c}_{t-1}$ are kept for time t, the input gate $\boldsymbol{i}_t$ controls the amount of update to the cell state, and the output gate $\boldsymbol{o}_t$ gives how much $\boldsymbol{c}_t$ reveals to $\boldsymbol{h}_t$. Ideally, the elements of these gates have nearly binary values. For example, an element of $\boldsymbol{f}_t$ being close to 1 may suggest the presence of a feature in the sequence data. Similar to the skip connections in residual nets, the cell state $\boldsymbol{c}_t$ has an additive recursive formula, which helps back-propagation and thus captures long-range dependencies.

3.2.3 Multilayer RNNs

Multilayer RNNs are generalization of the one-hidden-layer RNN discussed above. Figure 9 shows a vanilla RNN with two hidden layers. In place of (13), the recursive formula for an RNN with L hidden layers now reads

$$\boldsymbol{h}_t^\ell = \mathbf{tanh}\left[\mathbf{W}^\ell \begin{pmatrix} \boldsymbol{h}_t^{\ell-1} \\ \boldsymbol{h}_{t-1}^{\ell} \\ 1 \end{pmatrix}\right], \quad \text{for all } \ell \in [L], \qquad \boldsymbol{h}_t^0 \triangleq \boldsymbol{x}_t.$$

Note that a multilayer RNN has two dimensions: the sequence length T and depth L. Two special cases are the feed-forward neural nets (where $T = 1$) introduced in Section 2, and RNNs with one hidden layer (where $L = 1$). Multilayer RNNs usually do not have very large depth (e.g., 2–5), since T is already very large.

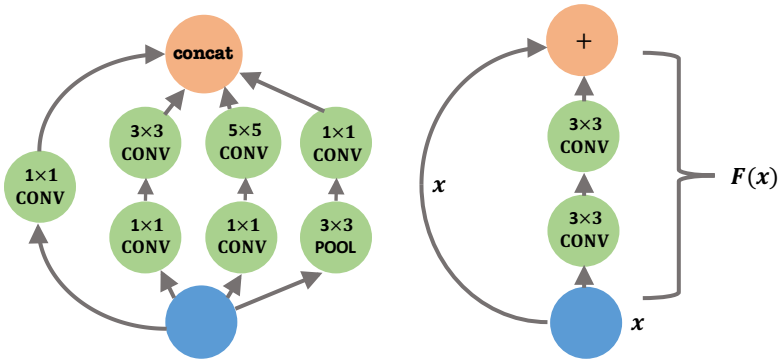
Finally, we remark that CNNs, RNNs, and other neural nets can be easily combined to tackle tasks that involve different sources of input data. For example, in image captioning, the images are first processed through a CNN, and then the high-level features are fed into an RNN as inputs. Theses neural nets combined together form a large computational graph, so they can be trained using back-propagation. This generic training method provides much flexibility in various applications.

3.3 Modules

Deep neural nets are essentially composition of many nonlinear functions. A component function may be designed to have specific properties in a given task, and it can be itself resulted from composing a few simpler functions. In LSTM, we have seen that the building block consists of several intermediate variables, including cell states and forget gates that can capture long-term dependency and alleviate numerical issues.

This leads to the idea of designing *modules* for building more complex neural net models. Desirable modules usually have low computational costs, alleviate numerical issues in training, and lead to good statistical accuracy. Since modules and the resulting neural net models form computational graphs, training follows the same principle briefly described in Section 2.

Here, we use the examples of *Inception* and *skip connections* to illustrate the ideas behind modules. Figure 10(a) is an example of "Inception" modules used in GoogleNet [123]. As before, all the convolutional layers are followed by the ReLU activation function. The concatenation of information from filters with different sizes give the model great flexibility to capture spatial information. Note that 1×1 filters is an $1 \times 1 \times d_3$ tensor (where d_3 is the number of feature maps), so its convolutional operation does not interact with other spatial coordinates, only serving to aggregate information from different feature maps at the same coordinate. This reduces the number of parameters and speeds up the computation. Similar ideas appear in other work [78, 57].



(a) "Inception" module (b) Skip connections

Figure 10: (a) The "Inception" module from GoogleNet. `Concat` means combining all features maps into a tensor. (b) Skip connections are added every two layers in ResNets.

Another module, usually called *skip connections*, is widely used to alleviate numerical issues in very deep neural nets, with additional benefits in optimization efficiency and statistical accuracy. Training very deep

neural nets are generally more difficult, but the introduction of skip connections in *residual networks* [50, 51] has greatly eased the task.

The high level idea of skip connections is to add an identity map to an existing nonlinear function. Let $\mathbf{F}(\boldsymbol{x})$ be an arbitrary nonlinear function represented by a (fragment of) neural net, then the idea of skip connections is simply replacing $\mathbf{F}(\boldsymbol{x})$ with $\boldsymbol{x}+\mathbf{F}(\boldsymbol{x})$. Figure 10(b) shows a well-known structure from residual networks [50]—for every two layers, an identity map is added:

$$\boldsymbol{x} \longmapsto \boldsymbol{\sigma}(\boldsymbol{x} + \mathbf{F}(\boldsymbol{x})) = \boldsymbol{\sigma}(\boldsymbol{x} + \mathbf{W}'\boldsymbol{\sigma}(\mathbf{W}\boldsymbol{x} + \boldsymbol{b}) + \boldsymbol{b}'), \tag{14}$$

where $\boldsymbol{x}$ can be hidden nodes from any layer and $\mathbf{W}, \mathbf{W}', \boldsymbol{b}, \boldsymbol{b}'$ are corresponding parameters. By repeating (namely composing) this structure throughout all layers, [50, 51] are able to train neural nets with hundreds of layers easily, which overcomes well-observed training difficulties in deep neural nets. Moreover, deep residual networks also improve statistical accuracy, as the classification error on ImageNet challenge was reduced by 46% from 2014 to 2015. As a side note, skip connections can be used flexibly. They are not restricted to the form in (14), and can be used between any pair of layers ℓ, ℓ' [55].

4 Deep unsupervised learning

In supervised learning, given labelled training set $\{(y_i, \mathbf{x}_i)\}$, we focus on discriminative models, which essentially represents $\mathbb{P}(y \,|\, \boldsymbol{x})$ by a deep neural net $f(\boldsymbol{x}; \boldsymbol{\theta})$ with parameters $\boldsymbol{\theta}$. Unsupervised learning, in contrast, aims at extracting *information* from *unlabeled* data $\{\boldsymbol{x}_i\}$, where the labels $\{y_i\}$ are absent. In regard to this information, it can be a low-dimensional embedding of the data $\{\boldsymbol{x}_i\}$ or a generative model with latent variables to approximate the distribution $\mathbb{P}_{\mathbf{X}}(\boldsymbol{x})$. To achieve these goals, we introduce two popular unsupervised deep leaning models, namely, autoencoders and generative adversarial networks (GANs). The first one can be viewed as a dimension reduction technique, and the second as a density estimation method. DNNs are the key elements for both of these two models.

4.1 Autoencoders

Recall that in dimension reduction, the goal is to reduce the dimensionality of the data and at the same time preserve its salient features. In particular, in principal component analysis (PCA), the goal is to embed the data $\{\boldsymbol{x}_i\}_{1\le i\le n}$ into a low-dimensional space via a linear function $\boldsymbol{f}$ such that maximum variance can be explained. Equivalently, we want to find linear functions $\boldsymbol{f} : \mathbb{R}^d \to \mathbb{R}^k$ and $\boldsymbol{g} : \mathbb{R}^k \to \mathbb{R}^d$ $(k \le d)$ such that the difference between $\boldsymbol{x}_i$ and $\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}_i))$ is minimized. Formally, we let

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{W}_f\boldsymbol{x} \triangleq \boldsymbol{h} \quad \text{and} \quad \boldsymbol{g}(\boldsymbol{h}) = \boldsymbol{W}_g\boldsymbol{h}, \quad \text{where} \quad \boldsymbol{W}_f \in \mathbb{R}^{k\times d} \text{ and } \boldsymbol{W}_g \in \mathbb{R}^{d\times k}.$$

Here, for simplicity, we assume that the intercept/bias terms for $\boldsymbol{f}$ and $\boldsymbol{g}$ are zero. Then, PCA amounts to minimizing the quadratic loss function

$$\text{minimize}_{\boldsymbol{W}_f, \boldsymbol{W}_g} \qquad \frac{1}{n}\sum_{i=1}^{n} \|\boldsymbol{x}_i - \boldsymbol{W}_f\boldsymbol{W}_g\boldsymbol{x}_i\|_2^2 . \tag{15}$$

It is the same as minimizing $\|\mathbf{X} - \mathbf{W}\mathbf{X}\|_{\mathrm{F}}^2$ subject to $\mathrm{rank}(\mathbf{W}) \le k$, where $\mathbf{X} \in \mathbb{R}^{p\times n}$ is the design matrix. The solution is given by the singular value decomposition of $\mathbf{X}$ [44, Thm. 2.4.8], which is exactly what PCA does. It turns out that PCA is a special case of autoencoders, which is often known as the *undercomplete linear autoencoder*.

More broadly, autoencoders are neural network models for (nonlinear) dimension reduction, which generalize PCA. An autoencoder has two key components, namely, the encoder function $\boldsymbol{f}(\cdot)$, which maps the input $\boldsymbol{x} \in \mathbb{R}^d$ to a hidden code/representation $\boldsymbol{h} \triangleq \boldsymbol{f}(\boldsymbol{x}) \in \mathbb{R}^k$, and the decoder function $\boldsymbol{g}(\cdot)$, which maps the hidden representation $\boldsymbol{h}$ to a point $\boldsymbol{g}(\boldsymbol{h}) \in \mathbb{R}^d$. Both functions can be multilayer neural networks as (3). See Figure 11 for an illustration of autoencoders. Let $\mathcal{L}(\boldsymbol{x}_1, \boldsymbol{x}_2)$ be a loss function that measures the difference between $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ in $\mathbb{R}^d$. Similar to PCA, an autoencoder is used to find the encoder $\boldsymbol{f}$ and

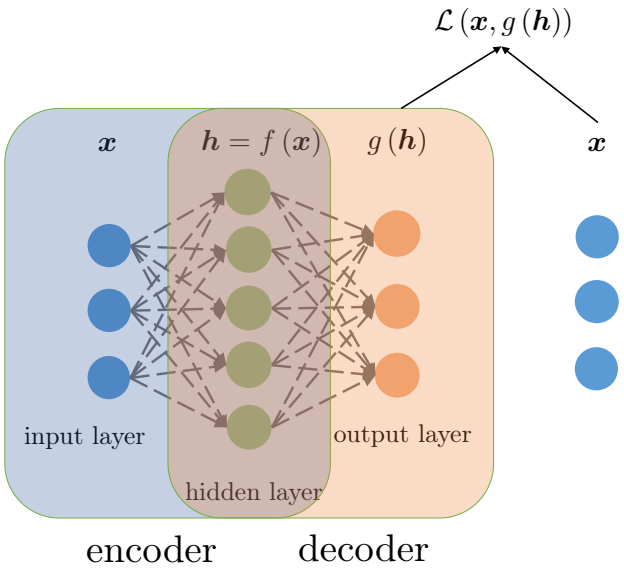


Figure 11: First an input $\boldsymbol{x}$ goes through the decoder $\boldsymbol{f}(\cdot)$, and we obtain its hidden representation $\boldsymbol{h} = \boldsymbol{f}(\boldsymbol{x})$. Then, we use the decoder $\boldsymbol{g}(\cdot)$ to get $\boldsymbol{g}(\boldsymbol{h})$ as a reconstruction of $\boldsymbol{x}$. Finally, the loss is determined from the difference between the original input $\boldsymbol{x}$ and its reconstruction $\boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x}))$.

decoder $\boldsymbol{g}$ such that $\mathcal{L}(\boldsymbol{x}, \boldsymbol{g}(\boldsymbol{f}(\boldsymbol{x})))$ is as small as possible. Mathematically, this amounts to solving the following minimization problem

$$\text{minimize}_{\boldsymbol{f},\boldsymbol{g}} \quad \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(\boldsymbol{x}_i, \boldsymbol{g}\left(\boldsymbol{h}_i\right)\right) \quad \text{with} \quad \boldsymbol{h}_i = \boldsymbol{f}\left(\boldsymbol{x}_i\right), \quad \text{for all } i \in [n]. \tag{16}$$

One needs to make structural assumptions on the functions $\boldsymbol{f}$ and $\mathbf{g}$ in order to find useful representations of the data, which leads to different types of autoencoders. Indeed, if no assumption is made, choosing $\boldsymbol{f}$ and $\boldsymbol{g}$ to be identity functions clearly minimizes the above optimization problem. To avoid this trivial solution, one natural way is to require that the encoder f maps the data onto a space with a smaller dimension, i.e., $k < d$. This is the *undercomplete autoencoder* that includes PCA as a special case. There are other structured autoencoders which add desired properties to the model such as sparsity or robustness, mainly through regularization terms. Below we present two other common types of autoencoders.

- *Sparse autoencoders.* One may believe that the dimension k of the hidden code $\boldsymbol{h}_i$ is larger than the input dimension d, and that $\boldsymbol{h}_i$ admits a sparse representation. As with LASSO [126] or SCAD [36], one may add a regularization term to the reconstruction loss $\mathcal{L}$ in (16) to encourage sparsity [98]. A sparse autoencoder solves

$$\min_{\boldsymbol{f},\boldsymbol{g}} \underbrace{\frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\left(\boldsymbol{x}_i, \boldsymbol{g}\left(\boldsymbol{h}_i\right)\right)}_{\text{loss}} + \underbrace{\lambda \left\|\boldsymbol{h}_i\right\|_1}_{\text{regularizer}} \quad \text{with} \quad \boldsymbol{h}_i = \boldsymbol{f}\left(\boldsymbol{x}_i\right), \text{ for all } i \in [n].$$

  This is similar to *dictionary learning*, where one aims at finding a sparse representation of input data on an overcomplete basis. Due to the imposed sparsity, the model can potentially learn useful features of the data.

- *Denoising autoencoders.* One may hope that the model is robust to noise in the data: even if the input data $\boldsymbol{x}_i$ are corrupted by small noise $\boldsymbol{\xi}_i$ or miss some components (the noise level or the missing probability is typically small), an ideal autoencoder should faithfully recover the original data. A denoising autoencoder [128] achieves this robustness by explicitly building a noisy data $\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i + \boldsymbol{\xi}_i$ as the new input,

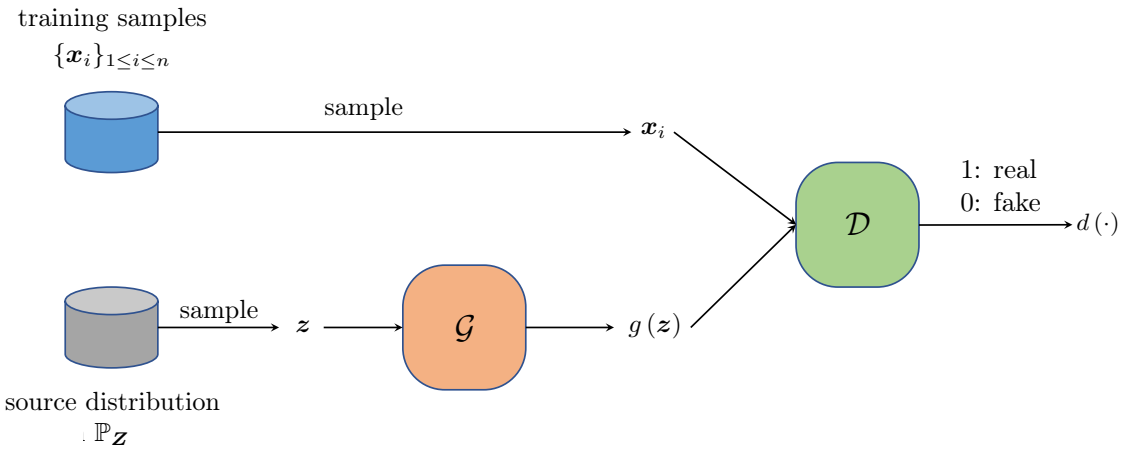


Figure 12: GANs consist of two components, a generator $\mathcal{G}$ which generates fake samples and a discriminator $\mathcal{D}$ which differentiate the true ones from the fake ones.

and then solves an optimization problem similar to (16) where $\mathcal{L}(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{h}_i))$ is replaced by $\mathcal{L}(\boldsymbol{x}_i, \boldsymbol{g}(\boldsymbol{f}(\tilde{\boldsymbol{x}}_i)))$. A denoising autoencoder encourages the encoder/decoder to be stable in the neighborhood of an input, which is generally a good statistical property. An alternative way could be constraining f and g in the optimization problem, but that would be very difficult to optimize. Instead, sampling by adding small perturbations in the input provides a simple implementation. We shall see similar ideas in Section 6.3.3.

4.2 Generative adversarial networks

Given unlabeled data $\{\boldsymbol{x}_i\}_{1\le i\le n}$, density estimation aims to estimate the underlying probability density function $\mathbb{P}_{\boldsymbol{X}}$ from which the data is generated. Both parametric and nonparametric estimators [115] have been proposed and studied under various assumptions on the underlying distribution. Different from these classical density estimators, where the density function is explicitly defined in relatively low dimension, generative adversarial networks (GANs) [46] can be categorized as an *implicit* density estimator in much higher dimension. The reasons are twofold: (1) GANs put more emphasis on sampling from the distribution $\mathbb{P}_{\boldsymbol{X}}$ than estimation; (2) GANs define the density estimation implicitly through a source distribution $\mathbb{P}_{\boldsymbol{Z}}$ and a generator function $g(\cdot)$, which is usually a deep neural network. We introduce GANs from the perspective of sampling from $\mathbb{P}_{\boldsymbol{X}}$ and later we will generalize the vanilla GANs using its relation to density estimators.

4.2.1 Sampling view of GANs

Suppose the data $\{\boldsymbol{x}_i\}_{1\le i\le n}$ at hand are all real images, and we want to generate *new* natural images. With this goal in mind, GAN models a *zero-sum* game between two players, namely, the generator $\mathcal{G}$ and the discriminator $\mathcal{D}$. The generator $\mathcal{G}$ tries to generate fake images akin to the true images $\{\boldsymbol{x}_i\}_{1\le i\le n}$ while the discriminator $\mathcal{D}$ aims at differentiating the fake ones from the true ones. Intuitively, one hopes to learn a generator $\mathcal{G}$ to generate images where the *best* discriminator $\mathcal{D}$ cannot distinguish. Therefore the payoff is higher for the generator $\mathcal{G}$ if the probability of the discriminator $\mathcal{D}$ getting wrong is higher, and correspondingly the payoff for the discriminator correlates positively with its ability to tell wrong from truth.

Mathematically, the generator $\mathcal{G}$ consists of two components, an source distribution $\mathbb{P}_{\boldsymbol{Z}}$ (usually a standard multivariate Gaussian distribution with hundreds of dimensions) and a function $\mathbf{g}(\cdot)$ which maps a sample $\boldsymbol{z}$ from $\mathbb{P}_{\boldsymbol{Z}}$ to a point $\mathbf{g}(\boldsymbol{z})$ living in the same space as $\boldsymbol{x}$. For generating images, $\mathbf{g}(\boldsymbol{z})$ would be a 3D tensor. Here $\mathbf{g}(\boldsymbol{z})$ is the fake sample generated from $\mathcal{G}$. Similarly the discriminator $\mathcal{D}$ is composed of one function which takes an image $\boldsymbol{x}$ (real or fake) and return a number $d(\boldsymbol{x}) \in [0, 1]$, the probability of $\boldsymbol{x}$ being a real sample from $\mathbb{P}_{\boldsymbol{X}}$ or not. Oftentimes, both the generating function $\mathbf{g}(\cdot)$ and the discriminating function $d(\cdot)$ are realized by deep neural networks, e.g., CNNs introduced in Section 3.1. See Figure 12 for an illustration for GANs. Denote $\boldsymbol{\theta}_{\mathcal{G}}$ and $\boldsymbol{\theta}_{\mathcal{D}}$ the parameters in $\mathbf{g}(\cdot)$ and $d(\cdot)$, respectively. Then GAN tries to solve the following *min-max* problem:

$$\min_{\boldsymbol{\theta}_{\mathcal{G}}} \max_{\boldsymbol{\theta}_{\mathcal{D}}} \quad \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\boldsymbol{X}}}\left[\log\left(d\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{\boldsymbol{z}\sim\mathbb{P}_{\boldsymbol{Z}}}\left[\log\left(1-d\left(\mathbf{g}\left(\boldsymbol{z}\right)\right)\right)\right]. \tag{17}$$

Recall that $d(\boldsymbol{x})$ models the belief / probability that the discriminator thinks that $\boldsymbol{x}$ is a true sample. Fix the parameters $\boldsymbol{\theta}_{\mathcal{G}}$ and hence the generator $\mathcal{G}$ and consider the inner maximization problem. We can see that the goal of the discriminator is to maximize its ability of differentiation. Similarly, if we fix $\boldsymbol{\theta}_{\mathcal{D}}$ (and hence the discriminator), the generator tries to generate more realistic images $\mathbf{g}(\boldsymbol{z})$ to fool the discriminator.

4.2.2 Density estimation view of GANs

Let us now take a density-estimation view of GANs. Fixing the source distribution $\mathbb{P}_{\boldsymbol{Z}}$, any generator $\mathcal{G}$ induces a distribution $\mathbb{P}_{\mathcal{G}}$ over the space of images. Removing the restrictions on $d(\cdot)$, one can then rewrite (17) as

$$\min_{\mathbb{P}_{\mathcal{G}}} \max_{d(\cdot)} \quad \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\boldsymbol{X}}}\left[\log\left(d\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\mathcal{G}}}\left[\log\left(1-d\left(\boldsymbol{x}\right)\right)\right]. \tag{18}$$

Observe that the inner maximization problem is solved by the likelihood ratio, i.e.

$$d^{*}\left(\boldsymbol{x}\right) = \frac{\mathbb{P}_{\boldsymbol{X}}\left(\boldsymbol{x}\right)}{\mathbb{P}_{\boldsymbol{X}}\left(\boldsymbol{x}\right)+\mathbb{P}_{\mathcal{G}}\left(\boldsymbol{x}\right)}.$$

As a result, (18) can be simplified as

$$\min_{\mathbb{P}_{\mathcal{G}}} \quad \mathrm{JS}\left(\mathbb{P}_{\boldsymbol{X}} \parallel \mathbb{P}_{\mathcal{G}}\right), \tag{19}$$

where $\mathrm{JS}(\cdot\|\cdot)$ denotes the Jensen–Shannon divergence between two distributions

$$\mathrm{JS}\left(\mathbb{P}_{\boldsymbol{X}}\|\mathbb{P}_{\mathcal{G}}\right) = \frac{1}{2}\mathrm{KL}\big(\mathbb{P}_{\boldsymbol{X}} \parallel \tfrac{\mathbb{P}_{\boldsymbol{X}}+\mathbb{P}_{\mathcal{G}}}{2}\big) + \frac{1}{2}\mathrm{KL}\big(\mathbb{P}_{\mathcal{G}} \parallel \tfrac{\mathbb{P}_{\boldsymbol{X}}+\mathbb{P}_{\mathcal{G}}}{2}\big).$$

In words, the vanilla GAN (17) seeks a density $\mathbb{P}_{\mathcal{G}}$ that is closest to $\mathbb{P}_{\boldsymbol{X}}$ in terms of the Jensen–Shannon divergence. This view allows to generalize GANs to other variants, by changing the distance metric. Examples include f-GAN [90], Wasserstein GAN (W-GAN) [6], MMD GAN [75], etc. We single out the Wasserstein GAN (W-GAN) [6] to introduce due to its popularity. As the name suggests, it minimizes the Wasserstein distance between $\mathbb{P}_{\boldsymbol{X}}$ and $\mathbb{P}_{\mathcal{G}}$:

$$\min_{\boldsymbol{\theta}_{\mathcal{G}}} \quad \mathrm{WS}\left(\mathbb{P}_{\boldsymbol{X}}\|\mathbb{P}_{\mathcal{G}}\right) \;=\; \min_{\boldsymbol{\theta}_{\mathcal{G}}} \sup_{f:f \text{ 1-Lipschitz}} \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\boldsymbol{X}}}\left[f\left(\boldsymbol{x}\right)\right] - \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_{\mathcal{G}}}\left[f\left(\boldsymbol{x}\right)\right], \tag{20}$$

where $f(\cdot)$ is taken over all Lipschitz functions with coefficient 1. Comparing W-GAN (20) with the original formulation of GAN (17), one finds that the Lipschitz function f in (20) corresponds to the discriminator $\mathcal{D}$ in (17) in the sense that they share similar objectives to differentiate the true distribution $\mathbb{P}_{\boldsymbol{X}}$ from the fake one $\mathbb{P}_{\mathcal{G}}$. In the end, we would like to mention that GANs are more difficult to train than supervised deep learning models such as CNNs [110]. Apart from the training difficulty, how to evaluate GANs objectively and effectively is an ongoing research.

5 Representation power: approximation theory

Having seen the building blocks of deep learning models in the previous sections, it is natural to ask: what is the benefits of composing multiple layers of nonlinear functions. In this section, we address this question from a approximation theoretical point of view. Mathematically, letting $\mathcal{H}$ be the space of functions representable by neural nets (NNs), how well can a function f (with certain properties) be approximated by functions in $\mathcal{H}$. We first revisit universal approximation theories, which are mostly developed for shallow neural nets (neural nets with a single hidden layer), and then provide recent results that demonstrate the benefits of depth in neural nets. Other notable works include Kolmogorov-Arnold superposition theorem [7, 120], and circuit complexity for neural nets [91].

5.1 Universal approximation theory for shallow NNs

The universal approximation theories study the approximation of f in a space $\mathcal{F}$ by a function represented by a one-hidden-layer neural net

$$g(\boldsymbol{x}) = \sum_{j=1}^{N} c_j \sigma_*(\boldsymbol{w}_j^\top \boldsymbol{x} - b_j), \tag{21}$$

where $\sigma_* : \mathbb{R} \to \mathbb{R}$ is certain activation function and N is the number of hidden units in the neural net. For different space $\mathcal{F}$ and activation function σ_*, there are upper bounds and lower bounds on the approximation error $\|f - g\|$. See [93] for a comprehensive overview. Here we present representative results.

First, as $N \to \infty$, any continuous function f can be approximated by some g under mild conditions. Loosely speaking, this is because each component $\sigma_*(\boldsymbol{w}_j^\top \boldsymbol{x} - b_j)$ behaves like a basis function and functions in a suitable space $\mathcal{F}$ admits a basis expansion. Given the above heuristics, the next natural question is: what is the rate of approximation for a finite N?

Let us restrict the domain of $\boldsymbol{x}$ to a unit ball B^d in $\mathbb{R}^d$. For $p \in [1, \infty)$ and integer $m \geq 1$, consider the L^p space and the Sobolev space with standard norms

$$\|f\|_p = \Big[\int_{B^n} |g(\boldsymbol{x})|^p \, d\boldsymbol{x} \Big]^{1/p}, \qquad \|f\|_{m,p} = \Big[\sum_{0 \leq |\boldsymbol{k}| \leq m} \|D^{\boldsymbol{k}} f\|_p^p \Big]^{1/p},$$

where $D^{\boldsymbol{k}} f$ denotes partial derivatives indexed by $\boldsymbol{k} \in \mathbb{Z}_+^d$. Let $\mathcal{F} \triangleq \mathcal{F}_p^m$ be the space of functions f in the Sobolev space with $\|f\|_{m,p} \leq 1$. Note that functions in $\mathcal{F}$ have bounded derivatives up to m-th order, and that smoothness of functions is controlled by m (larger m means smoother). Denote by $\mathcal{H}_N$ the space of functions with the form (21). The following general upper bound is due to [85].

Theorem 1 (Theorem 2.1 in [85]). *Assume $\sigma_* : \mathbb{R} \to \mathbb{R}$ is such that σ_* has arbitrary order derivatives in an open interval I, and that σ_* is not a polynomial on I. Then, for any $p \in [1, \infty)$, $d \geq 2$, and integer $m \geq 1$,*

$$\sup_{f \in \mathcal{F}_p^m} \inf_{g \in \mathcal{H}_N} \|f - g\|_p \leq C_{d,m,p} \, N^{-m/d},$$

where $C_{d,m,p}$ is independent of N, the number of hidden units.

In the above theorem, the condition on $\sigma_*(\cdot)$ is mainly technical. This upper bound is useful when the dimension d is not large. It clearly implies that the one-hidden-layer neural net is able to approximate any smooth function with enough hidden units. However, it is unclear how to find a good approximator g; nor do we have control over the magnitude of the parameters (huge weights are impractical). While increasing the number of hidden units N leads to better approximation, the exponent $-m/d$ suggests the presence of the *curse of dimensionality*. The following (nearly) matching lower bound is stated in [80].

Theorem 2 (Theorem 5 in [80]). *Let $p \geq 1$, $m \geq 1$ and $N \geq 2$. If the activation function is the standard sigmoid function $\sigma(t) = (1 + e^{-t})^{-1}$, then*

$$\sup_{f \in \mathcal{F}_p^m} \inf_{g \in \mathcal{H}_N} \|f - g\|_p \geq C'_{d,m,p} \, (N \log N)^{-m/d}, \tag{22}$$

where $C'_{d,m,p}$ is independent of N.

Results for other activation functions are also obtained by [80]. Moreover, the term $\log N$ can be removed if we assume an additional continuity condition [85].

For the natural space $\mathcal{F}_p^m$ of smooth functions, the exponential dependence on d in the upper and lower bounds may look unappealing. However, [12] showed that for a different function space, there is a good dimension-free approximation by the neural nets. Suppose that a function $f : \mathbb{R}^d \mapsto \mathbb{R}$ has a Fourier representation

$$f(\boldsymbol{x}) = \int_{\mathbb{R}^d} e^{i\langle \boldsymbol{\omega}, \boldsymbol{x} \rangle} \tilde{f}(\boldsymbol{\omega}) \, d\boldsymbol{\omega}, \tag{23}$$

where $\tilde{f}(\boldsymbol{\omega}) \in \mathbb{C}$. Assume that $f(\mathbf{0}) = 0$ and that the following quantity is finite

$$C_f = \int_{\mathbb{R}^d} \|\boldsymbol{\omega}\|_2 |\tilde{f}(\boldsymbol{\omega})| \, d\boldsymbol{\omega}. \tag{24}$$

[12] uncovers the following dimension-free approximation guarantee.

Theorem 3 (Proposition 1 in [12]). *Fix a $C > 0$ and an arbitrary probability measure μ on the unit ball B^d in $\mathbb{R}^d$. For every function f with $C_f \le C$ and every $N \ge 1$, there exists some $g \in \mathcal{H}_N$ such that*

$$\left[\int_{B^d} (f(\boldsymbol{x}) - g(\boldsymbol{x}))^2 \, \mu(d\boldsymbol{x}) \right]^{1/2} \le \frac{2C}{\sqrt{N}}.$$

Moreover, the coefficients of g may be restricted to satisfy $\sum_{j=1}^N |c_j| \le 2C$.

The upper bound is now independent of the dimension d. However, C_f may implicitly depend on d, as the formula in (24) involves an integration over $\mathbb{R}^d$ (so for some functions C_f may depend exponentially on d). Nevertheless, this theorem does characterize an interesting function space with an improved upper bound. Details of the function space are discussed by [12]. This theorem can be generalized; see [81] for an example.

To help understand why a dimensionality-free approximation holds, let us appeal to a heuristic argument given by Monte Carlo simulations. It is well-known that Monte Carlo approximation errors are independent of dimensionality in evaluation of high-dimensional integrals. Let us generate $\{\boldsymbol{\omega}_j\}_{1 \le j \le N}$ randomly from a given density $p(\cdot)$ in $\mathbb{R}^d$. Consider the approximation to (23) by

$$g_N(\boldsymbol{x}) = \frac{1}{N} \sum_{j=1}^N c_j e^{i\langle \boldsymbol{\omega}_j, \boldsymbol{x} \rangle}, \qquad c_j = \frac{\tilde{f}(\boldsymbol{\omega}_j)}{p(\boldsymbol{\omega}_j)}. \tag{25}$$

Then, $g_N(\boldsymbol{x})$ is a one-hidden-layer neural network with N units and the sinusoid activation function. Note that $\mathbb{E} g_N(\boldsymbol{x}) = f(\boldsymbol{x})$, where the expectation is taken with respect to randomness $\{\boldsymbol{\omega}_j\}$. Now, by independence, we have

$$\mathbb{E}(g_N(\boldsymbol{x}) - f(\boldsymbol{x}))^2 = \frac{1}{N} \mathrm{Var}(c_j e^{i\langle \boldsymbol{\omega}_j, \boldsymbol{x} \rangle}) \le \frac{1}{N} \mathbb{E} c_j^2,$$

if $\mathbb{E} c_j^2 < \infty$. Therefore, the rate is independent of the dimensionality d, though the constant can be.

5.2 Approximation theory for multi-layer NNs

The approximation theory for multilayer neural nets is less understood compared with neural nets with one hidden layer. Driven by the success of deep learning, there are many recent papers focusing on expressivity of deep neural nets. As studied by [125, 35, 84, 94, 15, 111, 77, 103], deep neural nets excel at representing *composition* of functions. This is perhaps not surprising, since deep neural nets are themselves defined by composing layers of functions. Nevertheless, it points to a new territory rarely studied in statistics before. Below we present a result based on [77, 103].

Suppose that the inputs $\boldsymbol{x}$ have a bounded domain $[-1, 1]^d$ for simplicity. As before, let $\sigma_* : \mathbb{R} \to \mathbb{R}$ be a generic function, and $\boldsymbol{\sigma}_* = (\sigma_*, \cdots, \sigma_*)^\top$ be element-wise application of σ_*. Consider a neural net which is similar to (3) but with scaler output: $g(\boldsymbol{x}) = \mathbf{W}_\ell \boldsymbol{\sigma}_*(\cdots \boldsymbol{\sigma}_*(\mathbf{W}_2 \boldsymbol{\sigma}_*(\mathbf{W}_1 \boldsymbol{x})) \cdots)$. A unit or neuron refers to an element of vectors $\boldsymbol{\sigma}_*(\mathbf{W}_k \cdots \boldsymbol{\sigma}_*(\mathbf{W}_2 \boldsymbol{\sigma}_*(\mathbf{W}_1 \boldsymbol{x})) \cdots)$ for any $k = 1, \ldots, \ell - 1$. For a multivariate polynomial p, define $m_k(p)$ to be the smallest integer such that, for any $\epsilon > 0$, there exists a neural net $g(\boldsymbol{x})$ satisfying $\sup_{\boldsymbol{x}} |p(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$, with k hidden layers (i.e., $\ell = k + 1$) and no more than $m_k(p)$ neurons in total. Essentially, $m_k(p)$ is the minimum number of neurons required to approximate p arbitrarily well.

Theorem 4 (Theorem 4.1 in [103]). *Let $p(\boldsymbol{x})$ be a monomial $x_1^{r_1} x_2^{r_2} \cdots x_d^{r_d}$ with $q = \sum_{j=1}^d r_j$. Suppose that σ_* has derivatives of order $2q$ at the origin, and that they are nonzero. Then,*
(i) $m_1(p) = \prod_{j=1}^d (r_j + 1)$;
(ii) $\min_k m_k(p) \le \sum_{j=1}^d (7\lceil \log_2(r_j) \rceil + 4)$.

This theorem reveals a sharp distinction between shallow networks (one hidden layer) and deep networks. To represent a monomial function, a shallow network requires exponentially many neurons in terms of the dimension d, whereas linearly many neurons suffice for a deep network (with bounded r_j). The exponential dependence on d, as shown in Theorem 4(i), is resonant with the curse of dimensionality widely seen in many fields; see [30]. One may ask: how does depth help? Depth circumvents this issue, at least for certain functions, by allowing us to represent function composition efficiently. Indeed, Theorem 4(ii) offers a nice result with clear intuitions: it is known that the product of two scalar inputs can be represented using 4 neurons [77], so by composing multiple products, we can express monomials with $O(d)$ neurons.

Recent advances in nonparametric regressions also support the idea that deep neural nets excel at representing composition of functions [15, 111]. In particular, [15] considered the nonparametric regression setting where we want to estimate a function $\hat{f}_n(\boldsymbol{x})$ from i.i.d. data $\mathcal{D}_n = \{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$. If the true regression function $f(\boldsymbol{x})$ has certain hierarchical structure with intrinsic dimensionality[6] d^*, then the error

$$\mathbb{E}_{\mathcal{D}_n}\mathbb{E}_{\boldsymbol{x}}\left|\hat{f}_n(\boldsymbol{x}) - f(\boldsymbol{x})\right|^2$$

has an optimal minimax convergence rate $O(n^{-\frac{2q}{2q+d^*}})$, rather than the usual rate $O(n^{-\frac{2q}{2q+d}})$ that depends on the ambient dimension d. Here q is the smoothness parameter. This provides another justification for deep neural nets: if data are truly hierarchical, then the quality of approximators by deep neural nets depends on the intrinsic dimensionality, which avoids the curse of dimensionality.

We point out that the approximation theory for deep learning is far from complete. For example, in Theorem 4, the condition on σ_* excludes the widely used ReLU activation function, there are no constraints on the magnitude of the weights (so they can be unreasonably large).

6 Training deep neural nets

The *existence* of a good function approximator in the NN function class does not explain why in practice we can easily *find* them. In this section, we introduce standard methods, namely *stochastic gradient descent* (SGD) and its variants, to train deep neural networks (or to find such a good approximator). As with many statistical machine learning tasks, training DNNs follows the *empirical risk minimization* (ERM) paradigm which solves the following optimization problem

$$\text{minimize}_{\boldsymbol{\theta}\in\mathbb{R}^p} \qquad \ell_n(\boldsymbol{\theta}) \triangleq \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}\left(f\left(\boldsymbol{x}_i;\boldsymbol{\theta}\right), y_i\right). \tag{26}$$

Here $\mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ measures the discrepancy between the prediction $f(\boldsymbol{x}_i;\boldsymbol{\theta})$ of the neural network and the true label y_i. Correspondingly, denote by $\ell(\boldsymbol{\theta}) \triangleq \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}[\mathcal{L}(f(\boldsymbol{x};\boldsymbol{\theta}), \boldsymbol{y})]$ the out-of-sample error, where $\mathcal{D}$ is the joint distribution over $(y, \boldsymbol{x})$. Solving ERM (26) for deep neural nets faces various challenges that roughly fall into the following three categories.

- *Scalability and nonconvexity.* Both the sample size n and the number of parameters p can be huge for modern deep learning applications, as we have seen in Table 1. Many optimization algorithms are not practical due to the computational costs and memory constraints. What is worse, the empirical loss function $\ell_n(\boldsymbol{\theta})$ in deep learning is often nonconvex. It is *a priori* not clear whether an optimization algorithm can drive the empirical loss (26) small.

- *Numerical stability.* With a large number of layers in DNNs, the magnitudes of the hidden nodes can be drastically different, which may result in the "exploding gradients" or "vanishing gradients" issue during the training process. This is because the recursive relations across layers often lead to exponentially increasing / decreasing values in both forward passes and backward passes.

- *Generalization performance.* Our ultimate goal is to find a parameter $\hat{\boldsymbol{\theta}}$ such that the out-of-sample error $\ell(\hat{\boldsymbol{\theta}})$ is small. However, in the over-parametrized regime where p is much larger than n, the underlying

[6] Roughly speaking, the true regression function can be represented by a tree where each node has at most d^* children. See [15] for the precise definition.

neural network has the potential to fit the training data perfectly while performing poorly on the test data. To avoid this overfitting issue, proper regularization, whether explicit or implicit, is needed in the training process for the neural nets to generalize.

In the following three subsections, we discuss practical solutions / proposals to address these challenges.

6.1 Stochastic gradient descent

Stochastic gradient descent (SGD) [101] is by far the most popular optimization algorithm to solve ERM (26) for large-scale problems. It has the following simple update rule:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t G(\boldsymbol{\theta}^t) \qquad \text{with} \qquad G\left(\boldsymbol{\theta}^t\right) = \nabla \mathcal{L}\left(f\left(\boldsymbol{x}_{i_t}; \boldsymbol{\theta}^t\right), y_{i_t}\right) \tag{27}$$

for $t = 0, 1, 2, \ldots$, where $\eta_t > 0$ is the step size (or learning rate), $\boldsymbol{\theta}^0 \in \mathbb{R}^p$ is an initial point and i_t is chosen randomly from $\{1, 2, \cdots, n\}$. It is easy to verify that $G(\boldsymbol{\theta}^t)$ is an unbiased estimate of $\nabla \ell_n(\boldsymbol{\theta}^t)$. The advantage of SGD is clear: compared with gradient descent, which goes over the entire dataset in every update, SGD uses a single example in each update and hence is considerably more efficient in terms of both computation and memory (especially in the first few iterations).

Apart from practical benefits of SGD, how well does SGD perform theoretically in terms of minimizing $\ell_n(\boldsymbol{\theta})$? We begin with the convex case, i.e., the case where the loss function is convex w.r.t. $\boldsymbol{\theta}$. It is well understood in literature that with proper choices of the step sizes $\{\eta_t\}$, SGD is guaranteed to achieve both *consistency* and *asymptotic normality*.

- *Consistency.* If $\ell(\boldsymbol{\theta})$ is a strongly convex function[7], then under some mild conditions[8], learning rates that satisfy

$$\sum_{t=0}^{\infty} \eta_t = +\infty \qquad \text{and} \qquad \sum_{t=0}^{\infty} \eta_t^2 < +\infty \tag{28}$$

  guarantee almost sure convergence to the unique minimizer $\boldsymbol{\theta}^* \triangleq \operatorname{argmin}_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta})$, i.e., $\boldsymbol{\theta}^t \xrightarrow{\text{a.s.}} \boldsymbol{\theta}^*$ as $t \to \infty$ [101, 64, 16, 69]. The requirements in (28) can be viewed from the perspective of bias-variance tradeoff: the first condition ensures that the iterates can reach the minimizer (controlled bias), and the second ensures that stochasticity does not prevent convergence (controlled variance).

- *Asymptotic normality.* It is proved by [97] that for robust linear regression with fixed dimension p, under the choice $\eta_t = t^{-1}$, $\sqrt{t}\left(\boldsymbol{\theta}^t - \boldsymbol{\theta}^*\right)$ is asymptotically normal under some regularity conditions (but $\boldsymbol{\theta}^t$ is not asymptotically efficient in general). Moreover, by averaging the iterates of SGD, [96] proved that even with a *larger* step size $\eta_t \propto t^{-\alpha}, \alpha \in (1/2, 1)$, the averaged iterate $\bar{\boldsymbol{\theta}}^t = t^{-1} \sum_{s=1}^{t} \boldsymbol{\theta}^s$ is asymptotic efficient for robust linear regression. These strong results show that SGD with averaging performs as well as the MLE asymptotically, in addition to its computational efficiency.

These classical results, however, fail to explain the effectiveness of SGD when dealing with nonconvex loss functions in deep learning. Admittedly, finding global minima of nonconvex functions is computationally infeasible in the worst case. Nevertheless, recent work [4, 32] bypasses the worst case scenario by focusing on losses incurred by over-parametrized deep learning models. In particular, they show that (stochastic) gradient descent converges linearly towards the *global* minimizer of $\ell_n(\boldsymbol{\theta})$ as long as the neural network is sufficiently *over-parametrized*. This phenomenon is formalized below.

Theorem 5 (Theorem 2 in [4]). *Let $\{(y_i, \boldsymbol{x}_i)\}_{1 \le i \le n}$ be a training set satisfying $\min_{i,j:i \neq j} \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \ge \delta > 0$. Consider fitting the data using a feed-forward neural network (1) with ReLU activations. Denote by L (resp. W) the depth (resp. width) of the network. Suppose that the neural network is sufficiently over-parametrized, i.e.,*

$$W \gg \mathsf{poly}\left(n, L, \frac{1}{\delta}\right), \tag{29}$$

[7] For results on consistency and asymptotic normality, we consider the case where in each step of SGD, the stochastic gradient is computed using a fresh sample $(y, \boldsymbol{x})$ from $\mathcal{D}$. This allows to view SGD as an optimization algorithm to minimize the population loss $\ell(\boldsymbol{\theta})$.

[8] One example of such condition can be constraining the second moment of the gradients: $\mathbb{E}\left[\|\nabla \mathcal{L}\left(\boldsymbol{x}_i, y_i; \boldsymbol{\theta}^t\right)\|_2^2\right] \le C_1 + C_2 \|\boldsymbol{\theta}^t - \boldsymbol{\theta}^*\|_2^2$ for some $C_1, C_2 > 0$. See [16] for details.

where poly *means a polynomial function. Then with high probability, running SGD (27) with* certain random initialization *and properly chosen step sizes yields* $\ell_n(\boldsymbol{\theta}^t) \le \varepsilon$ *in* $t \asymp \log \frac{1}{\varepsilon}$ *iterations.*

Two notable features are worth mentioning: (1) first, the network under consideration is sufficiently over-parametrized (cf. (29)) in which the number of parameters is *much* larger than the number of samples, and (2) one needs to initialize the weight matrices to be in near-isometry such that the magnitudes of the hidden nodes do not blow up or vanish. In a nutshell, *over-parametrization* and *random initialization* together ensure that the loss function (26) has a benign landscape[9] around the initial point, which in turn implies fast convergence of SGD iterates.

There are certainly other challenges for vanilla SGD to train deep neural nets: (1) training algorithms are often implemented in GPUs, and therefore it is important to tailor the algorithm to the infrastructure, (2) the vanilla SGD might converge very slowly for deep neural networks, albeit good theoretical guarantees for well-behaved problems, and (3) the learning rates $\{\eta_t\}$ can be difficult to tune in practice. To address the aforementioned challenges, three important variants of SGD, namely *mini-batch SGD, momentum-based SGD*, and *SGD with adaptive learning rates* are introduced.

6.1.1 Mini-batch SGD

Modern computational infrastructures (e.g., GPUs) can evaluate the gradient on a number (say 64) of examples as efficiently as evaluating that on a single example. To utilize this advantage, mini-batch SGD with batch size $K \ge 1$ forms the stochastic gradient through K random samples:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t G(\boldsymbol{\theta}^t) \qquad \text{with} \qquad G(\boldsymbol{\theta}^t) = \frac{1}{K}\sum_{k=1}^{K} \nabla \mathcal{L}\big(f\big(\boldsymbol{x}_{i_t^k}; \boldsymbol{\theta}^t\big), y_{i_t^k}\big), \tag{30}$$

where for each $1 \le k \le K$, i_t^k is sampled uniformly from $\{1, 2, \cdots, n\}$. Mini-batch SGD, which is an "interpolation" between gradient descent and stochastic gradient descent, achieves the best of both worlds: (1) using $1 \ll K \ll n$ samples to estimate the gradient, one effectively reduces the variance and hence accelerates the convergence, and (2) by taking the batch size K appropriately (say 64 or 128), the stochastic gradient $G(\boldsymbol{\theta}^t)$ can be efficiently computed using the matrix computation toolboxes on GPUs.

6.1.2 Momentum-based SGD

While mini-batch SGD forms the foundation of training neural networks, it can sometimes be slow to converge due to its oscillation behavior [122]. Optimization community has long investigated how to accelerate the convergence of gradient descent, which results in a beautiful technique called *momentum methods* [95, 88]. Similar to gradient descent with moment, *momentum-based SGD*, instead of moving the iterate $\boldsymbol{\theta}^t$ in the direction of the current stochastic gradient $G(\boldsymbol{\theta}^t)$, smooth the past (stochastic) gradients $\{G(\boldsymbol{\theta}^t)\}$ to stabilize the update directions. Mathematically, let $\boldsymbol{v}^t \in \mathbb{R}^p$ be the direction of update in the tth iteration, i.e.,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t \boldsymbol{v}^t.$$

Here $\boldsymbol{v}^0 = G(\boldsymbol{\theta}^0)$ and for $t = 1, 2, \cdots$

$$\boldsymbol{v}^t = \rho \boldsymbol{v}^{t-1} + G(\boldsymbol{\theta}^t) \tag{31}$$

with $0 < \rho < 1$. A typical choice of ρ is 0.9. Notice that $\rho = 0$ recovers the mini-batch SGD (30), where no past information of gradients is used. A simple unrolling of $\boldsymbol{v}^t$ reveals that $\boldsymbol{v}^t$ is actually an exponential averaging of the past gradients, i.e., $\boldsymbol{v}^t = \sum_{j=0}^{t} \rho^{t-j} G(\boldsymbol{\theta}^j)$. Compared with vanilla mini-batch SGD, the inclusion of the momentum "smoothes" the oscillation direction and accumulates the persistent descent direction. We want to emphasize that theoretical justifications of momentum in the *stochastic* setting is not fully understood [63, 60].

[9] In [4], the loss function $\ell_n(\boldsymbol{\theta})$ satisfies the PL condition.

6.1.3 SGD with adaptive learning rates

In optimization, *preconditioning* is often used to accelerate first-order optimization algorithms. In principle, one can apply this to SGD, which yields the following update rule:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta_t \boldsymbol{P}_t^{-1} G(\boldsymbol{\theta}^t) \tag{32}$$

with $\boldsymbol{P}_t \in \mathbb{R}^{p \times p}$ being a preconditioner at the t-th step. Newton's method can be viewed as one type of preconditioning where $\boldsymbol{P}_t = \nabla^2 \ell(\boldsymbol{\theta}^t)$. The advantages of preconditioning are two-fold: first, a good preconditioner reduces the condition number by changing the local geometry to be more homogeneous, which is amenable to fast convergence; second, a good preconditioner frees practitioners from laboring tuning of the step sizes, as is the case with Newton's method. AdaGrad, an adaptive gradient method proposed by [33], builds a preconditioner $\boldsymbol{P}_t$ based on information of the past gradients:

$$\boldsymbol{P}_t = \Big\{ \mathsf{diag}\Big(\sum_{j=0}^{t} G\left(\boldsymbol{\theta}^t\right) G\left(\boldsymbol{\theta}^t\right)^\top \Big) \Big\}^{1/2}. \tag{33}$$

Since we only require the diagonal part, this preconditioner (and its inverse) can be efficiently computed in practice. In addition, investigating (32) and (33), one can see that AdaGrad adapts to the importance of each coordinate of the parameters by setting smaller learning rates for frequent features, whereas larger learning rates for those infrequent ones. In practice, one adds a small quantity $\delta > 0$ (say 10^{-8}) to the diagonal entries to avoid singularity (numerical underflow). A notable drawback of AdaGrad is that the effective learning rate vanishes quickly along the learning process. This is because the historical sum of the gradients can only increase with time. RMSProp [52] is a popular remedy for this problem which incorporates the idea of exponential averaging:

$$\boldsymbol{P}_t = \Big\{ \mathsf{diag}\Big(\rho \boldsymbol{P}_{t-1} + (1-\rho) G\left(\boldsymbol{\theta}^t\right) G\left(\boldsymbol{\theta}^t\right)^\top \Big) \Big\}^{1/2}. \tag{34}$$

Again, the decaying parameter ρ is usually set to be 0.9. Later, Adam [65, 100] combines the momentum method and adaptive learning rate and becomes the default training algorithms in many deep learning applications.

6.2 Easing numerical instability

For very deep neural networks or RNNs with long dependencies, training difficulties often arise when the values of nodes have different magnitudes or when the gradients "vanish" or "explode" during back-propagation. Here we discuss three partial solutions to alleviate this problem.

6.2.1 ReLU activation function

One useful characteristic of the ReLU function is that its derivative is either 0 or 1, and the derivative remains 1 even for a large input. This is in sharp contrast with the standard sigmoid function $(1+e^{-t})^{-1}$ which results in a very small derivative when inputs have large magnitude. The consequence of small derivatives across many layers is that gradients tend to be "killed", which means that gradients become approximately zero in deep nets.

The popularity of the ReLU activation function and its variants (e.g., leaky ReLU) is largely attributable to the above reason. It has been well observed that the ReLU activation function has superior training performance over the sigmoid function [68, 79].

6.2.2 Skip connections

We have introduced skip connections in Section 3.3. Why are skip connections helpful for reducing numerical instability? This structure does not introduce a larger function space, since the identity map can be also represented with ReLU activations: $\boldsymbol{x} = \boldsymbol{\sigma}(\boldsymbol{x}) - \boldsymbol{\sigma}(-\boldsymbol{x})$.

One explanation is that skip connections bring ease to the training / optimization process. Suppose that we have a general nonlinear function $\mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)$. With a skip connection, we represent the map as $\boldsymbol{x}_{\ell+1} = \boldsymbol{x}_\ell + \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)$ instead. Now the gradient $\partial \boldsymbol{x}_{\ell+1}/\partial \boldsymbol{x}_\ell$ becomes

$$\frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell} = \mathbf{I} + \frac{\partial \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)}{\partial \boldsymbol{x}_\ell} \qquad \text{instead of} \qquad \frac{\partial \mathbf{F}(\boldsymbol{x}_\ell; \boldsymbol{\theta}_\ell)}{\partial \boldsymbol{x}_\ell}, \tag{35}$$

where $\mathbf{I}$ is an identity matrix. By the chain rule, gradient update requires computing products of many components, e.g., $\frac{\partial \boldsymbol{x}_L}{\partial \boldsymbol{x}_1} = \prod_{\ell=1}^{L-1} \frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell}$, so it is desirable to keep the spectra (singular values) of each component $\frac{\partial \boldsymbol{x}_{\ell+1}}{\partial \boldsymbol{x}_\ell}$ close to 1. In neural nets, with skip connections, this is easily achieved if the parameters have small values; otherwise, this may not be achievable even with careful initialization and tuning. Notably, training neural nets with hundreds of layers is possible with the help of skip connections.

6.2.3 Batch normalization

Recall that in regression analysis, one often standardizes the design matrix so that the features have zero mean and unit variance. Batch normalization extends this standardization procedure from the input layer to all the hidden layers. Mathematically, fix a mini-batch of input data $\{(\boldsymbol{x}_i, y_i)\}_{i\in\mathcal{B}}$, where $\mathcal{B} \subset [n]$. Let $\boldsymbol{h}_i^{(\ell)}$ be the feature of the i-th example in the ℓ-th layer ($\ell = 0$ corresponds to the input $\boldsymbol{x}_i$). The batch normalization layer computes the normalized version of $\boldsymbol{h}_i^{(\ell)}$ via the following steps:

$$\boldsymbol{\mu} \triangleq \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \boldsymbol{h}_i^{(\ell)}, \qquad \boldsymbol{\sigma}^2 \triangleq \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \big(\boldsymbol{h}_i^{(\ell)} - \boldsymbol{\mu}\big)^2 \qquad \text{and} \qquad \boldsymbol{h}_{i,\text{norm}}^{(l)} \triangleq \frac{\boldsymbol{h}_i^{(\ell)} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}.$$

Here all the operations are element-wise. In words, batch normalization computes the z-score for each feature over the mini-batch $\mathcal{B}$ and use that as inputs to subsequent layers. To make it more versatile, a typical batch normalization layer has two additional learnable parameters $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{\beta}^{(\ell)}$ such that

$$\boldsymbol{h}_{i,\text{new}}^{(l)} = \boldsymbol{\gamma}^{(l)} \odot \boldsymbol{h}_{i,\text{norm}}^{(l)} + \boldsymbol{\beta}^{(l)}.$$

Again $\odot$ denotes the element-wise multiplication. As can be seen, $\boldsymbol{\gamma}^{(\ell)}$ and $\boldsymbol{\beta}^{(\ell)}$ set the new feature $\boldsymbol{h}_{i\text{new}}^{(l)}$ to have mean $\boldsymbol{\beta}^{(\ell)}$ and standard deviation $\boldsymbol{\gamma}^{(\ell)}$. The introduction of batch normalization makes the training of neural networks much easier and smoother. More importantly, it allows the neural nets to perform well over a large family of hyper-parameters including the number of layers, the number of hidden units, etc. At test time, the batch normalization layer needs more care. For brevity we omit the details and refer to [58].

6.3 Regularization techniques

So far we have focused on training techniques to drive the empirical loss (26) small efficiently. Here we proceed to discuss common practice to improve the generalization power of trained neural nets.

6.3.1 Weight decay

One natural regularization idea is to add an ℓ_2 penalty to the loss function. This regularization technique is known as the weight decay in deep learning. We have seen one example in (9). For general deep neural nets, the loss to optimize is $\ell_n^\lambda(\boldsymbol{\theta}) = \ell_n(\boldsymbol{\theta}) + r_\lambda(\boldsymbol{\theta})$ where

$$r_\lambda(\boldsymbol{\theta}) = \lambda \sum_{\ell=1}^{L} \sum_{j,j'} \big[W_{j,j'}^{(\ell)}\big]^2.$$

Note that the bias (intercept) terms are not penalized. If $\ell_n(\boldsymbol{\theta})$ is a least square loss, then regularization with weight decay gives precisely ridge regression. The penalty $r_\lambda(\boldsymbol{\theta})$ is a smooth function and thus it can be also implemented efficiently with back-propagation.

6.3.2 Dropout

Dropout, introduced by [53], prevents overfitting by randomly dropping out subsets of features during training. Take the l-th layer of the feed-forward neural network as an example. Instead of propagating all the features in $\boldsymbol{h}^{(\ell)}$ for later computations, dropout randomly omits some of its entries by

$$\boldsymbol{h}^{(\ell)}_{\text{drop}} = \boldsymbol{h}^{(\ell)} \odot \mathsf{mask}^{\ell},$$

where $\odot$ denotes element-wise multiplication as before, and mask^{ℓ} is a vector of Bernoulli variables with success probability p. It is sometimes useful to rescale the features $\boldsymbol{h}^{(\ell)}_{\text{inv drop}} = \boldsymbol{h}^{(\ell)}_{\text{drop}}/p$, which is called *inverted dropout*. During training, mask^{ℓ} are i.i.d. vectors across mini-batches and layers. However, when testing on fresh samples, dropout is disabled and the original features $\boldsymbol{h}^{(\ell)}$ are used to compute the output label y. It has been nicely shown by [129] that for generalized linear models, dropout serves as adaptive regularization. In the simplest case of linear regression, it is equivalent to ℓ_2 regularization. Another possible way to understand the regularization effect of dropout is through the lens of bagging [45]. Since different mini-batches has different masks, dropout can be viewed as training a large ensemble of classifiers at the same time, with a further constraint that the parameters are shared. Theoretical justification remains elusive.

6.3.3 Data augmentation

Data augmentation is a technique of enlarging the dataset when we have knowledge about invariance structure of data. It implicitly increases the sample size and usually regularizes the model effectively. For example, in image classification, we have strong prior knowledge about what invariance properties a good classifier should possess. The label of an image should not be affected by translation, rotation, flipping, and even crops of the image. Hence one can augment the dataset by randomly translating, rotating and cropping the images in the original dataset.

Formally, during training we want to minimize the loss $\ell_n(\boldsymbol{\theta}) = \sum_i \mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ w.r.t. parameters $\boldsymbol{\theta}$, and we know a priori that certain transformation $T \in \mathcal{T}$ where $T : \mathbb{R}^d \to \mathbb{R}^d$ (e.g., affine transformation) should not change the category / label of a training sample. In principle, if computation costs were not a consideration, we could convert this knowledge to a constraint $f_{\boldsymbol{\theta}}(T\boldsymbol{x}_i) = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \forall\, T \in \mathcal{T}$ in the minimization formulation. Instead of solving a constrained optimization problem, data augmentation enlarges the training dataset by sampling $T \in \mathcal{T}$ and generating new data $\{(T\boldsymbol{x}_i, y_i)\}$. In this sense, data augmentation induces invariance properties through sampling, which results in a much bigger dataset than the original one.

7 Generalization power

Section 6 has focused on the in-sample / training error obtained via SGD, but this alone does not guarantee good performance with respect to the out-of-sample / test error. The gap between the in-sample error and the out-of-sample error, namely the *generalization gap*, has been the focus of statistical learning theory since its birth; see [112] for an excellent introduction to this topic.

While understanding the generalization power of deep neural nets is difficult [135, 99], we sample recent endeavors in this section. From a high level point of view, these approaches can be divided into two categories, namely *algorithm-independent controls* and *algorithm-dependent controls*. More specifically, algorithm-independent controls focus solely on bounding the *complexity* of the function class represented by certain deep neural networks. In contrast, algorithm-dependent controls take into account the algorithm (e.g., SGD) used to train the neural network.

7.1 Algorithm-independent controls: uniform convergence

The key to algorithm-independent controls is the notion of *complexity* of the function class parametrized by certain neural networks. Informally, as long as the complexity is not too large, the generalization gap of *any* function in the function class is well-controlled. However, the standard complexity measure (e.g., VC dimension [127]) is at least proportional to the number of weights in a neural network [5, 112], which fails to explain the practical success of deep learning. The caveat here is that the function class under consideration

is *all* the functions realized by certain neural networks, with *no* restrictions on the size of the weights at all. On the other hand, for the class of linear functions with bounded norm, i.e., $\{\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x} \,|\, \|\boldsymbol{w}\|_2 \le M\}$, it is well understood that the complexity of this function class (measured in terms of the empirical Rademacher complexity) with respect to a random sample $\{\boldsymbol{x}_i\}_{1\le i\le n}$ is upper bounded by $\max_i \|\boldsymbol{x}_i\|_2 M/\sqrt{n}$, which is independent of the number of parameters in $\boldsymbol{w}$. This motivates researchers to investigate the complexity of *norm-controlled* deep neural networks[10] [89, 14, 43, 74]. Setting the stage, we introduce a few necessary notations and facts. The key object under study is the function class parametrized by the following fully-connected neural network with depth L:

$$\mathcal{F}_L \triangleq \left\{\boldsymbol{x} \mapsto \boldsymbol{W}_L \boldsymbol{\sigma}\left(\boldsymbol{W}_{L-1}\boldsymbol{\sigma}\left(\cdots \boldsymbol{W}_2 \boldsymbol{\sigma}\left(\boldsymbol{W}_1 \boldsymbol{x}\right)\right)\right) \,\middle|\, (\boldsymbol{W}_1, \cdots, \boldsymbol{W}_L) \in \mathcal{W}\right\}. \tag{36}$$

Here $(\boldsymbol{W}_1, \boldsymbol{W}_2, \cdots, \boldsymbol{W}_L) \in \mathcal{W}$ represents a certain constraint on the parameters. For instance, one can restrict the Frobenius norm of each parameter $\boldsymbol{W}_l$ through the constraint $\|\boldsymbol{W}_l\|_{\mathrm{F}} \le M_{\mathrm{F}}(l)$, where $M_{\mathrm{F}}(l)$ is some positive quantity. With regard to the complexity measure, it is standard to use *Rademacher complexity* to control the capacity of the function class of interest.

Definition 1 (Empirical Rademacher complexity)**.** *The empirical Rademacher complexity of a function class $\mathcal{F}$ w.r.t. a dataset $S \triangleq \{\boldsymbol{x}_i\}_{1\le i\le n}$ is defined as*

$$\mathcal{R}_S(\mathcal{F}) = \mathbb{E}_{\boldsymbol{\varepsilon}}\left[\sup_{f\in\mathcal{F}} \frac{1}{n}\sum_{i=1}^n \varepsilon_i f(\boldsymbol{x}_i)\right], \tag{37}$$

where $\boldsymbol{\varepsilon} \triangleq (\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n)$ is composed of i.i.d. Rademacher random variables, i.e., $\mathbb{P}(\varepsilon_i = 1) = \mathbb{P}(\varepsilon_i = -1) = 1/2$.

In words, Rademacher complexity measures the ability of the function class to fit the random noise represented by $\boldsymbol{\varepsilon}$. Intuitively, a function class with a larger Rademacher complexity is more prone to overfitting. We now formalize the connection between the empirical Rademacher complexity and the out-of-sample error; see Chapter 24 in [112].

Theorem 6. *Assume that for all $f \in \mathcal{F}$ and all $(y, \boldsymbol{x})$ we have $|\mathcal{L}(f(\boldsymbol{x}), y)| \le 1$. In addition, assume that for any fixed y, the univariate function $\mathcal{L}(\cdot, y)$ is Lipschitz with constant 1. Then with probability at least $1-\delta$ over the sample $S \triangleq \{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n} \overset{\text{i.i.d.}}{\sim} \mathcal{D}$, one has for all $f \in \mathcal{F}$*

$$\underbrace{\mathbb{E}_{(y,\boldsymbol{x})\sim\mathcal{D}}\left[\mathcal{L}\left(f(\boldsymbol{x}), y\right)\right]}_{\text{out-of-sample error}} \le \underbrace{\frac{1}{n}\sum_{i=1}^n \mathcal{L}\left(f(\boldsymbol{x}_i), y_i\right)}_{\text{in-sample error}} + 2\mathcal{R}_S(\mathcal{F}) + 4\sqrt{\frac{\log(4/\delta)}{n}}.$$

In English, the generalization gap of any function f that lies in $\mathcal{F}$ is well-controlled as long as the Rademacher complexity of $\mathcal{F}$ is not too large. With this connection in place, we single out the following complexity bound.

Theorem 7 (Theorem 1 in [43])**.** *Consider the function class $\mathcal{F}_L$ in (36), where each parameter $\boldsymbol{W}_l$ has Frobenius norm at most $M_{\mathrm{F}}(l)$. Further suppose that the element-wise activation function $\sigma(\cdot)$ is 1-Lipschitz and positive-homogeneous (i.e., $\sigma(c\cdot x) = c\sigma(x)$ for all $c \ge 0$). Then the empirical Rademacher complexity (37) w.r.t. $S \triangleq \{\boldsymbol{x}_i\}_{1\le i\le n}$ satisfies*

$$\mathcal{R}_S(\mathcal{F}_L) \le \max_i \|\boldsymbol{x}_i\|_2 \cdot \frac{4\sqrt{L}\prod_{l=1}^L M_{\mathrm{F}}(l)}{\sqrt{n}}. \tag{38}$$

The upper bound of the empirical Rademacher complexity (38) is in a similar vein to that of linear functions with bounded norm, i.e., $\max_i \|\boldsymbol{x}_i\|_2 M/\sqrt{n}$, where $\sqrt{L}\prod_{l=1}^L M_{\mathrm{F}}(l)$ plays the role of M in the latter case. Moreover, ignoring the term $\sqrt{L}$, the upper bound (38) does not depend on the size of the network in an explicit way if $M_F(l)$ sharply concentrates around 1. This reveals that the capacity of the

[10] Such attempts have been made in the seminal work [13].

neural network is well-controlled, regardless of the number of parameters, as long as the Frobenius norm of the parameters is bounded. Extensions to other norm constraints, e.g., spectral norm constraints, path norm constraints have been considered by [89, 14, 74, 67, 34]. This line of work improves upon traditional capacity analysis of neural networks in the over-parametrized setting, because the upper bounds derived are often size-independent. Having said this, two important remarks are in order: (1) the upper bounds (e.g., $\prod_{l=1}^{L} M_{\mathrm{F}}(l)$) involve implicit dependence on the size of the weight matrix and the depth of the neural network, which is hard to characterize; (2) the upper bound on the Rademacher complexity offers a uniform bound over all functions in the function class, which is a pure statistical result. However, it stays silent about how and why standard training algorithms like SGD can obtain a function whose parameters have small norms.

7.2 Algorithm-dependent controls

In this subsection, we bring computational thinking into statistics and investigate the role of algorithms in the generalization power of deep learning. The consideration of algorithms is quite natural and well motivated: (1) local/global minima reached by different algorithms can exhibit totally different generalization behaviors due to extreme nonconvexity, which marks a huge difference from traditional models, (2) the *effective* capacity of neural nets is possibly not large, since a particular algorithm does not explore the entire parameter space.

These demonstrate the fact that on top of the complexity of the function class, the inherent property of the algorithm we use plays an important role in the generalization ability of deep learning. In what follows, we survey three different ways to obtain upper bounds on the generalization errors by exploiting properties of the algorithms.

7.2.1 Mean field view of neural nets

As we have emphasized, modern deep learning models are highly over-parametrized. A line of work [83, 117, 105, 25, 82, 61] approximates the ensemble of weights by an asymptotic limit as the number of hidden units tends to infinity, so that the dynamics of SGD can be studied via certain partial different equations.

More specifically, let $\hat{f}(\boldsymbol{x};\boldsymbol{\theta}) = N^{-1}\sum_{i=1}^{N}\sigma(\boldsymbol{\theta}_i^{\top}\boldsymbol{x})$ be a function given by a one-hidden-layer neural net with N hidden units, where $\sigma(\cdot)$ is the ReLU activation function and parameters $\boldsymbol{\theta} \triangleq [\boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_N]^{\top} \in \mathbb{R}^{N\times d}$ are suitably randomly initialized. Consider the regression setting where we want to minimize the population risk $R_N(\boldsymbol{\theta}) = \mathbb{E}[(y - \hat{f}(\boldsymbol{x};\boldsymbol{\theta}))^2]$ over parameters $\boldsymbol{\theta}$. A key observation is that this population risk depends on the parameters $\boldsymbol{\theta}$ only through its empirical distribution, i.e., $\hat{\rho}^{(N)} = N^{-1}\sum_{i=1}^{N}\delta_{\boldsymbol{\theta}_i}$ where $\delta_{\boldsymbol{\theta}_i}$ is a point mass at $\boldsymbol{\theta}_i$. This motivates us to view express $R_N(\boldsymbol{\theta})$ equivalently as $R(\hat{\rho}^{(N)})$, where $R(\cdot)$ is a functional that maps distributions to real numbers. Running SGD for $R_N(\cdot)$—in a suitable scaling limit—results in a gradient flow on the space of distributions endowed with the Wasserstein metric that minimizes $R(\cdot)$. It turns out that the empirical distribution $\hat{\rho}_k^{(N)}$ of the parameters after k steps of SGD is well approximated by the gradient follow, as long as the the neural net is over-parametrized (i.e., $N \gg d$) and the number of steps is not too large. In particular, [83] have shown that under certain regularity conditions,

$$\sup_{k\in[0,T/\varepsilon]\cap\mathbb{N}} \left| R(\hat{\rho}^{(N)}) - R\left(\rho_{k\varepsilon}\right) \right| \lesssim e^{T}\sqrt{\frac{1}{N}\vee\varepsilon}\cdot\sqrt{d+\log\frac{N}{\varepsilon}},$$

where $\varepsilon > 0$ is an proxy for the step size of SGD and $\rho_{k\varepsilon}$ is the distribution of the gradient flow at time $k\varepsilon$. In words, the out-of-sample error under $\boldsymbol{\theta}^k$ generated by SGD is well-approximated by that of $\rho_{k\varepsilon}$. Viewing the optimization problem from the distributional aspect greatly simplifies the problem conceptually, as the complicated optimization problem is now passed into its limit version—for this reason, this analytical approach is called the mean field perspective. In particular, [83] further demonstrated that in some simple settings, the out-of-sample error $R(\rho_{k\varepsilon})$ of the distributional limit can be fully characterized. Nevertheless, how well does $R(\rho_{k\varepsilon})$ perform and how fast it converges remain largely open for general problems.

7.2.2 Stability

A second way to understand the generalization ability of deep learning is through the *stability* of SGD. An algorithm is considered stable if a slight change of the input does not alter the output much. It has long been

observed that a stable algorithm has a small generalization gap; examples include k nearest neighbors [102, 29], bagging [18, 19], etc. The precise connection between stability and generalization gap is stated by [17, 113]. In what follows, we formalize the idea of *stability* and its connection with the generalization gap. Let $\mathcal{A}$ denote an algorithm (possibly randomized) which takes a sample $S \triangleq \{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ of size n and returns an estimated parameter $\hat{\boldsymbol{\theta}} \triangleq \mathcal{A}(S)$. Following [49], we have the following definition for *stability*.

Definition 2. *An algorithm (possibly randomized) $\mathcal{A}$ is ε-uniformly stable with respect to the loss function $\mathcal{L}(\cdot,\cdot)$ if for all datasets S, S' of size n which differ in at most one example, one has*

$$\sup_{\boldsymbol{x},y} \mathbb{E}_{\mathcal{A}} \left[\mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S\right)), y\right) - \mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S'\right)), y\right)\right] \le \varepsilon.$$

Here the expectation is taken w.r.t. the randomness in the algorithm $\mathcal{A}$ and ε might depend on n. The loss function $\mathcal{L}(\cdot,\cdot)$ takes an example (say $(\boldsymbol{x}, y)$) and the estimated parameter (say $\mathcal{A}(S)$) as inputs and outputs a real value.

Surprisingly, an ε-uniformly stable algorithm incurs small generalization gap *in expectation*, which is stated in the following lemma.

Lemma 1 (Theorem 2.2 in [49])**.** *Let $\mathcal{A}$ be ε-uniformly stable. Then the expected generalization gap is no larger than ε, i.e.,*

$$\left|\mathbb{E}_{\mathcal{A},S}\left[\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(f(\boldsymbol{x}_i; \mathcal{A}\left(S\right)), y_i) - \mathbb{E}_{(\boldsymbol{x},y)\sim\mathcal{D}}\left[\mathcal{L}\left(f(\boldsymbol{x}; \mathcal{A}\left(S\right)), y\right)\right]\right]\right| \le \varepsilon. \tag{39}$$

With Lemma 1 in hand, it suffices to prove stability bound on specific algorithms. It turns out that SGD introduced in Section 6 is uniformly stable when solving smooth nonconvex functions.

Theorem 8 (Theorem 3.12 in [49])**.** *Assume that for any fixed $(y, \boldsymbol{x})$, the loss function $\mathcal{L}(f(x;\boldsymbol{\theta}), y)$, viewed as a function of $\boldsymbol{\theta}$, is L-Lipschitz and β-smooth. Consider running SGD on the empirical loss function with decaying step size $\alpha_t \le c/t$, where c is some small absolute constant. Then SGD is uniformly stable with*

$$\varepsilon \lesssim \frac{T^{1-\frac{1}{\beta c+1}}}{n},$$

where we have ignored the dependency on β, c and L.

Theorem 8 reveals that SGD operating on nonconvex loss functions is indeed uniformly stable as long as the number of steps T is not large compared with n. This together with Lemma 1 demonstrates the generalization ability of SGD in expectation. Nevertheless, two important limitations are worth mentioning. First, Lemma 1 provides an upper bound on the out-of-sample error *in expectation*, but ideally, instead of an on-average guarantee under $\mathbb{E}_{\mathcal{A},S}$, we would like to have a high probability guarantee as in the convex case [37]. Second, controlling the generalization gap alone is not enough to achieve a small out-of-sample error, since it is unclear whether SGD can achieve a small training error within T steps.

7.2.3 Implicit regularization

In the presence of over-parametrization (number of parameters larger than the sample size), conventional wisdom informs us that we should apply some regularization techniques (e.g., ℓ_1 / ℓ_2 regularization) so that the model will not overfit the data. However, in practice, neural networks without explicit regularization generalize well. This phenomenon motivates researchers to look at the regularization effects introduced by training algorithms (e.g., SGD) in this over-parametrized regime. While there might exits multiple, if not infinite global minima of the empirical loss (26), it is possible that practical algorithms tend to converge to solutions with better generalization powers.

Take the underdetermined linear system $\boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{y}$ as a starting point. Here $\boldsymbol{X} \in \mathbb{R}^{n\times p}$ and $\boldsymbol{\theta} \in \mathbb{R}^p$ with p much larger than n. Running gradient descent on the loss $\frac{1}{2}\|\boldsymbol{X}\boldsymbol{\theta} - \boldsymbol{y}\|_2^2$ from the origin (i.e., $\boldsymbol{\theta}^0 = \boldsymbol{0}$) results in the solution with the minimum Euclidean norm, that is GD converges to

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \quad \|\boldsymbol{\theta}\|_2 \qquad \text{subject to} \quad \boldsymbol{X}\boldsymbol{\theta} = \boldsymbol{y}.$$

In words, without any ℓ_2 regularization in the loss function, gradient descent automatically finds the solution with the least ℓ_2 norm. This phenomenon, often called as *implicit regularization*, not only has been empirically observed in training neural networks, but also has been theoretically understood in some simplified cases, e.g., logistic regression with separable data. In logistic regression, given a training set $\{(y_i, \boldsymbol{x}_i)\}_{1\le i\le n}$ with $\boldsymbol{x}_i \in \mathbb{R}^p$ and $y_i \in \{1, -1\}$, one aims to fit a logistic regression model by solving the following program:

$$\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \quad \frac{1}{n}\sum_{i=1}^{n} \ell\big(y_i \boldsymbol{x}_i^\top \boldsymbol{\theta}^t\big). \tag{40}$$

Here, $\ell(u) \triangleq \log(1 + e^{-u})$ denotes the logistic loss. Further assume that the data is separable, i.e., there exists $\boldsymbol{\theta}^* \in \mathbb{R}^p$ such that $y_i \boldsymbol{\theta}^{*\top} \boldsymbol{x}_i > 0$ for all i. Under this condition, the loss function (40) can be arbitrarily close to zero for certain $\boldsymbol{\theta}$ with $\|\boldsymbol{\theta}\|_2 \to \infty$. What happens when we minimize (40) using gradient descent? [119] uncovers a striking phenomenon.

Theorem 9 (Theorem 3 in [119]). *Consider the logistic regression (40) with separable data. If we run GD*

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta \frac{1}{n}\sum_{i=1}^{n} y_i \boldsymbol{x}_i \ell'\big(y_i \boldsymbol{x}_i^\top \boldsymbol{\theta}^t\big)$$

from any initialization $\boldsymbol{\theta}^0$ with appropriate step size $\eta > 0$, then normalized $\boldsymbol{\theta}^t$ converges to a solution with the maximum ℓ_2 margin. That is,

$$\lim_{t\to\infty} \frac{\boldsymbol{\theta}^t}{\|\boldsymbol{\theta}^t\|_2} = \hat{\boldsymbol{\theta}}, \tag{41}$$

where $\hat{\boldsymbol{\theta}}$ is the solution to the hard margin support vector machine:

$$\hat{\boldsymbol{\theta}} \triangleq \arg\min_{\boldsymbol{\theta}\in\mathbb{R}^p} \|\boldsymbol{\theta}\|_2, \qquad \textit{subject to} \quad y_i \boldsymbol{x}_i^\top \boldsymbol{\theta} \ge 1 \quad \textit{for all } 1 \le i \le n. \tag{42}$$

The above theorem reveals that gradient descent, when solving logistic regression with separable data, implicitly regularizes the iterates towards the ℓ_2 max margin vector (cf. (41)), without any explicit regularization as in (42). Similar results have been obtained by [62]. In addition, [47] studied algorithms other than gradient descent and showed that coordinate descent produces a solution with the maximum ℓ_1 margin.

Moving beyond logistic regression, which can be viewed as a one-layer neural net, the theoretical understanding of implicit regularization in deeper neural networks is still limited; see [48] for an illustration in deep linear convolutional neural networks.

8 Discussion

Due to space limitations, we have omitted several important deep learning models; notable examples include deep reinforcement learning [86], deep probabilistic graphical models [109], variational autoencoders [66], transfer learning [133], etc. Apart from the modeling aspect, interesting theories on generative adversarial networks [10, 11], recurrent neural networks [3], connections with kernel methods [59, 9] are also emerging. We have also omitted the inverse-problem view of deep learning where the data are assumed to be generated from a certain neural net and the goal is to recover the weights in the NN with as few examples as possible. Various algorithms (e.g., GD with spectral initialization) have been shown to recover the weights successfully in some simplified settings [136, 118, 42, 87, 23, 39].

In the end, we identify a few important directions for future research.

- *New characterization of data distributions.* The success of deep learning relies on its power of efficiently representing complex functions relevant to real data. Comparatively, classical methods often have optimal guarantee if a problem has a certain known structure, such as smoothness, sparsity, and low-rankness [121, 31, 20, 24], but they are insufficient for complex data such as images. How to characterize the high-dimensional real data that can free us from known barriers, such as the curse of dimensionality is an interesting open question?

- *Understanding various computational algorithms for deep learning.* As we have emphasized throughout this survey, computational algorithms (e.g., variants of SGD) play a vital role in the success of deep learning. They allow fast training of deep neural nets and probably contribute towards the good generalization behavior of deep learning in practice. Understanding these computational algorithms and devising better ones are crucial components in understanding deep learning.
- *Robustness.* It has been well documented that DNNs are sensitive to small adversarial perturbations that are indistinguishable to humans [124]. This raises serious safety issues once if deploy deep learning models in applications such as self-driving cars, healthcare, etc. It is therefore crucial to refine current training practice to enhance robustness in a principled way [116].
- *Low SNRs.* Arguably, for image data and audio data where the signal-to-noise ratio (SNR) is high, deep learning has achieved great success. In many other statistical problems, the SNR may be very low. For example, in financial applications, the firm characteristic and covariates may only explain a small part of the financial returns; in healthcare systems, the uncertainty of an illness may not be predicted well from a patient's medical history. How to adapt deep learning models to excel at such tasks is an interesting direction to pursue?

Acknowledgements

J. Fan is supported in part by the NSF grants DMS-1712591 and DMS-1662139, the NIH grant R01-GM072611 and the ONR grant N00014-19-1-2120. We thank Ruying Bao, Yuxin Chen, Chenxi Liu, Weijie Su, Qingcan Wang and Pengkun Yang for helpful comments and discussions.

References

[1] Martín Abadi and et. al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Reza Abbasi-Asl, Yuansi Chen, Adam Bloniarz, Michael Oliver, Ben DB Willmore, Jack L Gallant, and Bin Yu. The deeptune framework for modeling and characterizing neurons in visual cortex area v4. *bioRxiv*, page 465534, 2018.

[3] Zeyuan Allen-Zhu and Yuanzhi Li. Can SGD Learn Recurrent Neural Networks with Provable Generalization? *ArXiv e-prints*, abs/1902.01028, 2019.

[4] Zeyuan Allen-Zhu, Yuanzhi Li, and Zhao Song. A convergence theory for deep learning via over-parameterization. *arXiv preprint arXiv:1811.03962*, 2018.

[5] Martin Anthony and Peter L Bartlett. *Neural network learning: Theoretical foundations*. cambridge university press, 2009.

[6] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. 70:214–223, 06–11 Aug 2017.

[7] Vladimir I Arnold. On functions of three variables. *Collected Works: Representations of Functions, Celestial Mechanics and KAM Theory, 1957–1965*, pages 5–8, 2009.

[8] Sanjeev Arora and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[9] Sanjeev Arora, Simon S Du, Wei Hu, Zhiyuan Li, and Ruosong Wang. Fine-grained analysis of optimization and generalization for overparameterized two-layer neural networks. *arXiv preprint arXiv:1901.08584*, 2019.

[10] Sanjeev Arora, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. Generalization and equilibrium in generative adversarial nets (GANs). In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 224–232. JMLR. org, 2017.

[11] Yu Bai, Tengyu Ma, and Andrej Risteski. Approximability of discriminators implies diversity in GANs. *arXiv preprint arXiv:1806.10586*, 2018.

[12] Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3):930–945, 1993.

[13] Peter L Bartlett. The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE transactions on Information Theory*, 44(2):525–536, 1998.

[14] Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6240–6249. Curran Associates, Inc., 2017.

[15] Benedikt Bauer and Michael Kohler. On deep learning as a remedy for the curse of dimensionality in nonparametric regression. Technical report, Technical report, 2017.

[16] Léon Bottou. Online learning and stochastic approximations. *On-line learning in neural networks*, 17(9):142, 1998.

[17] Olivier Bousquet and André Elisseeff. Stability and generalization. *Journal of machine learning research*, 2(Mar):499–526, 2002.

[18] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[19] Leo Breiman et al. Heuristics of instability and stabilization in model selection. *The annals of statistics*, 24(6):2350–2383, 1996.

[20] Emmanuel J Candès and Terence Tao. The power of convex relaxation: Near-optimal matrix completion. *arXiv preprint arXiv:0903.1476*, 2009.

[21] Chensi Cao, Feng Liu, Hai Tan, Deshou Song, Wenjie Shu, Weizhong Li, Yiming Zhou, Xiaochen Bo, and Zhi Xie. Deep learning and its applications in biomedicine. *Genomics, proteomics & bioinformatics*, 16(1):17–32, 2018.

[22] Tianqi Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. *arXiv preprint arXiv:1806.07366*, 2018.

[23] Yuxin Chen, Yuejie Chi, Jianqing Fan, and Cong Ma. Gradient descent with random initialization: Fast global convergence for nonconvex phase retrieval. *Mathematical Programming*, pages 1–33, 2019.

[24] Yuxin Chen, Yuejie Chi, Jianqing Fan, Cong Ma, and Yuling Yan. Noisy matrix completion: Understanding statistical guarantees for convex relaxation via nonconvex optimization. *arXiv preprint arXiv:1902.07698*, 2019.

[25] Lenaic Chizat and Francis Bach. On the global convergence of gradient descent for over-parameterized models using optimal transport. In *Advances in neural information processing systems*, pages 3040–3050, 2018.

[26] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[27] R Dennis Cook et al. Fisher lecture: Dimension reduction in regression. *Statistical Science*, 22(1):1–26, 2007.

[28] Jeffrey De Fauw, Joseph R Ledsam, Bernardino Romera-Paredes, Stanislav Nikolov, Nenad Tomasev, Sam Blackwell, Harry Askham, Xavier Glorot, Brendan O'Donoghue, Daniel Visentin, et al. Clinically applicable deep learning for diagnosis and referral in retinal disease. *Nature medicine*, 24(9):1342, 2018.

[29] Luc Devroye and Terry Wagner. Distribution-free performance bounds for potential function rules. *IEEE Transactions on Information Theory*, 25(5):601–604, 1979.

[30] David L Donoho. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS math challenges lecture*, 1(2000):32, 2000.

[31] David L Donoho and Jain M Johnstone. Ideal spatial adaptation by wavelet shrinkage. *biometrika*, 81(3):425–455, 1994.

[32] Simon S Du, Jason D Lee, Haochuan Li, Liwei Wang, and Xiyu Zhai. Gradient descent finds global minima of deep neural networks. *arXiv preprint arXiv:1811.03804*, 2018.

[33] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[34] Weinan E, Chao Ma, and Qingcan Wang. A priori estimates of the population risk for residual networks. *arXiv preprint arXiv:1903.02154*, 2019.

[35] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. In *Conference on Learning Theory*, pages 907–940, 2016.

[36] Jianqing Fan and Runze Li. Variable selection via nonconcave penalized likelihood and its oracle properties. *Journal of the American statistical Association*, 96(456):1348–1360, 2001.

[37] Vitaly Feldman and Jan Vondrak. High probability generalization bounds for uniformly stable algorithms with nearly optimal rate. *arXiv preprint arXiv:1902.10710*, 2019.

[38] Jerome H Friedman and Werner Stuetzle. Projection pursuit regression. *Journal of the American statistical Association*, 76(376):817–823, 1981.

[39] Haoyu Fu, Yuejie Chi, and Yingbin Liang. Local geometry of one-hidden-layer neural networks for logistic regression. *arXiv preprint arXiv:1802.06463*, 2018.

[40] Kunihiko Fukushima and Sei Miyake. Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition. In *Competition and cooperation in neural nets*, pages 267–285. Springer, 1982.

[41] Chao Gao, Jiyi Liu, Yuan Yao, and Weizhi Zhu. Robust estimation and generative adversarial nets. *arXiv preprint arXiv:1810.02030*, 2018.

[42] Surbhi Goel, Adam Klivans, and Raghu Meka. Learning one convolutional layer with overlapping patches. *arXiv preprint arXiv:1802.02547*, 2018.

[43] Noah Golowich, Alexander Rakhlin, and Ohad Shamir. Size-independent sample complexity of neural networks. *arXiv preprint arXiv:1712.06541*, 2017.

[44] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU Press, 4 edition, 2013.

[45] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[46] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[47] Suriya Gunasekar, Jason Lee, Daniel Soudry, and Nathan Srebro. Characterizing implicit bias in terms of optimization geometry. *arXiv preprint arXiv:1802.08246*, 2018.

[48] Suriya Gunasekar, Jason D Lee, Daniel Soudry, and Nati Srebro. Implicit bias of gradient descent on linear convolutional networks. In *Advances in Neural Information Processing Systems*, pages 9482–9491, 2018.

[49] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015.

[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European conference on computer vision*, pages 630–645. Springer, 2016.

[52] Geoffrey Hinton, Nitish Srivastava, and Kevin Swersky. Neural networks for machine learning lecture 6a overview of mini-batch gradient descent. 2012.

[53] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[54] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[55] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[56] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[57] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[58] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[59] Arthur Jacot, Franck Gabriel, and Clément Hongler. Neural tangent kernel: Convergence and generalization in neural networks. In *Advances in neural information processing systems*, pages 8580–8589, 2018.

[60] Prateek Jain, Sham M Kakade, Rahul Kidambi, Praneeth Netrapalli, and Aaron Sidford. Accelerating stochastic gradient descent. *arXiv preprint arXiv:1704.08227*, 2017.

[61] Adel Javanmard, Marco Mondelli, and Andrea Montanari. Analysis of a two-layer neural network via displacement convexity. *arXiv preprint arXiv:1901.01375*, 2019.

[62] Ziwei Ji and Matus Telgarsky. Risk and parameter convergence of logistic regression. *arXiv preprint arXiv:1803.07300*, 2018.

[63] Rahul Kidambi, Praneeth Netrapalli, Prateek Jain, and Sham Kakade. On the insufficiency of existing momentum schemes for stochastic optimization. In *2018 Information Theory and Applications Workshop (ITA)*, pages 1–9. IEEE, 2018.

[64] Jack Kiefer, Jacob Wolfowitz, et al. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[65] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[66] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[67] Jason M Klusowski and Andrew R Barron. Risk bounds for high-dimensional ridge function combinations including neural networks. *arXiv preprint arXiv:1607.01434*, 2016.

[68] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[69] Harold Kushner and G George Yin. *Stochastic approximation and recursive algorithms and applications*, volume 35. Springer Science & Business Media, 2003.

[70] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[71] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[72] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *Advances in Neural Information Processing Systems*, pages 6391–6401, 2018.

[73] Ker-Chau Li. Sliced inverse regression for dimension reduction. *Journal of the American Statistical Association*, 86(414):316–327, 1991.

[74] Xingguo Li, Junwei Lu, Zhaoran Wang, Jarvis Haupt, and Tuo Zhao. On tighter generalization bound for deep neural networks: Cnns, resnets, and beyond. *arXiv preprint arXiv:1806.05159*, 2018.

[75] Yujia Li, Kevin Swersky, and Rich Zemel. Generative moment matching networks. In *International Conference on Machine Learning*, pages 1718–1727, 2015.

[76] Tengyuan Liang. How well can generative adversarial networks (GAN) learn densities: A nonparametric view. *arXiv preprint arXiv:1712.08244*, 2017.

[77] Henry W Lin, Max Tegmark, and David Rolnick. Why does deep and cheap learning work so well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017.

[78] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[79] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, page 3, 2013.

[80] VE Maiorov and Ron Meir. On the near optimality of the stochastic approximation of smooth functions by neural networks. *Advances in Computational Mathematics*, 13(1):79–103, 2000.

[81] Yuly Makovoz. Random approximants and neural networks. *Journal of Approximation Theory*, 85(1):98–109, 1996.

[82] Song Mei, Theodor Misiakiewicz, and Andrea Montanari. Mean-field theory of two-layers neural networks: dimension-free bounds and kernel limit. *arXiv preprint arXiv:1902.06015*, 2019.

[83] Song Mei, Andrea Montanari, and Phan-Minh Nguyen. A mean field view of the landscape of two-layer neural networks. *Proceedings of the National Academy of Sciences*, 115(33):E7665–E7671, 2018.

[84] Hrushikesh Mhaskar, Qianli Liao, and Tomaso Poggio. Learning functions: when is deep better than shallow. *arXiv preprint arXiv:1603.00988*, 2016.

[85] Hrushikesh N Mhaskar. Neural networks for optimal approximation of smooth and analytic functions. *Neural computation*, 8(1):164–177, 1996.

[86] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[87] Marco Mondelli and Andrea Montanari. On the connection between learning two-layers neural networks and tensor decomposition. *arXiv preprint arXiv:1802.07301*, 2018.

[88] Yurii E Nesterov. A method for solving the convex programming problem with convergence rate o (1/kˆ 2). In *Dokl. Akad. Nauk SSSR*, volume 269, pages 543–547, 1983.

[89] Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. Norm-based capacity control in neural networks. In *Conference on Learning Theory*, pages 1376–1401, 2015.

[90] Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. In *Advances in Neural Information Processing Systems*, pages 271–279, 2016.

[91] Ian Parberry. *Circuit complexity and neural networks*. MIT press, 1994.

[92] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[93] Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

[94] Tomaso Poggio, Hrushikesh Mhaskar, Lorenzo Rosasco, Brando Miranda, and Qianli Liao. Why and when can deep-but not shallow-networks avoid the curse of dimensionality: a review. *International Journal of Automation and Computing*, 14(5):503–519, 2017.

[95] Boris T Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[96] Boris T Polyak and Anatoli B Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992.

[97] Boris Teodorovich Polyak and Yakov Zalmanovich Tsypkin. Adaptive estimation algorithms: convergence, optimality, stability. *Avtomatika i Telemekhanika*, (3):71–84, 1979.

[98] Christopher Poultney, Sumit Chopra, Yann LeCun, et al. Efficient learning of sparse representations with an energy-based model. In *Advances in neural information processing systems*, pages 1137–1144, 2007.

[99] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do cifar-10 classifiers generalize to cifar-10? *arXiv preprint arXiv:1806.00451*, 2018.

[100] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. 2018.

[101] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[102] William H Rogers and Terry J Wagner. A finite sample distribution-free performance bound for local discrimination rules. *The Annals of Statistics*, pages 506–514, 1978.

[103] David Rolnick and Max Tegmark. The power of deeper networks for expressing natural functions. *arXiv preprint arXiv:1705.05502*, 2017.

[104] Yaniv Romano, Matteo Sesia, and Emmanuel J Candès. Deep knockoffs. *arXiv preprint arXiv:1811.06687*, 2018.

[105] Grant M Rotskoff and Eric Vanden-Eijnden. Neural networks as interacting particle systems: Asymptotic convexity of the loss landscape and universal scaling of the approximation error. *arXiv preprint arXiv:1805.00915*, 2018.

[106] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[107] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[108] Haşim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*, 2014.

[109] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial intelligence and statistics*, pages 448–455, 2009.

[110] Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training GANs. In *Advances in Neural Information Processing Systems*, pages 2234–2242, 2016.

[111] Johannes Schmidt-Hieber. Nonparametric regression using deep neural networks with relu activation function. *arXiv preprint arXiv:1708.06633*, 2017.

[112] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[113] Shai Shalev-Shwartz, Ohad Shamir, Nathan Srebro, and Karthik Sridharan. Learnability, stability and uniform convergence. *Journal of Machine Learning Research*, 11(Oct):2635–2670, 2010.

[114] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[115] Bernard W Silverman. *Density estimation for statistics and data analysis*. Chapman & Hall, CRC, 1998.

[116] Chandan Singh, W James Murdoch, and Bin Yu. Hierarchical interpretations for neural network predictions. *arXiv preprint arXiv:1806.05337*, 2018.

[117] Justin Sirignano and Konstantinos Spiliopoulos. Mean field analysis of neural networks. *arXiv preprint arXiv:1805.01053*, 2018.

[118] Mahdi Soltanolkotabi. Learning relus via gradient descent. In *Advances in Neural Information Processing Systems*, pages 2007–2017, 2017.

[119] Daniel Soudry, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. The implicit bias of gradient descent on separable data. *The Journal of Machine Learning Research*, 19(1):2822–2878, 2018.

[120] David A Sprecher. On the structure of continuous functions of several variables. *Transactions of the American Mathematical Society*, 115:340–355, 1965.

[121] Charles J Stone. Optimal global rates of convergence for nonparametric regression. *The annals of statistics*, pages 1040–1053, 1982.

[122] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147, 2013.

[123] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[124] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[125] Matus Telgarsky. Benefits of depth in neural networks. *arXiv preprint arXiv:1602.04485*, 2016.

[126] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288, 1996.

[127] VN Vapnik and A Ya Chervonenkis. On the uniform convergence of relative frequencies of events to their probabilities. *Theory of Probability & Its Applications*, 16(2):264–280, 1971.

[128] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103. ACM, 2008.

[129] Stefan Wager, Sida Wang, and Percy S Liang. Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pages 351–359, 2013.

[130] E Weinan, Jiequn Han, and Arnulf Jentzen. Deep learning-based numerical methods for high-dimensional parabolic partial differential equations and backward stochastic differential equations. *Communications in Mathematics and Statistics*, 5(4):349–380, 2017.

[131] Ashia C Wilson, Rebecca Roelofs, Mitchell Stern, Nati Srebro, and Benjamin Recht. The marginal value of adaptive gradient methods in machine learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 4148–4158. Curran Associates, Inc., 2017.

[132] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[133] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In *Advances in neural information processing systems*, pages 3320–3328, 2014.

[134] Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.

[135] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

[136] Kai Zhong, Zhao Song, Prateek Jain, Peter L Bartlett, and Inderjit S Dhillon. Recovery guarantees for one-hidden-layer neural networks. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4140–4149. JMLR. org, 2017.

Top Deep Learning Interview Questions You Must Know

1.3K Views

Kurt
Last updated on May 22,2019

Deep Learning is one of the Hottest topics of 2018-19 and for a good reason. There have been so many advancements in the Industry wherein the time has come when machines or Computer Programs are actually replacing Humans. Artificial Intelligence is going to create 2.3 million Jobs by 2020 and to crack those job interview I have come up with a set of Deep Learning Interview Questions. I have divided this article into two sections:

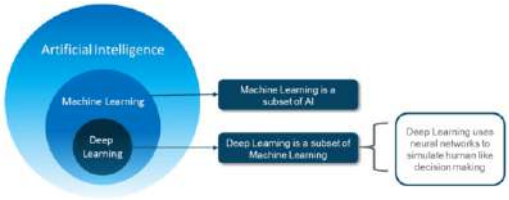
- Basic Deep Learning Interview Questions
- Advance Deep Learning Interview Questions

Basics Deep Learning Interview Questions

Q1. Differentiate between AI, Machine Learning and Deep Learning.

Artificial Intelligence is a technique which enables machines to mimic human behavior.

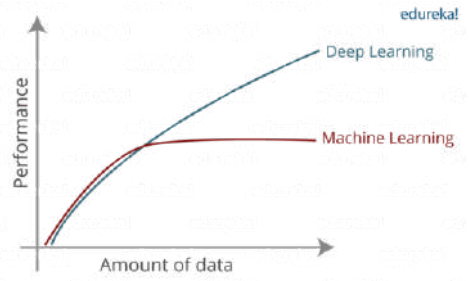
Machine Learning is a subset of AI technique which uses statistical methods to enable machines to improve with experience.



Deep learning is a subset of ML which make the computation of multi-layer neural network feasible. It uses Neural networks to simulate human-like decision making.

Q2. Do you think Deep Learning is Better than Machine Learning? If so, why?

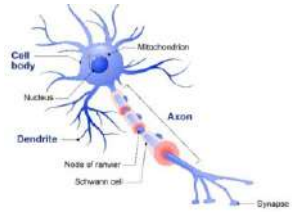
Though traditional ML algorithms solve a lot of our cases, they are not useful while working with high dimensional data, that is where we have a large number of inputs and outputs. For example, in the case of handwriting recognition, we have a large amount of input where we will have a different type of inputs associated with different type of handwriting.



The second major challenge is to tell the computer what are the features it should look for that will play an important role in predicting the outcome as well as to achieve better accuracy while doing so.

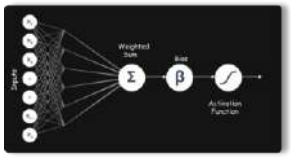
Q3. What is Perceptron? And How does it Work?

If we focus on the structure of a biological neuron, it has dendrites which are used to receive inputs. These inputs are summed in the cell body and using the Axon it is passed on to the next biological neuron as shown below.



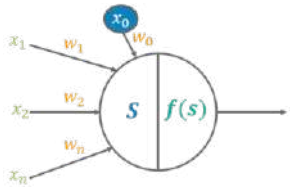
- **Dendrite:** Receives signals from other neurons
- **Cell Body:** Sums all the inputs
- **Axon:** It is used to transmit signals to the other cells

Similarly, a perceptron receives multiple inputs, applies various transformations and functions and provides an output. A Perceptron is a linear model used for binary classification. It models a neuron which has a set of inputs, each of which is given a specific weight. The neuron computes some function on these weighted inputs and gives the output.



Q4. What is the role of weights and bias?

For a perceptron, there can be one more input called **bias**. While the weights determine the **slope** of the classifier line, bias allows us to shift the line towards left or right. Normally bias is treated as another weighted input with the input value x_0.



Q5. What are the activation functions?

Activation function translates the inputs into outputs. Activation function decides whether a neuron should be activated or not by calculating the weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron.

There can be many Activation functions like:

- Linear or Identity
- Unit or Binary Step
- Sigmoid or Logistic
- Tanh
- ReLU
- Softmax

Q6. Explain Learning of a Perceptron.

1. Initializing the weights and threshold.
2. Provide the input and calculate the output.
3. Update the weights.
4. Repeat Steps 2 and 3

$$Wj\ (t+1) = Wj\ (t) + n\ (d-y)\ x$$

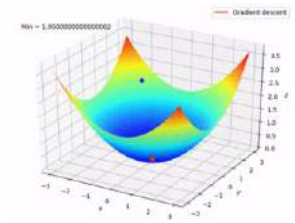
Wj (t+1) – Updated Weight
Wj (t) – Old Weight
d – Desired Output
y – Actual Output
x – Input

Q7. What is the significance of a Cost/Loss function?

A cost function is **a measure of the accuracy** of the neural network with respect to a given training sample and expected output. It provides the performance of a neural network as a whole. In deep learning, the goal is to minimize the cost function. For that, we use the concept of gradient descent.

Q8. What is gradient descent?

Gradient descent is an optimization algorithm used to minimize some function by iteratively moving in the direction of steepest descent as defined by the negative of the gradient.



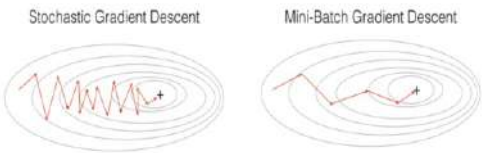
Stochastic Gradient Descent: Uses only a single training example to calculate the gradient and update parameters.

Batch Gradient Descent: Calculate the gradients for the whole dataset and perform just one update at each iteration.

Mini-batch Gradient Descent: Mini-batch gradient is a variation of stochastic gradient descent where instead of single training example, mini-batch of samples is used. It's one of the most popular optimization algorithms.

Q9. What are the benefits of mini-batch gradient descent?

- This is more efficient compared to stochastic gradient descent.
- The generalization by finding the flat minima.
- Mini-batches allows help to approximate the gradient of the entire training set which helps us to avoid local minima.



Q10.What are the steps for using a gradient descent algorithm?

- Initialize random weight and bias.
- Pass an input through the network and get values from the output layer.
- Calculate the error between the actual value and the predicted value.
- Go to each neuron which contributes to the error and then change its respective values to reduce the error.
- Reiterate until you find the best weights of the network.

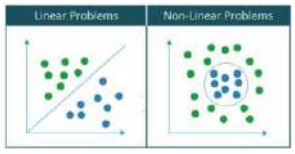
Q11. Create a Gradient Descent in python.

```
params = [weights_hidden, weights_output, bias_hidden, bias_output]

def sgd(cost, params, lr=0.05):

grads = T.grad(cost=cost, wrt=params)
updates = []

for p, g in zip(params, grads):
updates.append([p, p - g * lr])

return updates

updates = sgd(cost, params)
```

Q12. What are the shortcomings of a single layer perceptron?

Well, there are two major problems:

- Single-Layer Perceptrons cannot classify non-linearly separable data points.
- Complex problems, that involve a lot of parameters cannot be solved by Single-Layer Perceptrons



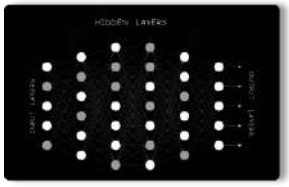
Q13. What is a Multi-Layer-Perceptron

A multilayer perceptron (MLP) is a deep, artificial neural network. It is composed of more than one perceptron. They are composed of an input layer to receive the signal, an output layer that makes a decision or prediction about the input, and in between those two, an arbitrary number of hidden layers that are the true computational engine of the MLP.

Q14. What are the different parts of a multi-layer perceptron?

Input Nodes: The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.

Hidden Nodes: The Hidden nodes perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a "Hidden Layer". While a network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.



Output Nodes: The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

Q15. What Is Data Normalization And Why Do We Need It?

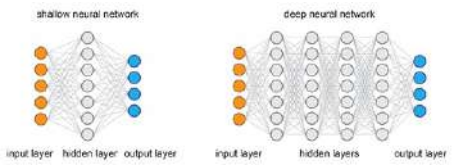
Data normalization is very important preprocessing step, used to rescale values to fit in a specific range to assure better convergence during backpropagation. In general, it boils down to subtracting the mean of each data point and dividing by its standard deviation.

These were some basic Deep Learning Interview Questions. Now, let's move on to some advanced ones.

Advance Interview Questions

Q16. Which is Better Deep Networks or Shallow ones? and Why?

Both the Networks, be it shallow or Deep are capable of approximating any function. But what matters is how precise that network is in terms of getting the results. A shallow network works with only a few features, as it can't extract more. But a deep network goes deep by computing efficiently and working on more features/parameters.



Q17. Why is Weight Initialization important in Neural Networks?

Weight initialization is one of the very important steps. A bad weight initialization can prevent a network from learning but good weight initialization helps in giving a quicker convergence and a better overall error.

Biases can be generally initialized to zero. The rule for setting the weights is to be close to zero without being too small.

Q18. What's the difference between a feed-forward and a backpropagation neural network?

A Feed-Forward Neural Network is a type of Neural Network architecture where the connections are "fed forward", i.e. do not form cycles. The term "Feed-Forward" is also used when you input something at the input layer and it travels from input to hidden and from hidden to the output layer.

Backpropagation is a training algorithm consisting of 2 steps:

- Feed-Forward the values.
- Calculate the error and propagate it back to the earlier layers.

So to be precise, forward-propagation is part of the backpropagation algorithm but comes before back-propagating.

Q19. What are the Hperparameteres? Name a few used in any Neural Network.

Hyperparameters are the variables which determine the network structure(Eg: Number of Hidden Units) and the variables which determine how the network is trained(Eg: Learning Rate). Hyperparameters are set before training.

- Number of Hidden Layers
- Network Weight Initialization
- Activation Function
- Learning Rate
- Momentum
- Number of Epochs
- Batch Size

Q20. Explain the different Hyperparameters related to Network and Training.

Network Hyperparameters

The number of Hidden Layers: Many hidden units within a layer with regularization techniques can increase accuracy. Smaller number of units may cause underfitting.

Network Weight Initialization: Ideally, it may be better to use different weight initialization schemes according to the activation function used on each layer. Mostly uniform distribution is used.

Activation function: Activation functions are used to introduce nonlinearity to models, which allows deep learning models to learn nonlinear prediction boundaries.

Training Hyperparameters

Learning Rate: The learning rate defines how quickly a network updates its parameters. Low learning rate slows down the learning process but converges smoothly. Larger learning rate speeds up the learning but may not converge.

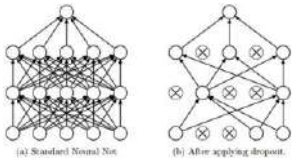
Momentum: Momentum helps to know the direction of the next step with the knowledge of the previous steps. It helps to prevent oscillations. A typical choice of momentum is between 0.5 to 0.9.

The number of epochs: Number of epochs is the number of times the whole training data is shown to the network while training. Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing(overfitting).

Batch size: Mini batch size is the number of sub-samples given to the network after which parameter update happens. A good default for batch size might be 32. Also try 32, 64, 128, 256, and so on.

Q21. What is Dropout?

Dropout is a regularization technique to avoid overfitting thus increasing the generalizing power. Generally, we should use a small dropout value of 20%-50% of neurons with 20% providing a good starting point. A probability too low has minimal effect and a value too high results in under-learning by the network.



Use a larger network. You are likely to get better performance when dropout is used on a larger network, giving the model more of an opportunity to learn independent representations.

Q22. In training a neural network, you notice that the loss does not decrease in the few starting epochs. What could be the reason?

The reasons for this could be:

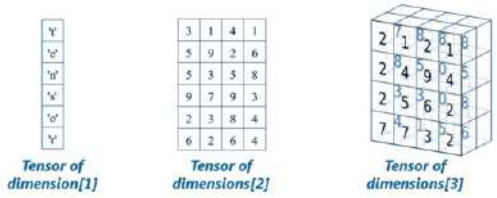
- The learning is rate is low
- Regularization parameter is high
- Stuck at local minima

Q23. Name a few deep learning frameworks

- TensorFlow
- Caffe
- The Microsoft Cognitive Toolkit/CNTK
- Torch/PyTorch
- MXNet
- Chainer
- Keras

Q24. What are Tensors?

Tensors are nothing but a de facto for representing the data in deep learning. They are just multidimensional arrays, that allows you to represent data having higher dimensions. In general, Deep Learning you deal with high dimensional data sets where dimensions refer to different features present in the data set.

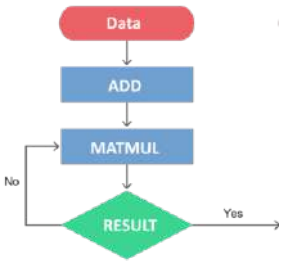


Q25. List a few advantages of TensorFlow?

- It has platform flexibility
- It is easily trainable on CPU as well as GPU for distributed computing.
- TensorFlow has auto differentiation capabilities
- It has advanced support for threads, asynchronous computation, and queue es.
- It is a customizable and open source.

Q26. What is Computational Graph?

A computational graph is a series of TensorFlow operations arranged as nodes in the graph. Each node takes zero or more tensors as input and produces a tensor as output.



Basically, one can think of a Computational Graph as an alternative way of conceptualizing mathematical calculations that takes place in a TensorFlow program. The operations assigned to different nodes of a Computational Graph can be performed in parallel, thus, providing better performance in terms of computations.

Q27. What is a CNN?

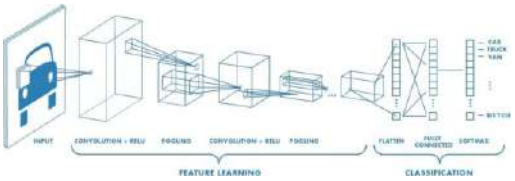
Convolutional neural network (CNN, or ConvNet) is a class of deep neural networks, most commonly applied to analyzing visual imagery. Unlike neural networks, where the input is a vector, here the input is a multi-channeled image. CNNs use a variation of multilayer perceptrons designed to require minimal preprocessing.

Q28. Explain the different Layers of CNN.

There are four layered concepts we should understand in Convolutional Neural Networks:

Convolution: The convolution layer comprises of a set of independent filters. All these filters are initialized randomly and become our parameters which will be learned by the network subsequently.

ReLu: This layer is used with the convolutional layer.



Pooling: Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network. Pooling layer operates on each feature map independently.

Full Connectedness: Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

Q29. What is an RNN?

Recurrent Networks are a type of artificial neural network designed to recognize patterns in sequences of data, such as text, genomes, handwriting, the spoken word, numerical times series data. Recurrent Neural Networks use **backpropagation** algorithm for training Because of their **internal memory**, RNN's are able to remember important things about the input they received, which enables them to be very precise in predicting what's coming next.

Q30. What are some issues faced while training an RNN?

Recurrent Neural Networks use backpropagation algorithm for training, but it is applied for every timestamp. It is commonly known as **Back-propagation Through Time** (BTT).

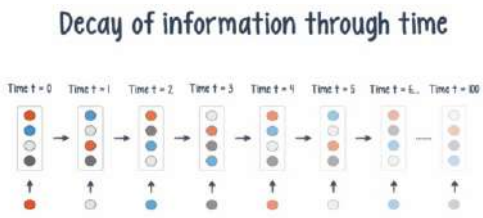
There are some issues with Back-propagation such as:

- Vanishing Gradient
- Exploding Gradient

Q31. What is Vanishing Gradient? And how is this harmful?

When we do Back-propagation, the gradients tend to get smaller and smaller as we keep on moving backward in the Network. This means that the neurons in the Earlier layers learn very slowly as compared to the neurons in the later layers in the Hierarchy.

Earlier layers in the Network are important because they are responsible to learn and detecting the simple patterns and are actually the building blocks of our Network.



Obviously, if they give improper and inaccurate results, then how can we expect the next layers and the complete Network to perform nicely and produce accurate results. The Training process takes too long and the Prediction Accuracy of the Model will decrease.

Q32. What is Exploding Gradient Descent?

Exploding gradients are a problem when large error gradients accumulate and result in very large updates to neural network model weights during training.

Gradient Descent process works best when these updates are small and controlled. When the magnitudes of the gradients accumulate, an unstable network is likely to occur, which can cause poor prediction of results or even a model that reports nothing useful what so ever.

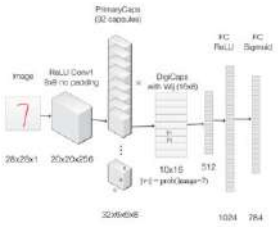
Q33. Explain the importance of LSTM.

Long short-term memory(LSTM) is an artificial recurrent neural network architecture used in the field of deep learning. Unlike standard feedforward neural networks, LSTM has feedback connections that make it a "general purpose computer". It can not only process single data points, but also entire sequences of data.

They are a special kind of Recurrent Neural Networks which are capable of learning long-term dependencies.

Q34. What are capsules in Capsule Neural Network?

Capsules are a vector specifying the features of the object and its likelihood. These features can be any of the instantiation parameters like position, size, orientation, deformation, velocity, hue, texture and much more.

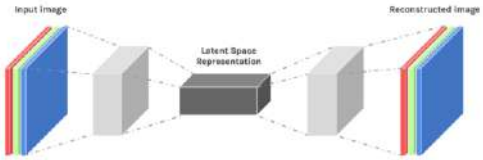


A capsule can also specify its attributes like angle and size so that it can represent the same generic information. Now, just like a neural network has layers of neurons, a capsule network can have layers of capsules.

Now, let's continue this Deep Learning Interview Questions and move to the section of autoencoders and RBMs.

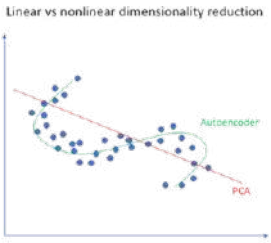
Q35. Explain Autoencoders and it's uses.

An autoencoder neural network is an Unsupervised Machine learning algorithm that applies backpropagation, setting the target values to be equal to the inputs. Autoencoders are used to reduce the size of our inputs into a smaller representation. If anyone needs the original data, they can reconstruct it from the compressed data.



Q36. In terms of Dimensionality Reduction, How does Autoencoder differ from PCAs?

- An autoencoder can learn non-linear transformations with a non-linear activation function and multiple layers.
- It doesn't have to learn dense layers. It can use convolutional layers to learn which is better for video, image and series data.
- It is more efficient to learn several layers with an autoencoder rather than learn one huge transformation with PCA.
- An autoencoder provides a representation of each layer as the output.
- It can make use of pre-trained layers from another model to apply transfer learning to enhance the encoder/decoder.



Q37. Give some real-life examples where autoencoders can be applied.

Image Coloring: Autoencoders are used for converting any black and white picture into a colored image. Depending on what is in the picture, it is possible to tell what the color should be.

Feature variation: It extracts only the required features of an image and generates the output by removing any noise or unnecessary interruption.

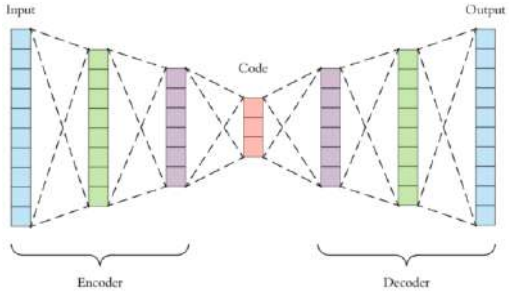
Dimensionality Reduction: The reconstructed image is the same as our input but with reduced dimensions. It helps in providing a similar image with a reduced pixel value.

Denoising Image: The input seen by the autoencoder is not the raw input but a stochastically corrupted version. A denoising autoencoder is thus trained to reconstruct the original input from the noisy version.

Q38. what are the different layers of Autoencoders?

An Autoencoder consist of three layers:

- Encoder
- Code
- Decoder



Q39. Explain the architecture of an Autoencoder.

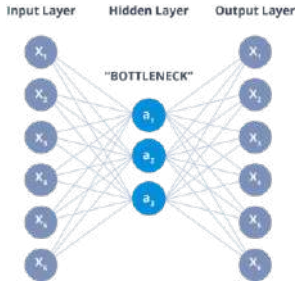
Encoder: This part of the network compresses the input into a latent space representation. The encoder layer encodes the input image as a compressed representation in a reduced dimension. The compressed image is the distorted version of the original image.

Code: This part of the network represents the compressed input which is fed to the decoder.

Decoder: This layer decodes the encoded image back to the original dimension. The decoded image is a lossy reconstruction of the original image and it is reconstructed from the latent space representation.

Q40. What is a Bottleneck in autoencoder and why is it used?

The layer between the encoder and decoder, ie. the code is also known as Bottleneck. This is a well-designed approach to decide which aspects of observed data are relevant information and what aspects can be discarded.



It does this by balancing two criteria:

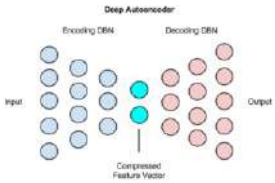
- Compactness of representation, measured as the compressibility.
- It retains some behaviourally relevant variables from the input.

Q41. Is there any variation of Autoencoders?

- Convolution Autoencoders
- Sparse Autoencoders
- Deep Autoencoders
- Contractive Autoencoders

Q42. What are Deep Autoencoders?

The extension of the simple Autoencoder is the Deep Autoencoder. The first layer of the Deep Autoencoder is used for first-order features in the raw input. The second layer is used for second-order features corresponding to patterns in the appearance of first-order features. Deeper layers of the Deep Autoencoder tend to learn even higher-order features.

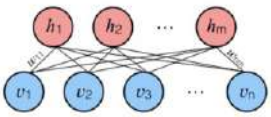


A deep autoencoder is composed of two, symmetrical deep-belief networks:

- First four or five shallow layers representing the encoding half of the net.
- The second set of four or five layers that make up the decoding half.

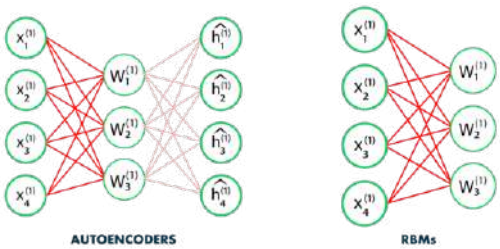
Q43. What is a Restricted Boltzmann Machine?

Restricted Boltzmann Machine is an undirected graphical model that plays a major role in Deep Learning Framework in recent times.
It is an algorithm which is useful for dimensionality reduction, classification, regression, collaborative filtering, feature learning, and topic modeling.



Q44. How Does RBM differ from Autoencoders?

Autoencoder is a simple 3-layer neural network where output units are directly connected back to input units. Typically, the number of hidden units is much less than the number of visible ones. The task of training is to minimize an error or reconstruction, i.e. find the most efficient compact representation for input data.



RBM shares a similar idea, but it uses stochastic units with particular distribution instead of deterministic distribution. The task of training is to find out how these two sets of variables are actually

Math of Deep Learning Neural Networks – Simplified

· Roopam Upadhyay

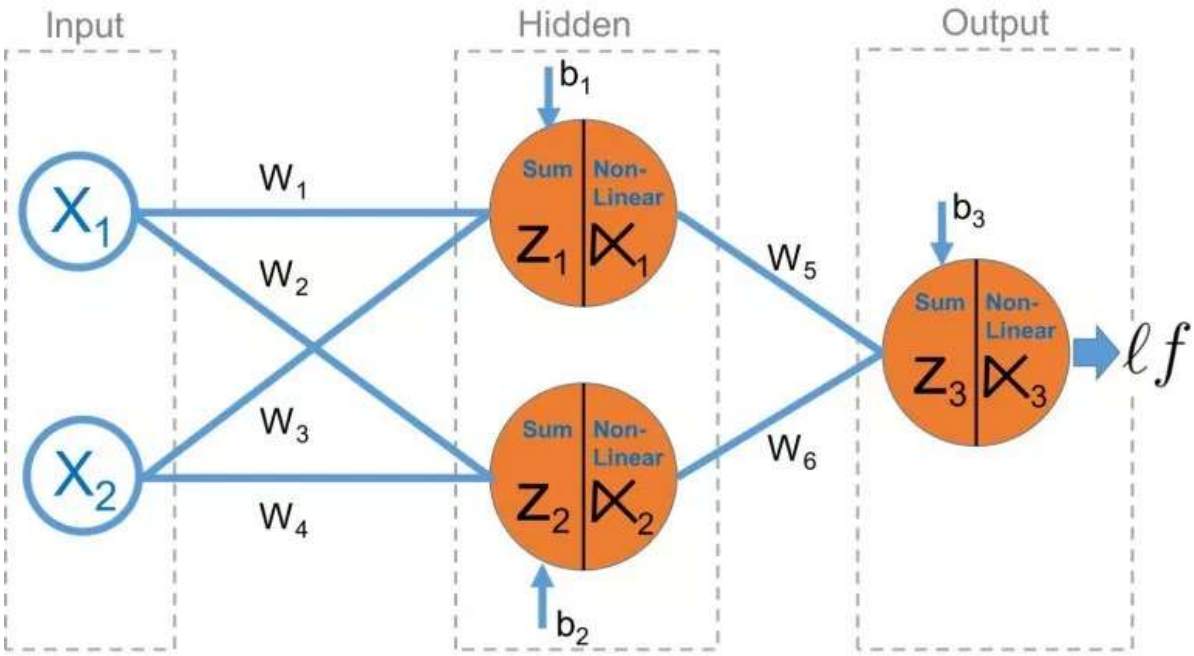
The Math of Deep Learning Neural Networks – by Roopam

to build an AI of your own. We will use the math of deep learning to make an image recognition AI in the next part. But before that let's create the links between…

The Math of Deep Learning and Plumbing

Last time we noticed that neural networks are like the networks of water pipes. The goal of neural networks is to identify the right settings for the knobs (6 in this schematic) to get the right output given the input.

Shown below is a familiar schematic of neural networks almost identical to the water pipelines above. The only exception is the additional bias terms (b_1,b_2, and b_3) added to the nodes.



In this post, we will solve this network to understand the math of deep learning. Note that a deep learning model has multiple hidden layers, unlike this simple neural network. However, this simple neural network can easily be generalized to the deep learning models. The math of deep learning does not change a lot with additional complexity and hidden layers. Here, our objective is to identify the values of the parameters {W (W_1,…, W_6) and b (b_1,b_2, and b_3)}. We will soon use the

backpropagation algorithm along with **gradient descent optimization** to solve this network and identify the optimal values of these weights.

Backpropagation and Gradient Descent

In the previous post, we discussed that the backpropagation algorithm works similar to me shouting back at my plumber while he was working in the duct. Remember, I was telling the plumber about the difference in actual water pressure from the expected. The plumber of neural networks, unlike my building's plumber, learns from this information to optimize the positions of the knobs. The method that the neural networks plumber uses to iteratively correct the weights or settings of the knobs is called gradient descent.

We have discussed the **gradient descent algorithm** in an earlier post to solve a logistic regression model. I recommend that you read that article to get a good grasp of the things we will discuss in this post. Essentially, the idea is to iteratively correct the value of the weights (W_i) to produce the least difference between the actual and the expected values of the output. This difference is measured mathematically by the loss function i.e ℓf. The weights (W_i and b_i) are then iteratively improved using the gradient of the loss function wrt weights using this expression:

$$W_i \leftarrow W_i + \alpha \cdot \frac{\partial \ell f}{\partial W_i}$$

Here, α is called the learning rate – it's a hyperparameter and stays constant. Hence, the overall problem boils down to the identification of partial derivatives of the loss function with respect to the weights i.e. $\frac{\partial \ell f}{\partial W_i}$. For our problem, we just need to solve the partial derivatives for W_5 and W_1. The partial derivatives for other weights can then be easily derived using the same method used for W_5 and W_1.

Before we solve these partial derivatives, let's do some more plumbing jobs and look at a tap to develop intuitions about the results we will get from the gradient descent optimization.

Intuitive Math of Deep Learning for W_5 & A Tap

We will use this simple tap to identify an optimal setting for its knob. In this process, we will develop intuitions about gradient descent and the math of deep learning. Here, the input is the water coming from the pipe on the left of the image. Moreover, the output is the water coming out of the tap. You use the knob, on the top of the tap, to regulate the quantity of the output water given the input. Remember, you want to turn the knob in such a way that you get the desired output (i.e the quantity of water) to wash your hands. Keep in mind, the position of the knob is similar to the weight of a neural networks' parameters. Moreover, the input/output water is similar to the input/output variables.

Essentially, in math terms, you are trying to identify how the position of the knob influences the output water. The mathematical equation for the same is:

$$\frac{\partial(Output\ \ water\ \ from\ \ the\ \ tap)}{\partial(Position/Setting\ \ of\ \ the\ \ knob)} = \frac{\partial \ell f}{\partial W_i}$$

If you understand the influence of the knob on the output flow of water you can easily turn it to get the desired output. Now, let's develop an intuition about how much to twist the knob. When you use a tap you twist the knob until you get the right flow or the output. When the difference between the desired output and the actual output is large then you need a lot of twisting. On the other hand, when the difference is less then you turn the knob gently.

Moreover, the other factor on which your decision depends on is the input from the left pipe. If there is no water flowing from the left pipe then no matter how much you twist the knob it won't help. Essentially, your action depends on these two factors.

```
Your decision to turn the knob depends on
Factor 1: Difference between the actual output and the desir
ed Output and
Factor 2: Input from the grey pipe on the left
```

Soon you will get the same result by doing a seemingly complicated math for the gradient descent to solve the neural network.

$$\frac{\partial \ell f}{\partial W_5} = (Output\ \ Difference) \cdot (Input)$$

For our network, the output difference is $(\propto_3 - y)$ and input is $\propto_1$. Hence,

$$\frac{\partial \ell f}{\partial W_5} = (\propto_3 - y) \cdot \propto_1$$

Disclaimer

Please note, to make the concepts easy for you to understand, I had taken a few liberties while defining the factors in the previous section. I will make these factors much more theoretically grounded at the end of this article when I will discuss the chain rule to solve derivatives. For now, I will continue to take more liberties in the next section when I discuss the other weight modification for other parameters of neural networks.

Add More Knobs to Solve W_1 – Intuitive Math of Deep Learning

Neural networks, as discussed earlier, have several parameters (Ws and bs). To develop an intuition about the math to estimate the other parameters further away from the output (i.e. W_1), let's add another knob to the tap.

Here, we have added a red regulator knob to the tap we saw in the earlier section. Now, the output from the tap is governed by both these knobs. Referring to the neural network's image shown earlier, the red knob is similar to the parameters (W_1, W_2, W_3, W_4, b_1, and b_2) added to the hidden layers. The knob on top of the brass tap is like the parameters to the output layer (i.e. W_5, W_6, and b_3).

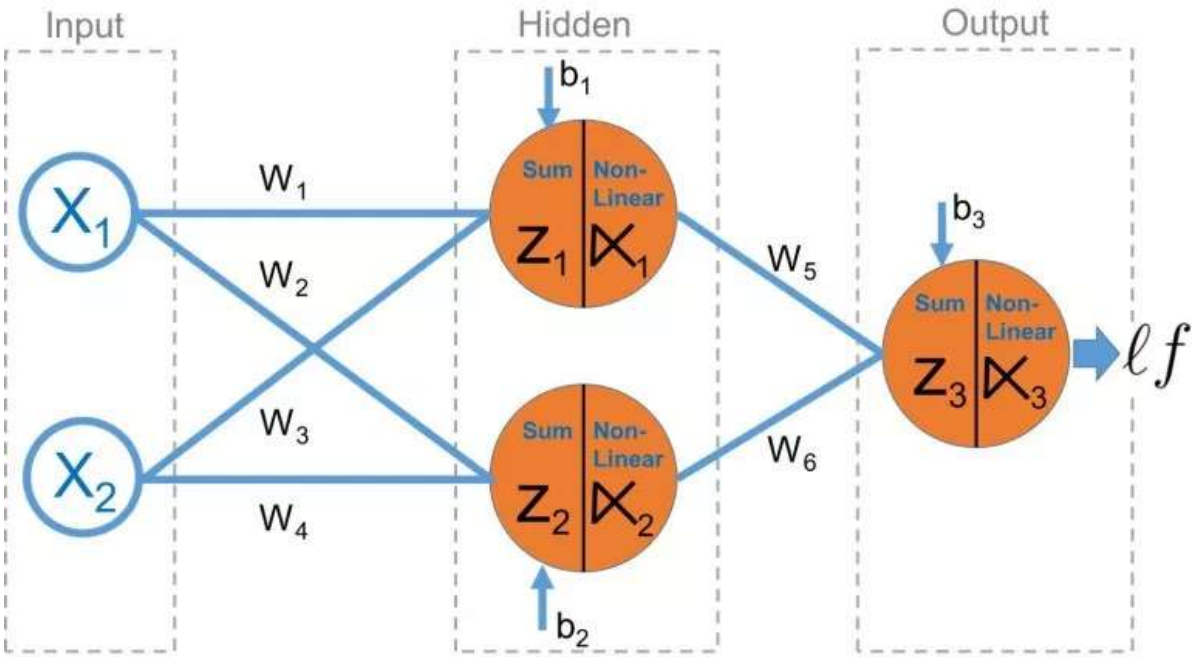
Now, you are also using the red knob, in addition to the knob on the tap, to get the desired output from the tap. Your effort of the red knob will depend on these factors.

```
Your decision to turn the red knob depends on
Factor 1: Difference between the actual and the desired fina
l output and
Factor 2: Position / setting of the knob on the brass tap an
d
Factor 3: Change in input to the brass tap caused by the red
knob and
Factor 4: Input from the pipe on the left into the red knob
```

Here, as already discussed earlier, factor 1 is $(\propto_3 - y)$. W_5 is the setting/weight for the knob of the brass tap. Factor 3 is $(\propto_1 - 1) \cdot \propto_1$. Finally, the last factor is the input or X_1. This completes our equation as:

$$\frac{\partial \ell f}{\partial W_1} = (\propto_3 - y) \cdot W_5 \cdot (\propto_1 - 1) \cdot \propto_1 \cdot X_1$$

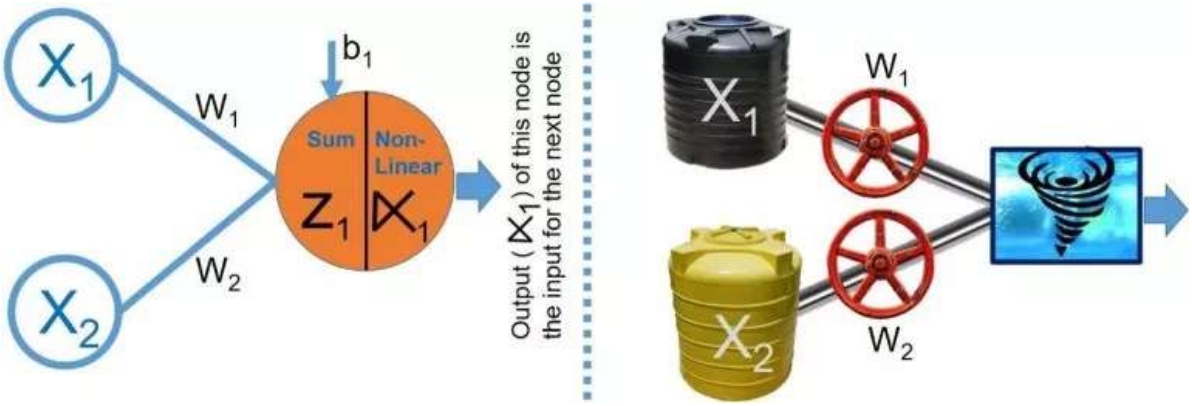
Now, before we do the math to get these results, we just need to discuss the components of our neural network in mathematical terms. We already know how it relates to the water pipelines discussed earlier.



Let's start with the nodes or the orange circles in the network diagram.

Nodes of Neural Networks

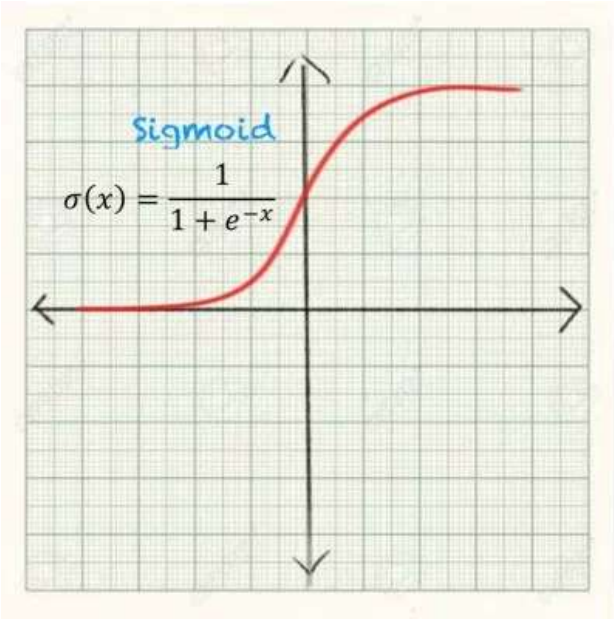
Here, these two networks are equivalent except the additional b_1 or bias for the neural networks.



The node for the neural network has two components i.e. sum and non-linear. The sum component (Z_1) is just a linear combination of the input and the weights.

$$Z_1 = W_1 \cdot X_1 + W_2 \cdot X_2 + b_1$$

The next term, i.e. non-linear, is the non-linear sigmoid activation function ($\aleph_1$). As discussed earlier, it is like a regulator of a fan that keeps the value of $\aleph_1$ between 0 and 1 or on/off.



The mathematical expression for this sigmoid activation function ($\ltimes_1$) is:

$$\ltimes_1 = \frac{e^{Z_1}}{1+e^{Z_1}}$$

The nodes in both the hidden and output layer behave the same as described above. Now, the last thing is to define the loss function (ℓf) which is to measure the difference between the expected and actual output. We will define the loss function for most common business problems.

Classification Problem – Math of Deep Learning

In practice, most business problems are about classification. They have binary or categorical outputs/answers such as:

- Is the last credit card transaction fraudulent or not?
- Will the borrower return the money or not?
- Was the last email in your mailbox a spam or ham?
- Is that a picture of a dog or cat? (this is not a business problem but a famous problem for deep learning)
- Is there an object in front of an autonomous car to generate a signal to push the break?
- Will the person surfing the web respond to the ad of a luxury condo?

Hence, we will design the loss function of our neural network for similar binary outputs. This binary loss function, aka binary cross entropy, can easily be extended for multiclass problems with minor modifications.

Loss Function and Cross Entropy

The loss function for binary output problems is:

$$\ell f = -(1-y) \cdot ln(1-\ltimes_3) - y \cdot ln(\ltimes_3)$$

This expression is also referred to as binary cross entropy. We can easily extend this binary cross-entropy to multi-class entropy if the output has many classes such as images of dog, cat, bus, car etc. We will learn about multiclass cross entropy and softmax function in the next part of this series. Now that we have identified all the components of the neural network, we are ready to solve it using the chain rule of differential equations.

Chain Rule for W_5 – Math of Deep Learning

We discussed the outcome for change observed in the loss function(ℓf) wrt to change in W_5 earlier using a single knob analogy. We know the answer to $\frac{\partial \ell f}{\partial W_5}$ is equal to $(\kappa_3 - y) \cdot \kappa_1$. Now, let's derive the same thing using the chain rule of derivatives. Essentially, this is similar to the change in water pressure observed at the output by turning the knob on the top of the tap. The chain rule states this:

$$\frac{\partial \ell f}{\partial W_5} = \frac{\partial \ell f}{\partial \kappa_3} \cdot \frac{\partial \kappa_3}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial W_5}$$

The above equation for chain rule is fairly simple since equation on the right-hand side will become the one on the left-hand side by simple division. More importantly, these equations suggest that the change in the output essentially the change observed at different components of the pipeline because of turning the knob.

Moreover, we already discussed the loss function which is the binary cross entropy i.e.

$$\ell f = -(1 - y) \cdot ln(1 - \kappa_3) - y \cdot ln(\kappa_3)$$

The first component of the chain rule is $\frac{\partial \ell f}{\partial \kappa_3}$ which is

$$\frac{\partial \ell f}{\partial \kappa_3} = \frac{(1-y)}{1-\kappa_3} - \frac{y}{\kappa_3} = \frac{(\kappa_3 - y)}{\kappa_3 \cdot (1-\kappa_3)}$$

This was fairly easy to compute if you only know that derivative of a natural log function is

$$\frac{\partial ln(x)}{\partial x} = \frac{1}{x}$$

This second component of the step function is $\frac{\partial \kappa_3}{\partial Z_3}$. This derivative of the sigmoid function (κ_3) is slightly more complicated. You could find here a detailed solution to **the derivative of the sigmoid function**. This implies,

$$\frac{\partial \kappa_3}{\partial Z_3} = \kappa_3 \cdot (1 - \kappa_3)$$

Finally, the third component of chain rule is again very easy to compute i.e.

$$\frac{\partial Z_3}{\partial W_5} = \kappa_1$$

Since we know,

$$Z_3 = W_5 \cdot \propto_1 + W_6 \cdot \propto_1 + b_3$$

Now, we just multiply these three components of the chain rule and we get the output i.e.

$$\frac{\partial \ell f}{\partial W_5} = \frac{(\propto_3 - y) \cdot (1 - \propto_3) \cdot \propto_3 \cdot \propto_1}{(1 - \propto_3) \cdot \propto_3} = (\propto_3 - y) \cdot \propto_1$$

Chain Rule for W_1 – Math of Deep Learning

The chain rule for the red knob or the additional layer is just an extension of the chain rule of the knob on the top of the tap. This one has a few more components because the water has to travel through more components i.e.

$$\frac{\partial \ell f}{\partial W_1} = \frac{\partial \ell f}{\partial \propto_3} \cdot \frac{\partial \propto_3}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial \propto_1} \cdot \frac{\partial \propto_1}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial W_1}$$

The first two components are exactly the same as the knob of the tap i.e. W_5. This makes sense since the water is flowing through the same pipeline towards the end. Hence, we will calculate the third component

$$\frac{\partial Z_3}{\partial \propto_1} = W_5$$

The fourth component is the derivative of the sigmoid function i.e. **the derivative of the sigmoid function**

$$\frac{\partial \propto_1}{\partial Z_1} = \propto_1 \cdot (1 - \propto_1)$$

The fifth and the final component is again easy to calculate.

$$\frac{\partial Z_1}{\partial W_1} = X_1$$

That's it. We now multiply these five components to get the results we have already seen for the additional red knob.

$$\frac{\partial \ell f}{\partial W_1} = \propto_3 \cdot (1 - \propto_3) \cdot W_5 \cdot \propto_1 \cdot (1 - \propto_1) \cdot X_1$$

Sign-off Node

This part of the series became a little math heavy. All this, however, will help us a lot when we will build an artificial intelligence to recognize images. See you then.

Super VIP Cheatsheet: Deep Learning

Afshine Amidi and Shervine Amidi

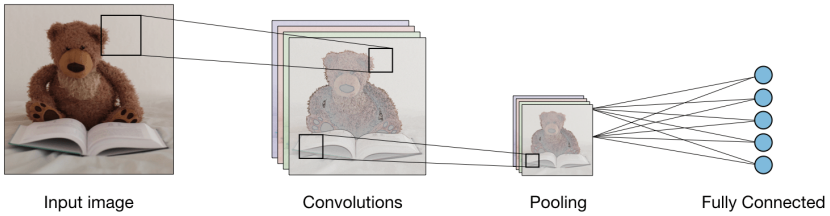
November 25, 2018

Contents

1 Convolutional Neural Networks

1.1 Overview

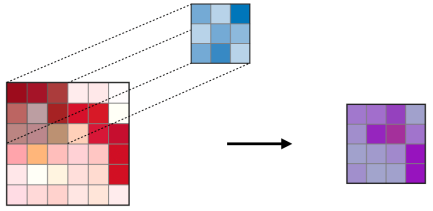
❒ **Architecture of a traditional CNN** – Convolutional neural networks, also known as CNNs, are a specific type of neural networks that are generally composed of the following layers:



The convolution layer and the pooling layer can be fine-tuned with respect to hyperparameters that are described in the next sections.

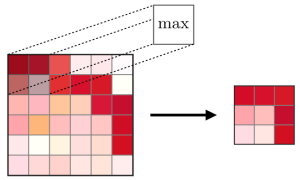
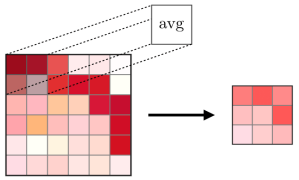
1.2 Types of layer

❒ **Convolutional layer (CONV)** – The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input I with respect to its dimensions. Its hyperparameters include the filter size F and stride S. The resulting output O is called *feature map* or *activation map*.

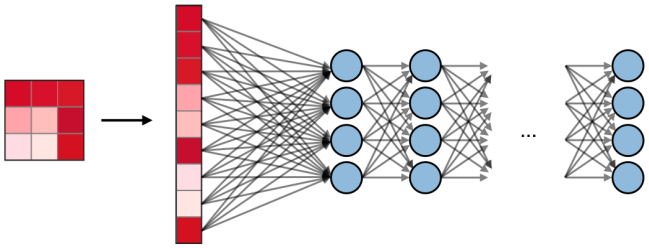


Remark: the convolution step can be generalized to the 1D and 3D cases as well.

❒ **Pooling (POOL)** – The pooling layer (POOL) is a downsampling operation, typically applied after a convolution layer, which does some spatial invariance. In particular, max and average pooling are special kinds of pooling where the maximum and average value is taken, respectively.

	Max pooling	Average pooling
Purpose	Each pooling operation selects the maximum value of the current view	Each pooling operation averages the values of the current view
Illustration		
Comments	- Preserves detected features - Most commonly used	- Downsamples feature map - Used in LeNet

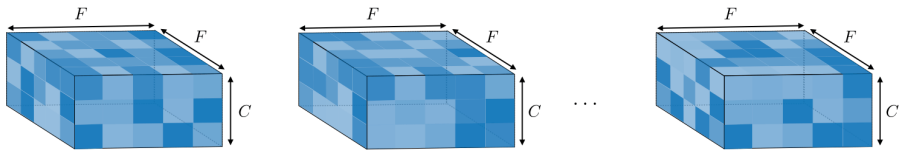
❒ **Fully Connected (FC)** – The fully connected layer (FC) operates on a flattened input where each input is connected to all neurons. If present, FC layers are usually found towards the end of CNN architectures and can be used to optimize objectives such as class scores.



1.3 Filter hyperparameters

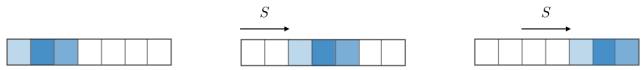
The convolution layer contains filters for which it is important to know the meaning behind its hyperparameters.

❒ **Dimensions of a filter** – A filter of size $F \times F$ applied to an input containing C channels is a $F \times F \times C$ volume that performs convolutions on an input of size $I \times I \times C$ and produces an output feature map (also called activation map) of size $O \times O \times 1$.

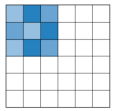
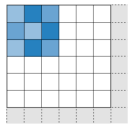
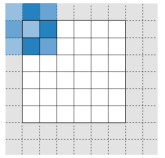


Remark: the application of K filters of size $F \times F$ results in an output feature map of size $O \times O \times K$.

❒ **Stride** – For a convolutional or a pooling operation, the stride S denotes the number of pixels by which the window moves after each operation.



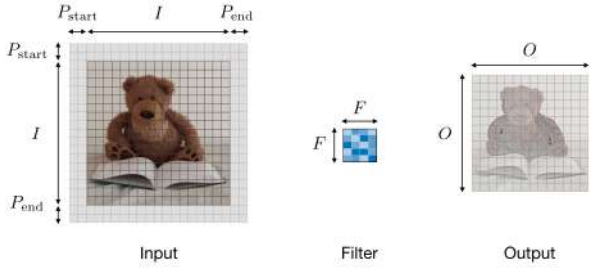
❒ **Zero-padding** – Zero-padding denotes the process of adding P zeroes to each side of the boundaries of the input. This value can either be manually specified or automatically set through one of the three modes detailed below:

	Valid	Same	Full
Value	$P = 0$	$P_{\text{start}} = \left\lfloor \frac{S\lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rfloor$ $P_{\text{end}} = \left\lceil \frac{S\lceil \frac{I}{S} \rceil - I + F - S}{2} \right\rceil$	$P_{\text{start}} \in [\![0, F-1]\!]$ $P_{\text{end}} = F - 1$
Illustration			
Purpose	- No padding - Drops last convolution if dimensions do not match	- Padding such that feature map size has size $\lceil \frac{I}{S} \rceil$ - Output size is mathematically convenient - Also called 'half' padding	- Maximum padding such that end convolutions are applied on the limits of the input - Filter 'sees' the input end-to-end

1.4 Tuning hyperparameters

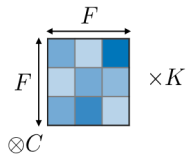
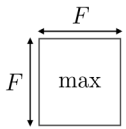
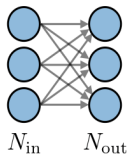
❒ **Parameter compatibility in convolution layer** – By noting I the length of the input volume size, F the length of the filter, P the amount of zero padding, S the stride, then the output size O of the feature map along that dimension is given by:

$$\boxed{O = \frac{I - F + P_{\text{start}} + P_{\text{end}}}{S} + 1}$$



Remark: often times, $P_{start} = P_{end} \triangleq P$, in which case we can replace $P_{start} + P_{end}$ by $2P$ in the formula above.

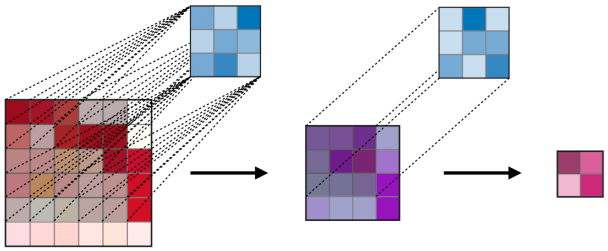
❒ **Understanding the complexity of the model** – In order to assess the complexity of a model, it is often useful to determine the number of parameters that its architecture will have. In a given layer of a convolutional neural network, it is done as follows:

	CONV	POOL	FC
Illustration			
Input size	$I \times I \times C$	$I \times I \times C$	N_{in}
Output size	$O \times O \times K$	$O \times O \times C$	N_{out}
Number of parameters	$(F \times F \times C + 1) \cdot K$	0	$(N_{\text{in}} + 1) \times N_{\text{out}}$
Remarks	- One bias parameter per filter - In most cases, $S < F$ - A common choice for K is $2C$	- Pooling operation done channel-wise - In most cases, $S = F$	- Input is flattened - One bias parameter per neuron - The number of FC neurons is free of structural constraints

❒ **Receptive field** – The receptive field at layer k is the area denoted $R_k \times R_k$ of the input that each pixel of the k-th activation map can 'see'. By calling F_j the filter size of layer j and S_i the stride value of layer i and with the convention $S_0 = 1$, the receptive field at layer k can be computed with the formula:

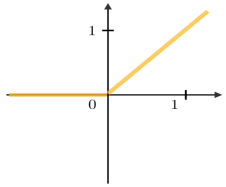
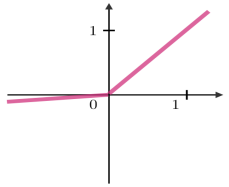
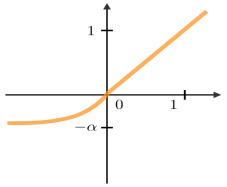
$$R_k = 1 + \sum_{j=1}^{k} (F_j - 1) \prod_{i=0}^{j-1} S_i$$

In the example below, we have $F_1 = F_2 = 3$ and $S_1 = S_2 = 1$, which gives $R_2 = 1+2 \cdot 1+2 \cdot 1 = 5$.



1.5 Commonly used activation functions

❒ **Rectified Linear Unit** – The rectified linear unit layer (ReLU) is an activation function g that is used on all elements of the volume. It aims at introducing non-linearities to the network. Its variants are summarized in the table below:

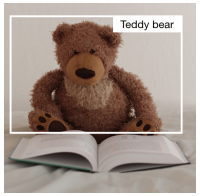
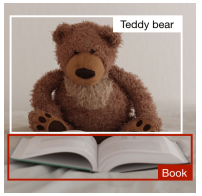
ReLU	Leaky ReLU	ELU
$g(z) = \max(0,z)$	$g(z) = \max(\epsilon z, z)$ with $\epsilon \ll 1$	$g(z) = \max(\alpha(e^z - 1), z)$ with $\alpha \ll 1$
		
Non-linearity complexities biologically interpretable	Addresses dying ReLU issue for negative values	Differentiable everywhere

❒ **Softmax** – The softmax step can be seen as a generalized logistic function that takes as input a vector of scores $x \in \mathbb{R}^n$ and outputs a vector of output probability $p \in \mathbb{R}^n$ through a softmax function at the end of the architecture. It is defined as follows:

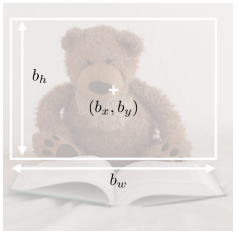
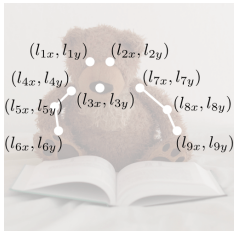
$$p = \begin{pmatrix} p_1 \\ \vdots \\ p_n \end{pmatrix} \quad \text{where} \quad p_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

1.6 Object detection

❒ **Types of models** – There are 3 main types of object recognition algorithms, for which the nature of what is predicted is different. They are described in the table below:

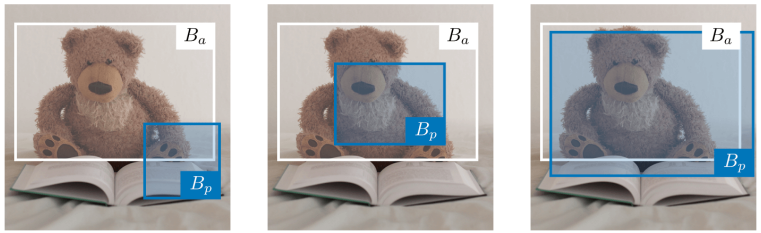
Image classification	Classification w. localization	Detection
		
- Classifies a picture - Predicts probability of object	- Detects object in a picture - Predicts probability of object and where it is located	- Detects up to several objects in a picture - Predicts probabilities of objects and where they are located
Traditional CNN	Simplified YOLO, R-CNN	YOLO, R-CNN

❒ **Detection** – In the context of object detection, different methods are used depending on whether we just want to locate the object or detect a more complex shape in the image. The two main ones are summed up in the table below:

Bounding box detection	Landmark detection
Detects the part of the image where the object is located	- Detects a shape or characteristics of an object (e.g. eyes) - More granular
	
Box of center (b_x,b_y), height b_h and width b_w	Reference points (l_{1x},l_{1y}), ...,(l_{nx},l_{ny})

❒ Intersection over Union – Intersection over Union, also known as IoU, is a function that quantifies how correctly positioned a predicted bounding box B_p is over the actual bounding box B_a. It is defined as:

$$\boxed{\text{IoU}(B_p,B_a) = \frac{B_p \cap B_a}{B_p \cup B_a}}$$



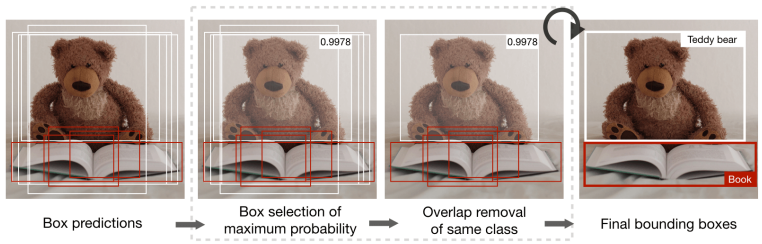
$\text{IoU}(B_p, B_a) = 0.1$ $\text{IoU}(B_p, B_a) = 0.5$ $\text{IoU}(B_p, B_a) = 0.9$

Remark: we always have $IoU \in [0,1]$. By convention, a predicted bounding box B_p is considered as being reasonably good if $IoU(B_p,B_a) \geqslant 0.5$.

❒ Anchor boxes – Anchor boxing is a technique used to predict overlapping bounding boxes. In practice, the network is allowed to predict more than one box simultaneously, where each box prediction is constrained to have a given set of geometrical properties. For instance, the first prediction can potentially be a rectangular box of a given form, while the second will be another rectangular box of a different geometrical form.

❒ Non-max suppression – The non-max suppression technique aims at removing duplicate overlapping bounding boxes of a same object by selecting the most representative ones. After having removed all boxes having a probability prediction lower than 0.6, the following steps are repeated while there are boxes remaining:

- Step 1: Pick the box with the largest prediction probability.
- Step 2: Discard any box having an IoU $\geqslant 0.5$ with the previous box.



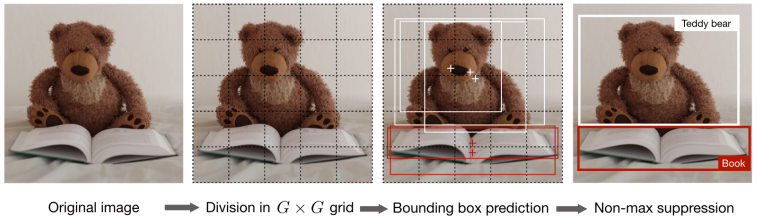
❒ YOLO – You Only Look Once (YOLO) is an object detection algorithm that performs the following steps:

- Step 1: Divide the input image into a $G \times G$ grid.
- Step 2: For each grid cell, run a CNN that predicts y of the following form:

$$\boxed{y = \Big[\underbrace{p_c,b_x,b_y,b_h,b_w,c_1,c_2,...,c_p}_{\text{repeated } k \text{ times}},...\Big]^T \in \mathbb{R}^{G\times G\times k\times(5+p)}}$$

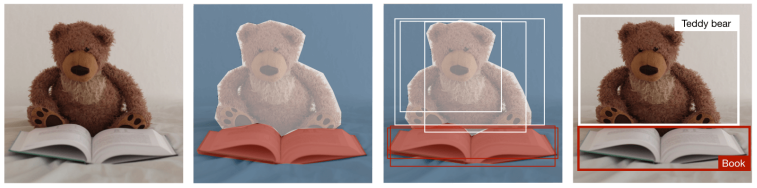
where p_c is the probability of detecting an object, b_x,b_y,b_h,b_w are the properties of the detected bouding box, c_1,...,c_p is a one-hot representation of which of the p classes were detected, and k is the number of anchor boxes.

- Step 3: Run the non-max suppression algorithm to remove any potential duplicate overlapping bounding boxes.



Remark: when $p_c = 0$, then the network does not detect any object. In that case, the corresponding predictions $b_x, ..., c_p$ have to be ignored.

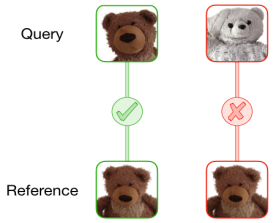
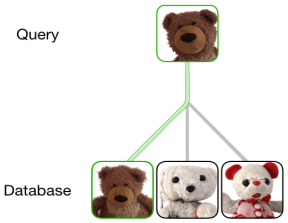
❒ R-CNN – Region with Convolutional Neural Networks (R-CNN) is an object detection algorithm that first segments the image to find potential relevant bounding boxes and then run the detection algorithm to find most probable objects in those bounding boxes.



Remark: although the original algorithm is computationally expensive and slow, newer architectures enabled the algorithm to run faster, such as Fast R-CNN and Faster R-CNN.

1.6.1 Face verification and recognition

❒ **Types of models** – Two main types of model are summed up in table below:

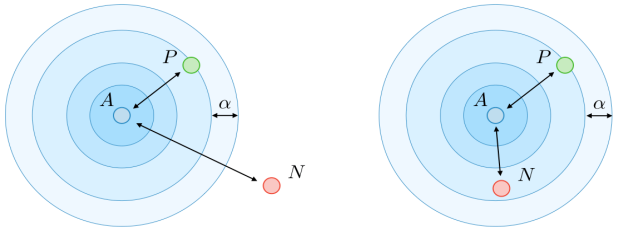
Face verification	Face recognition
- Is this the correct person? - One-to-one lookup	- Is this one of the K persons in the database? - One-to-many lookup
	

❒ **One Shot Learning** – One Shot Learning is a face verification algorithm that uses a limited training set to learn a similarity function that quantifies how different two given images are. The similarity function applied to two images is often noted $d(\text{image 1}, \text{image 2})$.

❒ **Siamese Network** – Siamese Networks aim at learning how to encode images to then quantify how different two images are. For a given input image $x^{(i)}$, the encoded output is often noted as $f(x^{(i)})$.

❒ **Triplet loss** – The triplet loss ℓ is a loss function computed on the embedding representation of a triplet of images A (anchor), P (positive) and N (negative). The anchor and the positive example belong to a same class, while the negative example to another one. By calling $\alpha \in \mathbb{R}^+$ the margin parameter, this loss is defined as follows:

$$\boxed{\ell(A,P,N) = \max\left(d(A,P) - d(A,N) + \alpha, 0\right)}$$



$\ell(A,P,N) = 0$ $\quad$ $\ell(A,P,N) > 0$

1.6.2 Neural style transfer

❒ **Motivation** – The goal of neural style transfer is to generate an image G based on a given content C and a given style S.

Content C $\quad$ Style S $\quad$ Generated image G

❒ **Activation** – In a given layer l, the activation is noted $a^{[l]}$ and is of dimensions $n_H \times n_w \times n_c$

❒ **Content cost function** – The content cost function $J_{\text{content}}(C,G)$ is used to determine how the generated image G differs from the original content image C. It is defined as follows:

$$\boxed{J_{\text{content}}(C,G) = \frac{1}{2}||a^{[l](C)} - a^{[l](G)}||^2}$$

❒ **Style matrix** – The style matrix $G^{[l]}$ of a given layer l is a Gram matrix where each of its elements $G^{[l]}_{kk'}$ quantifies how correlated the channels k and k' are. It is defined with respect to activations $a^{[l]}$ as follows:

$$\boxed{G^{[l]}_{kk'} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_w^{[l]}} a^{[l]}_{ijk} a^{[l]}_{ijk'}}$$

Remark: the style matrix for the style image and the generated image are noted $G^{[l](S)}$ and $G^{[l](G)}$ respectively.

❒ **Style cost function** – The style cost function $J_{\text{style}}(S,G)$ is used to determine how the generated image G differs from the style S. It is defined as follows:

$$\boxed{J^{[l]}_{\text{style}}(S,G) = \frac{1}{(2n_H n_w n_c)^2}||G^{[l](S)} - G^{[l](G)}||^2_F = \frac{1}{(2n_H n_w n_c)^2} \sum_{k,k'=1}^{n_c} \left(G^{[l](S)}_{kk'} - G^{[l](G)}_{kk'}\right)^2}$$

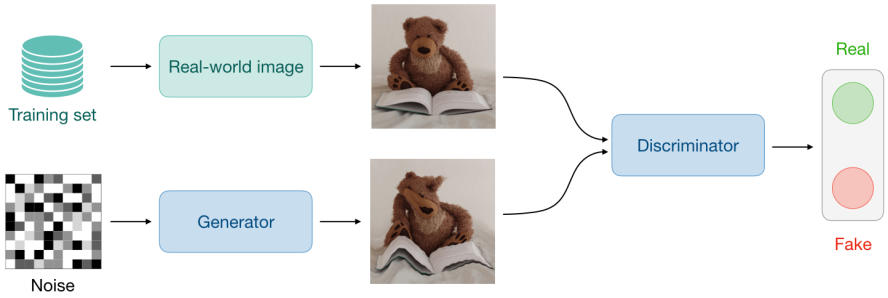
❒ **Overall cost function** – The overall cost function is defined as being a combination of the content and style cost functions, weighted by parameters α,β, as follows:

$$\boxed{J(G) = \alpha J_{\text{content}}(C,G) + \beta J_{\text{style}}(S,G)}$$

Remark: a higher value of α will make the model care more about the content while a higher value of β will make it care more about the style.

1.6.3 Architectures using computational tricks

❒ **Generative Adversarial Network** – Generative adversarial networks, also known as GANs, are composed of a generative and a discriminative model, where the generative model aims at generating the most truthful output that will be fed into the discriminative which aims at differentiating the generated and true image.



Remark: use cases using variants of GANs include text to image, music generation and synthesis.

❒ ResNet – The Residual Network architecture (also called ResNet) uses residual blocks with a high number of layers meant to decrease the training error. The residual block has the following characterizing equation:

$$\boxed{a^{[l+2]} = g(a^{[l]} + z^{[l+2]})}$$

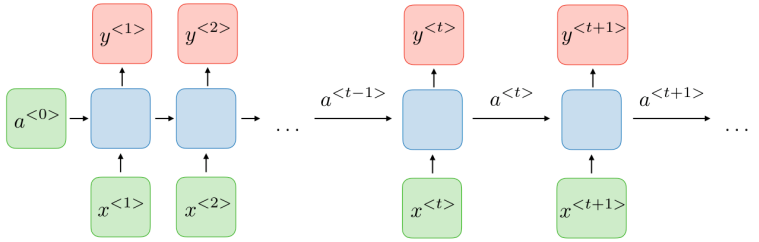
❒ Inception Network – This architecture uses inception modules and aims at giving a try at different convolutions in order to increase its performance. In particular, it uses the 1×1 convolution trick to lower the burden of computation.

⋆ ⋆ ⋆

2 Recurrent Neural Networks

2.1 Overview

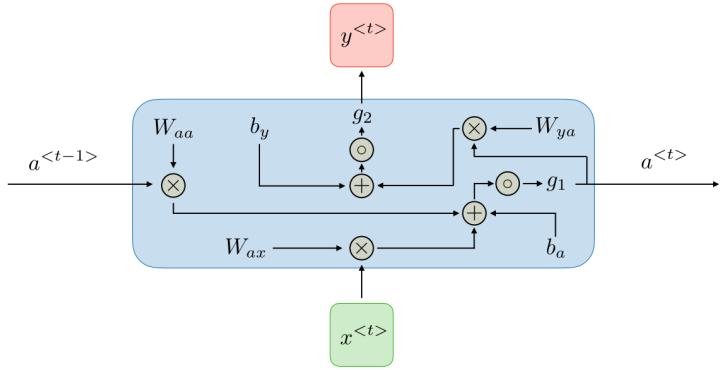
❒ Architecture of a traditional RNN – Recurrent neural networks, also known as RNNs, are a class of neural networks that allow previous outputs to be used as inputs while having hidden states. They are typically as follows:



For each timestep t, the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$\boxed{a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)} \quad \text{and} \quad \boxed{y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)}$$

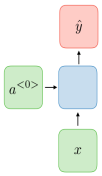
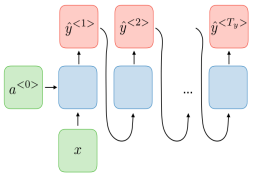
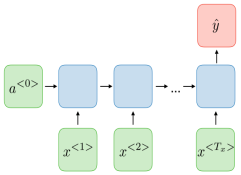
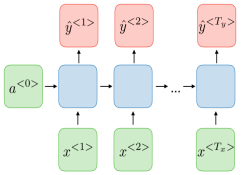
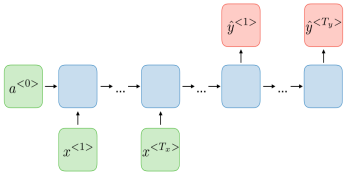
where $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$ are coefficients that are shared temporally and g_1, g_2 activation functions



The pros and cons of a typical RNN architecture are summed up in the table below:

Advantages	Drawbacks
- Possibility of processing input of any length - Model size not increasing with size of input - Computation takes into account historical information - Weights are shared across time	- Computation being slow - Difficulty of accessing information from a long time ago - Cannot consider any future input for the current state

❒ Applications of RNNs – RNN models are mostly used in the fields of natural language processing and speech recognition. The different applications are summed up in the table below:

Type of RNN	Illustration	Example
One-to-one $T_x = T_y = 1$		Traditional neural network
One-to-many $T_x = 1, T_y > 1$		Music generation
Many-to-one $T_x > 1, T_y = 1$		Sentiment classification
Many-to-many $T_x = T_y$		Name entity recognition
Many-to-many $T_x \neq T_y$		Machine translation

❒ **Loss function** – In the case of a recurrent neural network, the loss function $\mathcal{L}$ of all time steps is defined based on the loss at every time step as follows:

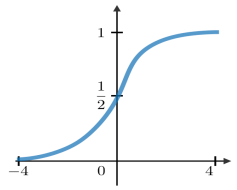
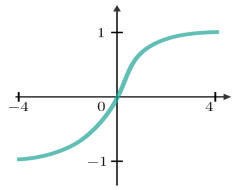
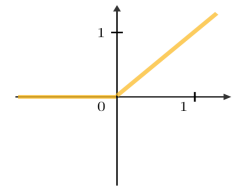
$$\boxed{\mathcal{L}(\widehat{y},y) = \sum_{t=1}^{T_y} \mathcal{L}(\widehat{y}^{<t>},y^{<t>})}$$

❒ **Backpropagation through time** – Backpropagation is done at each point in time. At timestep T, the derivative of the loss $\mathcal{L}$ with respect to weight matrix W is expressed as follows:

$$\boxed{\frac{\partial \mathcal{L}^{(T)}}{\partial W} = \sum_{t=1}^{T} \left.\frac{\partial \mathcal{L}^{(T)}}{\partial W}\right|_{(t)}}$$

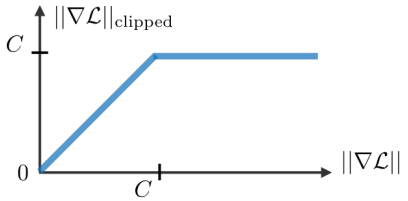
2.2 Handling long term dependencies

❒ **Commonly used activation functions** – The most common activation functions used in RNN modules are described below:

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0,z)$
		

❒ **Vanishing/exploding gradient** – The vanishing and exploding gradient phenomena are often encountered in the context of RNNs. The reason why they happen is that it is difficult to capture long term dependencies because of multiplicative gradient that can be exponentially decreasing/increasing with respect to the number of layers.

❒ **Gradient clipping** – It is a technique used to cope with the exploding gradient problem sometimes encountered when performing backpropagation. By capping the maximum value for the gradient, this phenomenon is controlled in practice.



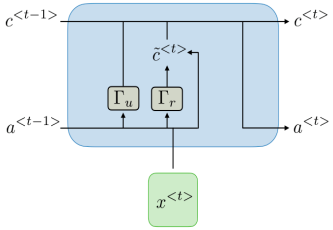
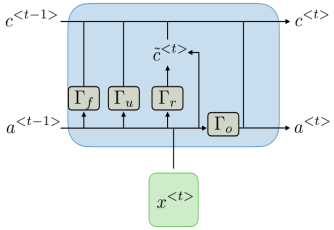
❒ **Types of gates** – In order to remedy the vanishing gradient problem, specific gates are used in some types of RNNs and usually have a well-defined purpose. They are usually noted Γ and are equal to:

$$\boxed{\Gamma = \sigma(Wx^{<t>} + Ua^{<t-1>} + b)}$$

where W, U, b are coefficients specific to the gate and σ is the sigmoid function. The main ones are summed up in the table below:

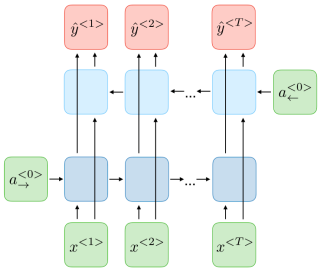
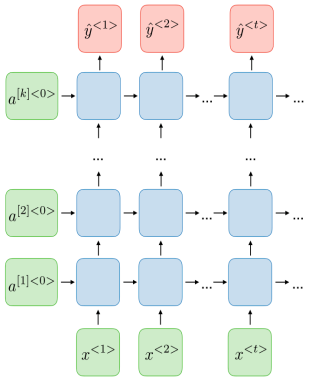
Type of gate	Role	Used in
Update gate Γ_u	How much past should matter now?	GRU, LSTM
Relevance gate Γ_r	Drop previous information?	GRU, LSTM
Forget gate Γ_f	Erase a cell or not?	LSTM
Output gate Γ_o	How much to reveal of a cell?	LSTM

❒ **GRU/LSTM** – Gated Recurrent Unit (GRU) and Long Short-Term Memory units (LSTM) deal with the vanishing gradient problem encountered by traditional RNNs, with LSTM being a generalization of GRU. Below is a table summing up the characterizing equations of each architecture:

	Gated Recurrent Unit (GRU)	**Long Short-Term Memory** (LSTM)
$\tilde{c}^{<t>}$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$	$\tanh(W_c[\Gamma_r \star a^{<t-1>}, x^{<t>}] + b_c)$
$c^{<t>}$	$\Gamma_u \star \tilde{c}^{<t>} + (1-\Gamma_u) \star c^{<t-1>}$	$\Gamma_u \star \tilde{c}^{<t>} + \Gamma_f \star c^{<t-1>}$
$a^{<t>}$	$c^{<t>}$	$\Gamma_o \star c^{<t>}$
Dependencies		

Remark: the sign $\star$ denotes the element-wise multiplication between two vectors.

❒ **Variants of RNNs** – The table below sums up the other commonly used RNN architectures:

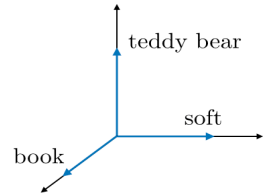
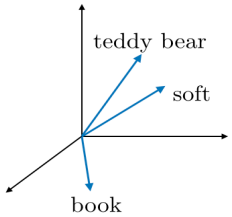
Bidirectional (BRNN)	**Deep** (DRNN)
	

2.3 Learning word representation

In this section, we note V the vocabulary and $|V|$ its size.

2.3.1 Motivation and notations

❒ **Representation techniques** – The two main ways of representing words are summed up in the table below:

1-hot representation	**Word embedding**
	
- Noted o_w - Naive approach, no similarity information	- Noted e_w - Takes into account words similarity

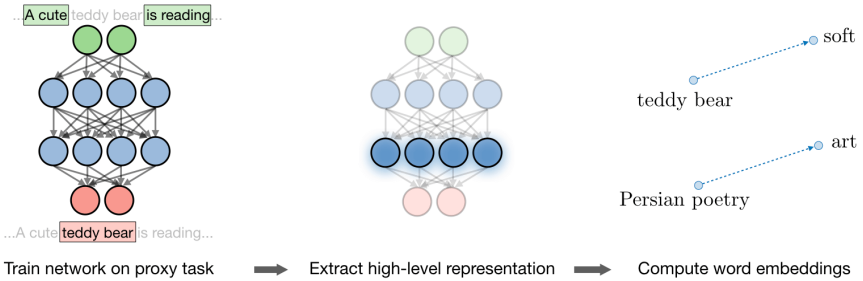
❒ **Embedding matrix** – For a given word w, the embedding matrix E is a matrix that maps its 1-hot representation o_w to its embedding e_w as follows:

$$\boxed{e_w = Eo_w}$$

Remark: learning the embedding matrix can be done using target/context likelihood models.

2.3.2 Word embeddings

❒ **Word2vec** – Word2vec is a framework aimed at learning word embeddings by estimating the likelihood that a given word is surrounded by other words. Popular models include skip-gram, negative sampling and CBOW.



❒ **Skip-gram** – The skip-gram word2vec model is a supervised learning task that learns word embeddings by assessing the likelihood of any given target word t happening with a context word c. By noting θ_t a parameter associated with t, the probability $P(t|c)$ is given by:

$$P(t|c) = \frac{\exp(\theta_t^T e_c)}{\sum_{j=1}^{|V|} \exp(\theta_j^T e_c)}$$

Remark: summing over the whole vocabulary in the denominator of the softmax part makes this model computationally expensive. CBOW is another word2vec model using the surrounding words to predict a given word.

❒ **Negative sampling** – It is a set of binary classifiers using logistic regressions that aim at assessing how a given context and a given target words are likely to appear simultaneously, with the models being trained on sets of k negative examples and 1 positive example. Given a context word c and a target word t, the prediction is expressed by:

$$P(y = 1|c,t) = \sigma(\theta_t^T e_c)$$

Remark: this method is less computationally expensive than the skip-gram model.

❒ **GloVe** – The GloVe model, short for global vectors for word representation, is a word embedding technique that uses a co-occurence matrix X where each $X_{i,j}$ denotes the number of times that a target i occurred with a context j. Its cost function J is as follows:

$$J(\theta) = \frac{1}{2} \sum_{i,j=1}^{|V|} f(X_{ij})(\theta_i^T e_j + b_i + b'_j - \log(X_{ij}))^2$$

here f is a weighting function such that $X_{i,j} = 0 \Longrightarrow f(X_{i,j}) = 0$.

Given the symmetry that e and θ play in this model, the final word embedding $e_w^{(\text{final})}$ is given by:

$$e_w^{(\text{final})} = \frac{e_w + \theta_w}{2}$$

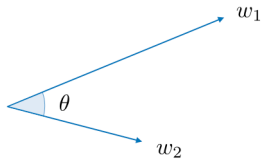
Remark: the individual components of the learned word embeddings are not necessarily interpretable.

2.4 Comparing words

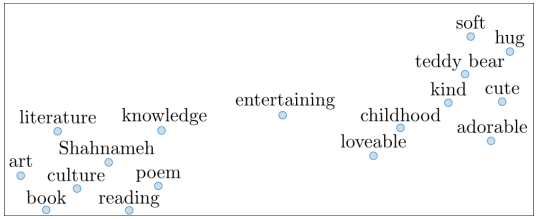
❒ **Cosine similarity** – The cosine similarity between words w_1 and w_2 is expressed as follows:

$$\text{similarity} = \frac{w_1 \cdot w_2}{||w_1||\ ||w_2||} = \cos(\theta)$$

Remark: θ is the angle between words w_1 and w_2.



❒ ***t*-SNE** – t-SNE (t-distributed Stochastic Neighbor Embedding) is a technique aimed at reducing high-dimensional embeddings into a lower dimensional space. In practice, it is commonly used to visualize word vectors in the 2D space.



2.5 Language model

❒ **Overview** – A language model aims at estimating the probability of a sentence $P(y)$.

❒ **n-gram model** – This model is a naive approach aiming at quantifying the probability that an expression appears in a corpus by counting its number of appearance in the training data.

❒ **Perplexity** – Language models are commonly assessed using the perplexity metric, also known as PP, which can be interpreted as the inverse probability of the dataset normalized by the number of words T. The perplexity is such that the lower, the better and is defined as follows:

$$\text{PP} = \prod_{t=1}^{T} \left(\frac{1}{\sum_{j=1}^{|V|} y_j^{(t)} \cdot \hat{y}_j^{(t)}} \right)^{\frac{1}{T}}$$

Remark: PP is commonly used in t-SNE.

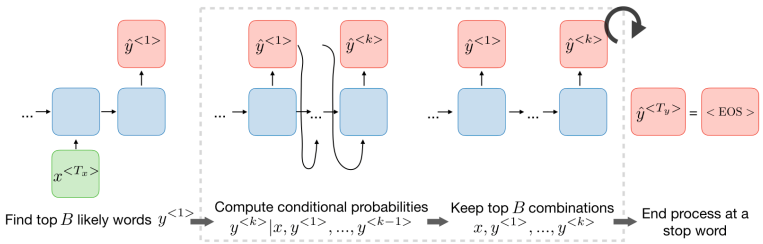
2.6 Machine translation

❒ **Overview** – A machine translation model is similar to a language model except it has an encoder network placed before. For this reason, it is sometimes referred as a conditional language model. The goal is to find a sentence y such that:

$$y = \underset{y^{<1>},\ldots,y^{<T_y>}}{\arg\max}\ P(y^{<1>},\ldots,y^{<T_y>}|x)$$

❒ **Beam search** – It is a heuristic search algorithm used in machine translation and speech recognition to find the likeliest sentence y given an input x.

- Step 1: Find top B likely words $y^{<1>}$
- Step 2: Compute conditional probabilities $y^{<k>}|x,y^{<1>},\ldots,y^{<k-1>}$
- Step 3: Keep top B combinations $x,y^{<1>},\ldots,y^{<k>}$



Remark: if the beam width is set to 1, then this is equivalent to a naive greedy search.

❒ **Beam width** – The beam width B is a parameter for beam search. Large values of B yield to better result but with slower performance and increased memory. Small values of B lead to worse results but is less computationally intensive. A standard value for B is around 10.

❒ **Length normalization** – In order to improve numerical stability, beam search is usually applied on the following normalized objective, often called the normalized log-likelihood objective, defined as:

$$\text{Objective } = \frac{1}{T_y^\alpha}\sum_{t=1}^{T_y}\log\left[p(y^{<t>}|x,y^{<1>},...,y^{<t-1>})\right]$$

Remark: the parameter α can be seen as a softener, and its value is usually between 0.5 and 1.

❒ **Error analysis** – When obtaining a predicted translation $\widehat{y}$ that is bad, one can wonder why we did not get a good translation y^* by performing the following error analysis:

Case	$P(y^*\|x) > P(\widehat{y}\|x)$	$P(y^*\|x) \leqslant P(\widehat{y}\|x)$
Root cause	Beam search faulty	RNN faulty
Remedies	Increase beam width	- Try different architecture - Regularize - Get more data

❒ **Bleu score** – The bilingual evaluation understudy (bleu) score quantifies how good a machine translation is by computing a similarity score based on n-gram precision. It is defined as follows:

$$\text{bleu score} = \exp\left(\frac{1}{n}\sum_{k=1}^{n}p_k\right)$$

where p_n is the bleu score on n-gram only defined as follows:

$$p_n = \frac{\sum_{\text{n-gram}\in\widehat{y}}\text{count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram}\in\widehat{y}}\text{count}(\text{n-gram})}$$

Remark: a brevity penalty may be applied to short predicted translations to prevent an artificially inflated bleu score.

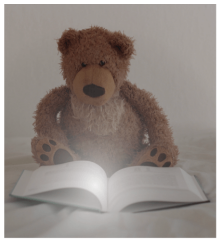
2.7 Attention

❒ **Attention model** – This model allows an RNN to pay attention to specific parts of the input that is considered as being important, which improves the performance of the resulting model in practice. By noting $\alpha^{<t,t'>}$ the amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ and $c^{<t>}$ the context at time t, we have:

$$c^{<t>} = \sum_{t'}\alpha^{<t,t'>}a^{<t'>} \quad \text{with} \quad \sum_{t'}\alpha^{<t,t'>} = 1$$

Remark: the attention scores are commonly used in image captioning and machine translation.

A cute teddy bear is reading Persian literature



A cute teddy bear is reading Persian literature

❒ **Attention weight** – The amount of attention that the output $y^{<t>}$ should pay to the activation $a^{<t'>}$ is given by $\alpha^{<t,t'>}$ computed as follows:

$$\alpha^{<t,t'>} = \frac{\exp(e^{<t,t'>})}{\sum_{t''=1}^{T_x}\exp(e^{<t,t''>})}$$

Remark: computation complexity is quadratic with respect to T_x.

⋆ ⋆ ⋆

3 Deep Learning Tips and Tricks

3.1 Data processing

❒ **Data augmentation** – Deep learning models usually need a lot of data to be properly trained. It is often useful to get more data from the existing ones using data augmentation techniques. The main ones are summed up in the table below. More precisely, given the following input image, here are the techniques that we can apply:

Original	Flip	Rotation	Random crop
			
- Image without any modification	- Flipped with respect to an axis for which the meaning of the image is preserved	- Rotation with a slight angle - Simulates incorrect horizon calibration	- Random focus on one part of the image - Several random crops can be done in a row

Color shift	Noise addition	Information loss	Contrast change
			
- Nuances of RGB is slightly changed - Captures noise that can occur with light exposure	- Addition of noise - More tolerance to quality variation of inputs	- Parts of image ignored - Mimics potential loss of parts of image	- Luminosity changes - Controls difference in exposition due to time of day

❒ **Batch normalization** – It is a step of hyperparameter γ, β that normalizes the batch $\{x_i\}$. By noting μ_B, σ_B^2 the mean and variance of that we want to correct to the batch, it is done as follows:

$$\boxed{x_i \longleftarrow \gamma \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta}$$

It is usually done after a fully connected/convolutional layer and before a non-linearity layer and aims at allowing higher learning rates and reducing the strong dependence on initialization.

3.2 Training a neural network

3.2.1 Definitions

❒ **Epoch** – In the context of training a model, epoch is a term used to refer to one iteration where the model sees the whole training set to update its weights.

❒ **Mini-batch gradient descent** – During the training phase, updating weights is usually not based on the whole training set at once due to computation complexities or one data point due to noise issues. Instead, the update step is done on mini-batches, where the number of data points in a batch is a hyperparameter that we can tune.

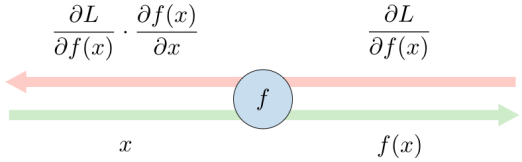
❒ **Loss function** – In order to quantify how a given model performs, the loss function L is usually used to evaluate to what extent the actual outputs y are correctly predicted by the model outputs z.

❒ **Cross-entropy loss** – In the context of binary classification in neural networks, the cross-entropy loss $L(z,y)$ is commonly used and is defined as follows:

$$\boxed{L(z,y) = -\Big[y\log(z) + (1-y)\log(1-z)\Big]}$$

3.2.2 Finding optimal weights

❒ **Backpropagation** – Backpropagation is a method to update the weights in the neural network by taking into account the actual output and the desired output. The derivative with respect to each weight w is computed using the chain rule.

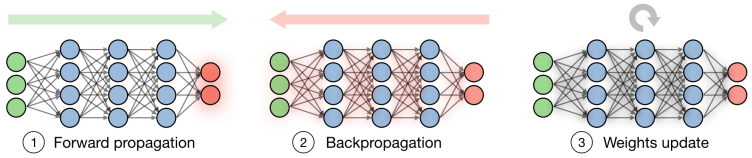


Using this method, each weight is updated with the rule:

$$\boxed{w \longleftarrow w - \alpha \frac{\partial L(z,y)}{\partial w}}$$

❒ **Updating weights** – In a neural network, weights are updated as follows:

- Step 1: Take a batch of training data and perform forward propagation to compute the loss.
- Step 2: Backpropagate the loss to get the gradient of the loss with respect to each weight.
- Step 3: Use the gradients to update the weights of the network.

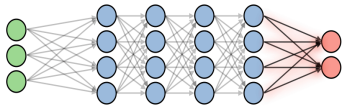
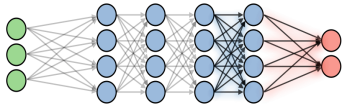
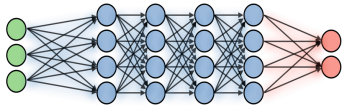


3.3 Parameter tuning

3.3.1 Weights initialization

❒ **Xavier initialization** – Instead of initializing the weights in a purely random manner, Xavier initialization enables to have initial weights that take into account characteristics that are unique to the architecture.

❒ **Transfer learning** – Training a deep learning model requires a lot of data and more importantly a lot of time. It is often useful to take advantage of pre-trained weights on huge datasets that took days/weeks to train, and leverage it towards our use case. Depending on how much data we have at hand, here are the different ways to leverage this:

Training size	Illustration	Explanation
Small		Freezes all layers, trains weights on softmax
Medium		Freezes most layers, trains weights on last layers and softmax
Large		Trains weights on layers and softmax by initializing weights on pre-trained ones

3.3.2 Optimizing convergence

❒ **Learning rate** – The learning rate, often noted α or sometimes η, indicates at which pace the weights get updated. It can be fixed or adaptively changed. The current most popular method is called Adam, which is a method that adapts the learning rate.

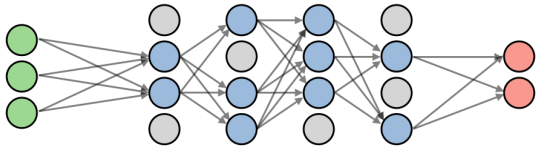
❒ **Adaptive learning rates** – Letting the learning rate vary when training a model can reduce the training time and improve the numerical optimal solution. While Adam optimizer is the most commonly used technique, others can also be useful. They are summed up in the table below:

Method	Explanation	Update of w	Update of b
Momentum	- Dampens oscillations - Improvement to SGD - 2 parameters to tune	$w - \alpha v_{dw}$	$b - \alpha v_{db}$
RMSprop	- Root Mean Square propagation - Speeds up learning algorithm by controlling oscillations	$w - \alpha \frac{dw}{\sqrt{s_{dw}}}$	$b \longleftarrow b - \alpha \frac{db}{\sqrt{s_{db}}}$
Adam	- Adaptive Moment estimation - Most popular method - 4 parameters to tune	$w - \alpha \frac{v_{dw}}{\sqrt{s_{dw}} + \epsilon}$	$b \longleftarrow b - \alpha \frac{v_{db}}{\sqrt{s_{db}} + \epsilon}$

Remark: other methods include Adadelta, Adagrad and SGD.

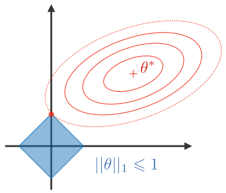
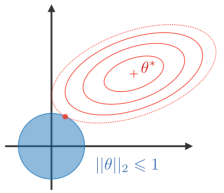
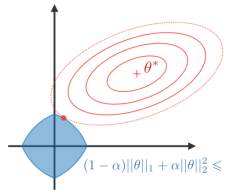
3.4 Regularization

❒ **Dropout** – Dropout is a technique used in neural networks to prevent overfitting the training data by dropping out neurons with probability $p > 0$. It forces the model to avoid relying too much on particular sets of features.

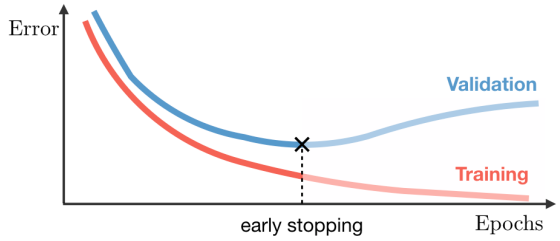


Remark: most deep learning frameworks parametrize dropout through the 'keep' parameter $1-p$.

❒ **Weight regularization** – In order to make sure that the weights are not too large and that the model is not overfitting the training set, regularization techniques are usually performed on the model weights. The main ones are summed up in the table below:

LASSO	Ridge	Elastic Net
- Shrinks coefficients to 0 - Good for variable selection	Makes coefficients smaller	Tradeoff between variable selection and small coefficients
		
$... + \lambda \|\|\theta\|\|_1$ $\lambda \in \mathbb{R}$	$... + \lambda \|\|\theta\|\|_2^2$ $\lambda \in \mathbb{R}$	$... + \lambda \left[(1-\alpha) \|\|\theta\|\|_1 + \alpha \|\|\theta\|\|_2^2 \right]$ $\lambda \in \mathbb{R}, \alpha \in [0,1]$

❒ **Early stopping** – This regularization technique stops the training process as soon as the validation loss reaches a plateau or starts to increase.



3.5 Good practices

❒ **Overfitting small batch** – When debugging a model, it is often useful to make quick tests to see if there is any major issue with the architecture of the model itself. In particular, in order to make sure that the model can be properly trained, a mini-batch is passed inside the network to see if it can overfit on it. If it cannot, it means that the model is either too complex or not complex enough to even overfit on a small batch, let alone a normal-sized training set.

❒ **Gradient checking** – Gradient checking is a method used during the implementation of the backward pass of a neural network. It compares the value of the analytical gradient to the numerical gradient at given points and plays the role of a sanity-check for correctness.

	Numerical gradient	**Analytical gradient**
Formula	$\frac{df}{dx}(x) \approx \frac{f(x+h)-f(x-h)}{2h}$	$\frac{df}{dx}(x) = f'(x)$
Comments	- Expensive; loss has to be computed two times per dimension - Used to verify correctness of analytical implementation -Trade-off in choosing h not too small (numerical instability) nor too large (poor gradient approx.)	- 'Exact' result - Direct computation - Used in the final implementation

⋆ ⋆ ⋆