← • previous table gives you a very good objective view of how much would be your "cieling" in terms of performance improvement, if you worked on each category. !

Incorrectly labeled Examples:

• When some of the ground-truth lables are wrong (the labeler did a mistake)

• DL Algorims are quite robust to random errors in the training set
↳ but not systematic errors.

• What about incorrectly labeled Examples in the dev set? or test set?

Do Error Analysis:

Image	Dog	Big Cat	Blurry	Incorrectly labled	Comment
1	✓				
2		✓	✓		
⋮				✓	labler missed the cat in Background
98		✓			
99				✓	Drawing of a cat, not a real cat.
100					
% of total	8%	43%	61%	6%	

• Now question is, Is it worth fixing the incorrect lables in the dev set?

• If it makes a significant difference to your ability to evaluate the Alg. using the dev set, then yes. Otherwise no

Example:

Overall dev-set error	10%	2%
Errors due to incorrect lables	0.6%	0.6%
Errors due to other causes	9.4%	1.4%
	not worth fixing	worth fixing

!! • Remember:
The purpose of dev set is to help you select between two classifiers A & B

Now if based on your dev set A has error 2.1%
B has error 1.9%

and you know 0.6% error in your dev set comes from incorrect lables, then your dev set is not here capable of reliably distinguish between A and B. So error must be fixed.

→

Notes on correcting incorrect dev/test set examples:

- Make sure you apply the fixing process to both dev and test sets. So that dev and test set must come from the same distribution.
- Consider both right (98%) as well as wrong (2%) examples (Positive & Negative)
- After fixing, the train and dev/test data may now come from slightly different distributions. But that's ok.
- Don't be reluctant in looking manually at your data. this gives you very good insights and intuitions on how your Algorithm works and what goes wrong.

Advise: Build your first System Quickly and then Iterate:

Speech Recognition Example:

- Noisy Background
  - Café noise
  - Car noise
- Accented speech
- Far from microphone
- young children's speech
- Stuttering, uh, um, ah, ..
  !

For any ML algorithm there might be 50 different reasonable directions to go. But which one has a better priority? Which one is more worthwhile?

- Build your First system Quickly
  - Define your dev/test set and evaluation metric (Define your target)
  - Build initial system quickly. (find training set and set up a system fast)
  - Use Bias/Variance and Error Analysis from this quickly built system to protrize next steps
  - Iterate

Mismatched training and dev/test set:

Training and testing on different distributions:

- Since DL Algorithms are very data hungry, many teams now a days train on data that is coming from different sources such as internet and may come from a different distribution from their test/dev dist.

Example: Cat app

200,000 Cat pic Samples from web — many data, but not from target dist.

10,000 Cat pic Sample from mobile cam — desired dist, but not much data

shuffle random shuffle

Option 1:

Random shuffle

train	dev	test
205,000	2500	2500

≈ 2381 web
≈ 119 mobile

→ Pro: • training, dev, test all come from the same distribution
• easy to manage

! **large disadvantage**: Majority of the dev/test set come from a distribution that is not the target dist. — Wrong target !

Option 2:

train		dev	test
web	app	App	App
205,000		2500	2500

- Pro: Aiming the target ! where it should be ≈ pictures uploaded from app
- Con: Training on a different dist than dev/test. But in the long run, that's ok.

Speech Recognition Example: Speech activated rearview mirror in Car

Training
- purchased
- smart speaker
- voice keyboard
- ...

500,000 utterances

dev / test

Speech activate rearview mirror

20,000 utterances

500k 10k 5k 5k !

Train	D	T
510,000		

Bias and Variance with mismatched data distribution:

- When the training and dev/set data come from different distributions, the way we use bias and variance to analyse the performance of the algorith changes.

Cat Classifier Example:

Assume humans get ≈ 0% error (proxy for Bayes error)

Training error	1%
Dev error	10%

- If dist of dev ≃ dist of train ⟹ Variance !
- But if dist of dev ≠ dist of train ⟹ what?? !

Maybe it is even doing very fine on dev set. We can't tell.

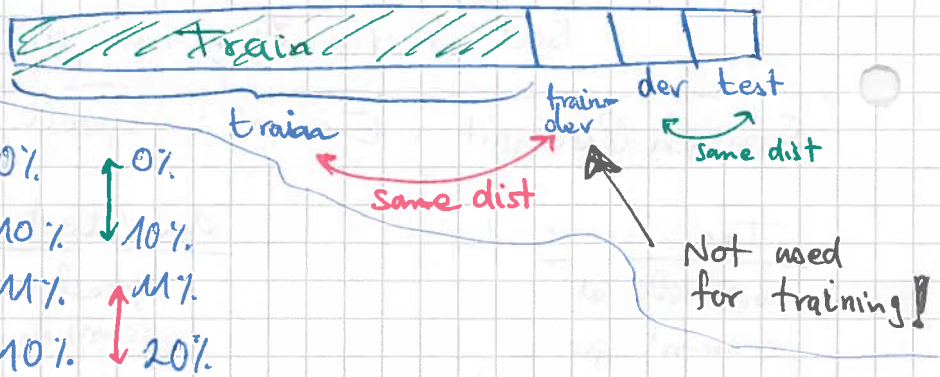
Problem: two things change at the same time.

- algorithm saw trained data but not dev data
- distribution of data is different in dev set

- We can't tell how much of the error comes from the fact that the algorithm did not generalize well to dev set, and how much of the error comes from the fact that dev and train data are different.

Solution: define a new set called "Training-dev Set" with same dist as training set but not used for training

Examples:



Human error	0%	0%	0%	0%
Train error	1%	1%	10%	10%
Train-dev error	9%	1.5%	11%	11%
dev error	10%	10%	10%	20%
	⇓ Variance Problem	⇓ Data mismatch problem	⇓ High Bias Problem	⇓ Two problems: • Bias • data mismatch

4%	Human-level error	→ Avoidable bias: how far you are from Bayes error
7%	Training set error	→ Variance: how well you generalize to dev set; how badly you are overfitting to train set
10%	Training-dev set error	→ data mismatch: how different is your dev set from your train set
12%	Dev error	→ Degree of over fitting to the dev set
12%	Test error	

Addressing Data mismatch:

- There are not systematic ways to address this problem :(But some good practices :)
- Carry out ~~data~~ manual error analysis to try to understand differences between training and dev/test sets
  - e.g. noisy, car noise
  - many street numbers are not recognised
- ⇒ try to make the training data more similar or collect more similar data for training
  - e.g. add simulated car noise
  - or record people saying street numbers.

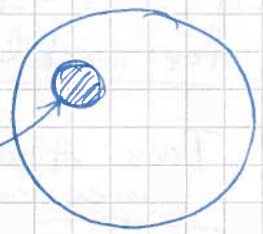
Artificial Data Synthesis:

Does work,
But be cautious of Overfitting

"The quick brown fox jumps over the lazy dog." + Car noise = Synthesized in-car audio

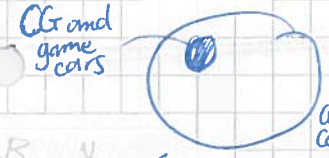
- Caution: imagine you have 10,000 hours of speech but only 1 hour of car noise

⇒ Your algorithm may very much overfit to this only 1 hour of car noise.



Overfitting is really a major concern! !

→ Another example:
Get cars from Computer graphics or games (Synthesized)



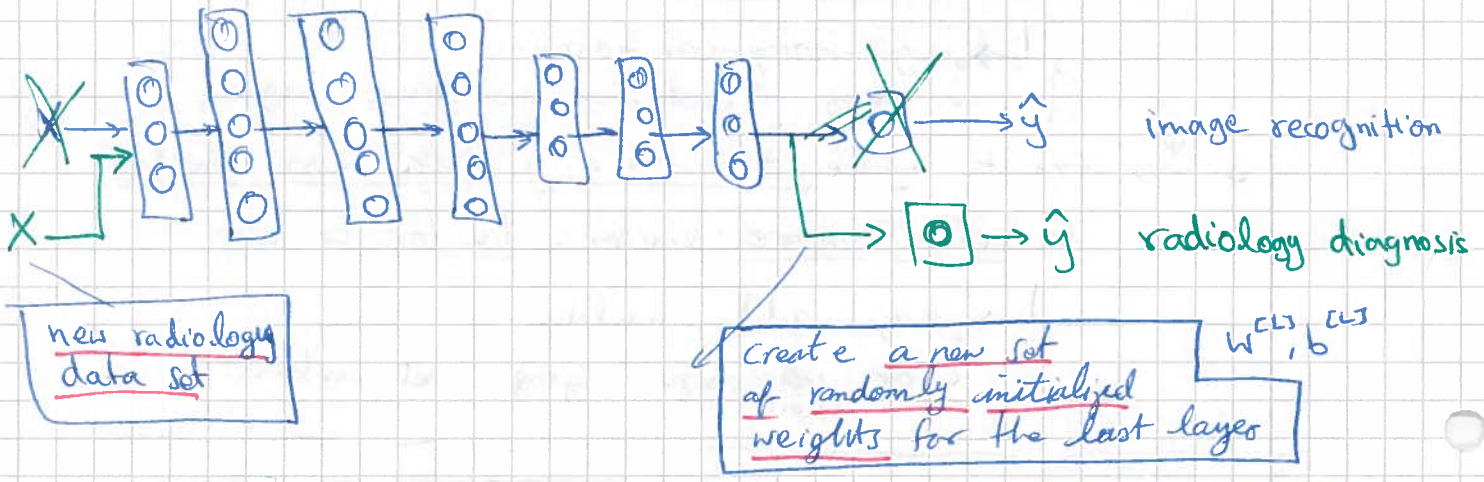
But there are many, many, many, different and divers cars in the real world compared to computer games

$1000000 $100000 $1000,200,20 ≈

Overfitting!

Transfer Learning: From problem A → to problem B

- Take knowledge that NN has learned from one task (A), and apply that knowledge to a separate task. (B)
- For example we use a NN trained for "cat recognition" to solve task for "recognizing radiology images"



- initial phase of training a NN is called "pre-training"
- after that training to the radiology task is called "fine tuning"
- If you have few data for the last dataset then only re-train the last layer. Otherwise re-train more layers
- you can add several new layers instead of the last layer.

When does Transfer Learning make sense?

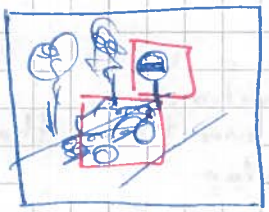
- you have a lot of data for problem A, but few data for problem B to which you want to transfer.
- Task A and B have the same input format (e.g. both take images, or both audio)
- ! low-level features from A could be helpful for learning B
  - edges, colors, ...
  - human speech, phonems, ...

Multitask Learning:

- In transfer learning you have a sequential process:
  You learn from task A and transfer that to task B

- In $\boxed{\text{multi task learning}}$, you start learning multiple tasks simultanously. And each of these tasks helps the others.

Example: Simplified Autonomous driving example:



$x^{(i)}$

Your NN should detect multiple different things in an image:

	$y^{(i)}$
Pedestrians	0
Cars	1
Stop signs	1
Trafic lights	0
⋮	

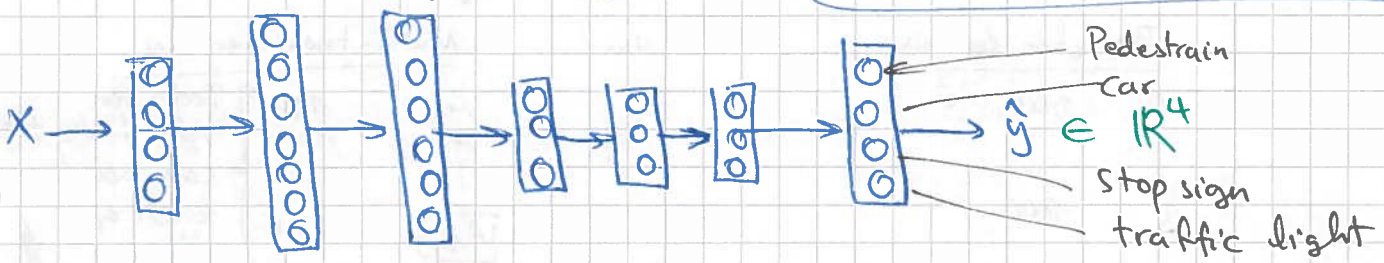
$(4,1)$

y is now 4 dimensional

$$Y = \begin{bmatrix} | & | & & | \\ y^{(1)} & y^{(2)} & \cdots & y^{(m)} \\ | & | & & | \end{bmatrix}$$

$Y : (4, m)$

Task: Train a NN to predict these values of y from an input image



Cost (cost?) $\hat{y}^{(i)}$ $(4,1)$:

$$\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{4} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

Summing over elements of y

usual logistic loss:

$$-y_j^{(i)} \log \hat{y}_j^{(i)} - (1 - y_j^{(i)}) \log (1 - \hat{y}_j^{(i)})$$

- Unlike softmax regression, where each image could have only 1 lable, now an image can have multiple lables.

- $\boxed{\text{Training an NN that can minimize this cost function is an example of Multi-task learning}}$ →

- A single NN looks at an image and solves 4 different problems at the same time

- An Alternative could be to train 4 different NNs, one for each task.

  ↳ But if there are fine-grained features that could be shared among these tasks, then training 1 NN to do 4 different things results in better performance!

- Multi task learning also works, when an input only has a subset of lables:

$$Y = \begin{bmatrix} 1 & 1 & & 0 & ? \\ 0 & 1 & & 1 & 1 \\ ? & ? & \cdots & 1 & ? \\ ? & ? & & 0 & ? \end{bmatrix}$$

↳ In Cost function, sum only over terms that have a value.

When does Multi-task learning make sense?

- Training on a set of tasks that could benefit from having shared lower-lered features.

- Amount of data for each task is quite similar:

Transfer learning		vs.	Multi-task Learning	
A	1,000,000		A_1	1,000
↓	↓		A_2	1,000
B	1000		⋮	⋮
			A_{100}	1,000

Data for All tasks together can boost each of them

training one task in isolation would not work well (only 1000 samples)

1,000,000

- You can train a big enough neural network to do well on all tasks. [The only time multi-task learning hurts performance compared to training separate NNs for each task, is when you cannot train a big enough NN.]

- Mult-task learning is in practice less used than transfer learning, because there are less tasks that require learning multiple things at the same time.

End-to-End learning:

- ML and Data processing systems used to require multiple processing stages.
- end-to-end learning discards those intermediate stages and trys to learn the task directly from input data.

Speech Recognition:

Traditional pipeline:

$x \xrightarrow{\text{MFCC}}$ features $\xrightarrow{\text{ML}}$ Phonems $\xrightarrow{\text{ML}}$ words $\longrightarrow y$

audio — transcript

end-to-end learning:

$x \longrightarrow y$

audio — transcript

:) Sociological effect: E2E learning makes the research of many people who worked on the intermediate stages of problems, obsolette :D.

:) $\Longrightarrow$ Many researchers find it hard to accept it.

- A challenge for E2E Learning is you need a lot of data.
  - ↳ e.g. 3000 hours of audio works well with the traditional pipeline but not with E2E.
  - But if you have 10,000 h or 100,000 h of audio then E2E learning beats the traditional pipeline.

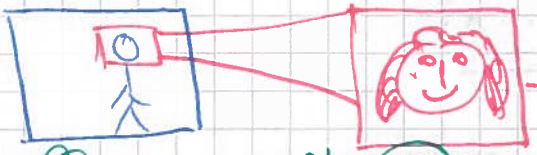
Face Recognition:

One way:

Image x $\longrightarrow$ y Identity

! But this does not work well.
(e.g. because of different positions and directions of the person in image)

Another way: A multi-step approach:



Step 1: Detect face in the Image

Step 2: Crop and zoom and center face and detect Id

Step 2: Actually takes two images and tells if the two images are the same person or not.

← Why is the second approach better? (two step)

- Each problem is simpler. (sub problems)
- [more important] You have a lot of data for each of the two subtasks (face detection, comparing faces and saying if the same person)

!

- In contrast you don't have a lot of data for direct, one step approach.

⟹ Sometimes breaking a problem into multiple steps gives better results compared to E2E learning.

More Examples:

- Machine translation:

(x, y) — English → French

Because there are milions of these pairs in Internet

(traditional) English → text analys → ~ → ~ → French

(E2E) English → French

→ Better

- Estimate child's age from X-Ray of hand (used for checking proper body growth :))

X-Ray

Better → Image → segment bones → look up table of bone sizes → age

Imag → age

Because there are a lot of data for bone segmentation from X-ray. But not enough (Imag, age) pairs for an E2E approach.

Summary: If E2E works, it works very well, and can really simplify the process;

But

There are cases where it does not work.

Wether to use E2E learning:

Pros: • Lets the data speaks: →

! If you have enough (x, y) pairs, your NN will be more able to capture the statistics behind

← For example in speech Recognition the concept of "phonemes" is good invention of linguists to analyse speech/language. But you might not want your algorithm to think in terms of phonemes and be possibly limited by their shortcommings

↳ If you don't force your Alg. then the result might be better.

- Less Hand-designed components needed.

Cons:

- May need large amount of data (x, y)
- Excludes potentially useful hand-designed components

  ↳ If you don't have a lot of data, then your ML alg. does not have much insight to gain from data. So it needs injected manual knowledge.

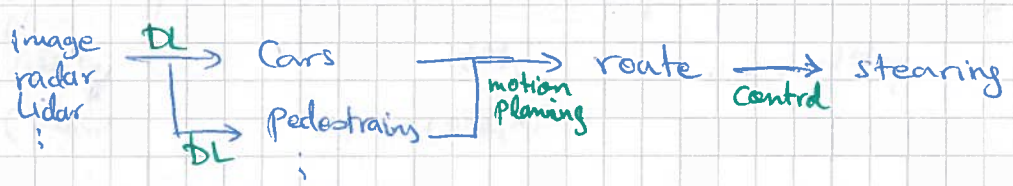
A learning algorithm can have two sources of knowledge:

- data
- hand-designed components ← Double edged sword; sometimes good. sometimes limitting.

Main Question:

Do you have enough data to learn a function of the complexity needed to map x to y?

Example: Autonomous Driving:



⇒
- Use DL to learn individual components.
- Carefully choose $x \to y$ mapping depending on what task you can get data for.

⇒ Although an E2E autonomous driving would be exciting, But it does not work well, because we donot have enough data:

image, radar → Steering

Course 2 – Improving Deep NNs – Week 2

Learning Objectives:
- different optimization methods: Stochastic GD, Momentum, RMSprop, Adam
- random mini batches
- learning rate decay

Week 2 : Optimization Algorithms

Mini-batch gradient descent (GD):

- Having fast optimization algorithms helps your team to iterate faster an be more efficient

- **Batch vs. mini-Batch GD:**
  - Batch GD: Process your entire training set:

$$\underset{(n_x, m)}{X} = \left[x^{(1)} \; x^{(2)} \; \cdots \; x^{(m)} \right]$$

$$\underset{(1, m)}{Y} = \left[y^{(1)} \; y^{(2)} \; \cdots \; y^{(m)} \right]$$

Vectorization helps fast processing of the entire TS (training set)
But what if the TS is too large? e.g. 5000,000 samples?

5000 mini-batches

  - Mini-Batch GD:

Break your TS into $\boxed{\text{smaller batches}}$: (of size, say 1000)

$$\underset{(n_x, m)}{X} = \Big[\underbrace{x^{(1)} \cdots x^{(1000)}}_{\underset{(n_x, 1000)}{X^{\{1\}}}} \Big| \underbrace{x^{(1001)} \cdots x^{(2000)}}_{\underset{(n_x, 1000)}{X^{\{2\}}}} \Big| \cdots \Big| \underbrace{\cdots x^{(m)}}_{\underset{(n_x, 1000)}{X^{\{5000\}}}} \Big]$$

mini batch

$$\underset{(1, m)}{Y} = \Big[\underbrace{y^{(1)} \cdots y^{(1000)}}_{\underset{(1, 1000)}{Y^{\{1\}}}} \Big| \underbrace{y^{(1001)} \cdots y^{(2000)}}_{\underset{(1, 1000)}{Y^{\{2\}}}} \Big| \cdots \Big| \underbrace{\cdots y^{(m)}}_{\underset{(1, 1000)}{Y^{\{5000\}}}} \Big]$$

→

Notation:

$x^{(i)}$: i-th training sample

$z^{[l]}$: z of l-th layer

$X^{\{t\}}, Y^{\{t\}}$: t-th mini batch

• Mini-batch GD:

repeat for # iterations { ← multiple iterations through the entire TS.

for $t = 1, \dots, 5000$ {

Forward prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$\vdots$$
$$A^{[L]} = g^{[L]}(Z^{[L]})$$

} Vectorized (1000 samples)

Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{1000} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \underbrace{\frac{\lambda}{2 \times 1000} \sum_{l} \|W^{[l]}\|_F^2}_{\text{if Regularization}}$

Both coming from Batch t: $X^{\{t\}}, Y^{\{t\}}$

Back prop to compute grads w.r.t. $J^{\{t\}}$ → Using only $X^{\{t\}}, Y^{\{t\}}$

Update the weights:

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

}

}

An "epoch" ! One pass through the training set

Summary:
- Do a GD for each mini-batch. (5000)
- Update params 5000 times instead of only one. !

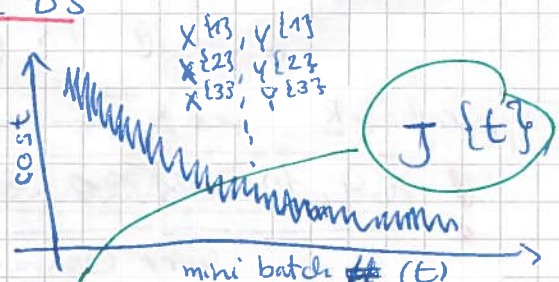
• Where as with Batch GD, a single pass throng the TS allows only one GD step with mini-batch GD, a single pass through the TS, i.e $\boxed{\text{an epoch}}$, allows 5000 GD steps! !

• Mini-batch GD works much faster than batch GD for a large DS. And almost everybody uses mini-batch GD on a large DS.

Understanding Mini-batch GD:

• Using Mini-batch GD you can start making progress and update weights even part-way through DS

vs.



plot $J^{\{t\}}$ updated using $X^{\{t\}}, Y^{\{t\}}$

$J^{\{t\}}$ has a decreasing trend, but noisy, may increase for a mini-batch, because $X^{\{t\}}$ might be easier than $X^{\{t'\}}$

Batch GD: J must monotonically decrease.

Choosing the size of mini-Batch:

Let's first consider two extremes: [Entire TS is a mini batch]

- mini batch size $= m$: Batch GD : $(X^{\{1\}}, Y^{\{1\}}) = (X, Y)$
- mini batch size $= 1$: Stochastic GD
  $(X^{\{1\}}, y^{\{1\}}) = (x^{(1)}, y^{(1)})$ .... [Every sample is a mini batch.] [Doing a GD on every single sample]

Batch GD
(mini batch size $= m$)

- low noise, large steps towards the minimum.
- too long, because you process the whole DS for each step of GD.

Stochastic GD: (mini batch size $= 1$)

- very noisy, on average goes towards the minimum, but sometimes in the wrong direction also.
- Won't ever converge. Always oscilates around the minimum.
- noisyness can be compensated by choosing a small learning rate.
- It is good that we start progress right away But:
- Main Disadvantage: Loose speedup from Vectorization

In practice the mini batch size should be somewhere in between.

- mini batch size not to big or too small.

- Fastest Learning, because:
- Vectorization speed
- make progress without processing the entire training set. Each epoch allows doing multiple GD steps.

- Like Stochastic GD, not guaranteed to always hit the minimum, but goes more consistently and faster towards the minimum
- Doesn't converge, oscilates around a very small minimum region.

Use Learning Rate Decay to solve it.

But how to choose a good mini batch size?

Guidelines:

- small TS ($m \leq 2000$) $\Longrightarrow$ just do batch GD
- Typical mB sizes: $\longrightarrow$ power of 2 (because of memory layout)

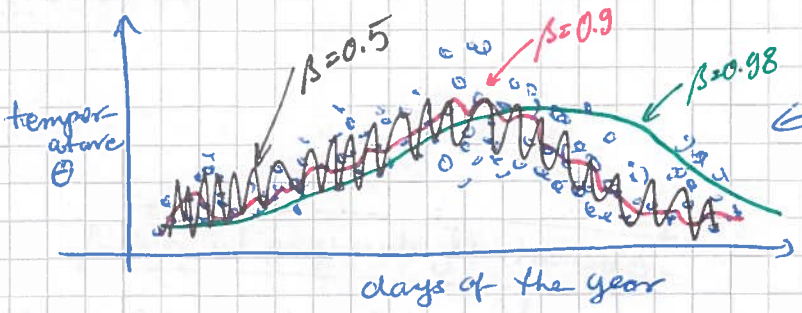
! $\underbrace{64, 128, 256, 512}_{\text{very common}}$, 1024,
↳ more rare

! Tip: Make sure the mini batch (the whole $X^{\{t\}}, Y^{\{t\}}$ set) fits into CPU/GPU memory. [depens on how large a training sample is]

- In practice the mB size is another Hyperparameter that requires a search. Try several different values and see which one makes your algorithm more efficient.

Exponentially weighted moving Average: (EWA)

- needed for some other faster optimization algorithms.



noisy temprature data

Compute the trend using a weighted moving average

Exponentionally weighted moving Avg.

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$$\begin{cases} V_0 = 0 \\ V_1 = 0.9 V_0 + 0.1\theta_1 \\ V_2 = 0.9 V_1 + 0.1\theta_2 \\ V_3 = 0.9 V_2 + 0.1\theta_3 \\ \vdots \\ V_t = 0.9 V_{t-1} + 0.1\theta_t \end{cases}$$

$\beta = 0.9 \approx$ 10 days Avg

$\beta = 0.98 \approx$ 50 days Avg

$\Leftarrow$ V_t approximately averages over $\approx \frac{1}{1-\beta}$ days temprature

- Curve is much smother, because averaging over more days. But is shifted to the right, because the window is larger and the plot adapts more slowly. (a delay)
- a lot of weight to previous values and much smaller weight to whatever you see right now.

$\beta = 0.5$ $\approx$ Avg over 2 days

- much shorter window.
  much more noisy
  much more susciptable to outliers.
  adapts much more quickly.

Understanding EWAs:

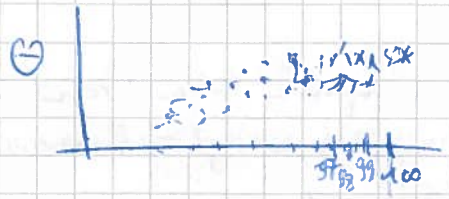
- Key component of many learning Optimization algorithms.

BRUNNEN

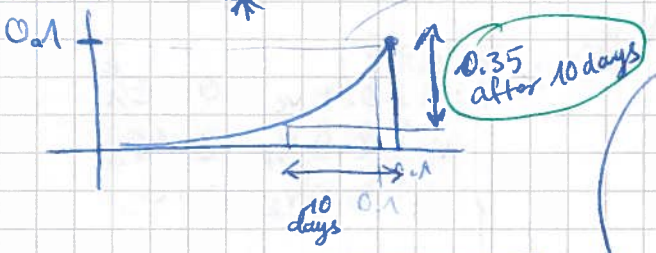
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$$V_{100} = 0.9 V_{99} + 0.1\,\theta_{100}$$
$$V_{99} = 0.9 V_{98} + 0.1\,\theta_{99}$$
$$V_{98} = 0.9 V_{97} + 0.1\,\theta_{98}$$
$$\vdots$$

$$\longrightarrow V_{100} = 0.1\theta_{100} + 0.9\big(0.1\theta_{99} + 0.9\underbrace{(0.1\theta_{98} + 0.9 V_{97})}_{}\big)$$
$$= 0.1\theta_{100} + 0.1 \times 0.9\,\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + 0.1(0.9)^4\theta_{96} + \dots$$



$*$



$\boxed{\text{exponentially decaying function}}$

for V_{100} $\left(0.1 + 0.1(0.9)\theta_{99} + 0.1(0.9)^2\theta_{98} + 0.1(0.9)^3\theta_{97} + \dots\right)$

- ~~Sum~~ of the coefficients will be near 1 up to the Bias Correction
- V_{100} = element-wise product between the two functions and summing up

$$(0.9)^{10} \simeq 0.35 \simeq \frac{1}{e}$$
$$\underbrace{(1-\overset{0.1}{\varepsilon})}_{0.9}{}^{1/\varepsilon} \simeq \frac{1}{e}$$

$\Rightarrow$ it takes around 10 days for the height of $\uparrow 1$ to decay to $1/3$ of the peak.

$\left(\frac{1}{3} \times \frac{1}{10} = 0.03\right)$ neglectable for day 11

in contrast $\beta = 0.98$

$\hookrightarrow (0.98)^{50} \simeq \frac{1}{e}$

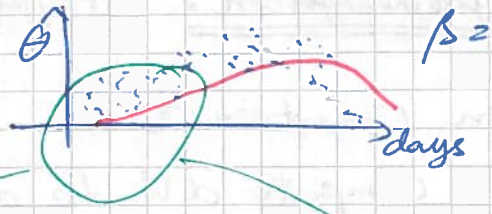
$\varepsilon = 0.02$, $(0.98)^{50} \simeq \frac{1}{\varepsilon}$ $\rightarrow$ it take 50 days for the coefficient to be come neglectable

Implementing EWAs:

```
V_θ = 0
repeat {
    get next θ_t
    V_θ := β V_θ + (1-β) θ_t
}
```

$\longrightarrow$ Very efficient, only one line and memory use is only one real number.

Bias Correction in EWAs



$\beta = 0.98$

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$V_0 = 0$

$V_1 = 0.98 \cancel{V_0}^{0} + 0.02\theta_1$

$V_2 = 0.98 V_1 + 0.02\theta_2$

$\vdots$

Start of the curve has a strong bias towards zero (as can be seen) !

Solution:

$$\frac{V_t}{1-\beta^t}$$

$t_2 : 1-\beta^t = 1-(0.98)^2 = 0.0396$

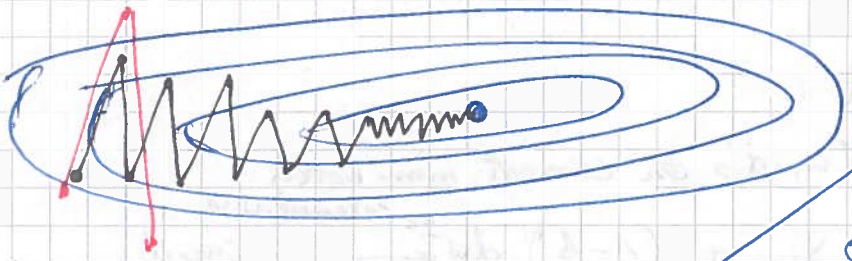
$$\frac{V_2}{0.0396} = \frac{0.0196\,\theta_1 + 0.02\,\theta_2}{0.0396} \Longrightarrow$$

$V_2 \uparrow$ becomes a weighted Avg of θ_1 and θ_2

- as $t \uparrow \longrightarrow \frac{V_t}{1-\beta t} \approx V_t \Longrightarrow$ Bias correction makes no difference.
- But as the beginning of the curve (initial phase of learning), you still warming up your estimates the bias correction can help obtaining a better estimate of parameters

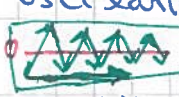
GD with momentum:

- Almost always works better than simple GD
- Idea: take the EWA of gradients to update the weights



- GD oscilates towards the minimum.
- to make sure that we don't "over shoot" and diverging
- This slows down the convergence and prevents us from using a large learning rate

we want slower learning to prevent oscilations



Averages out oscilation in ↑ direction ≈ Avg to zero :)

And force avg. on → directions

we want faster learning to aggressively move from left to right towards the minimum

Momentum:

Smothes out the steps of GD

On each iteration t:

compute dw, db on current mini batch.

$V_{dw} = \beta V_{dw} + (1-\beta) dw$

$W := W - \alpha V_{dW}$

Works for both mini batch and batch GD

Implementation:

Hyperparameter $\begin{cases} \text{learning rate } \alpha \\ \beta \end{cases}$

$\beta = 0.9$ often, most common works very well

on iteration t:

compute dW, db on the current mini-batch

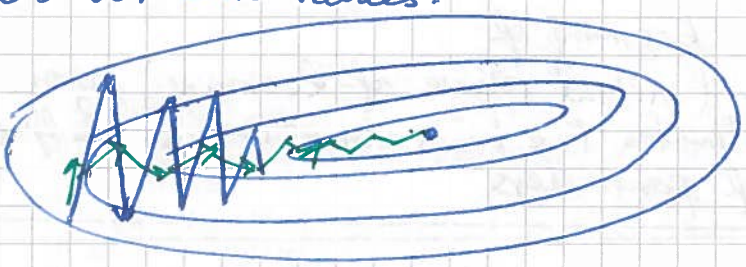
$$V_{dW} = \beta V_{dW} + (1-\beta) dW$$
$$V_{db} = \beta V_{db} + (1-\beta) db$$
$$W = W - \alpha V_{dW}, \quad b = b - \alpha V_{db}$$

- don't bother about bias correction $\cancel{\frac{V_{dW}}{1-\beta^t}}$

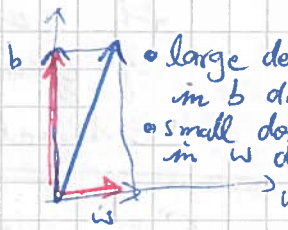
- sometimes $(1-\beta)$ is omitted (But not very intuitive). requires extra tuning of α.

RMS prop: Root Mean Squares prop.

- damp oscilations in the direction that has large oscilation (derivative values) and speed up in the direction with smaller derivative values.

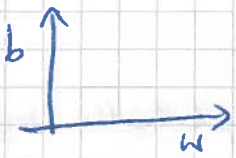


slow $\longleftrightarrow$ fast



- large derivative in b direction
- small derivative in w direction.

Imagine



vertical axis is b and horizontal axis is w $\Big\}$ we want to slow down the learning in vertical direction and speed up learning in horizontal direction.

On iteration t:

compute dw, db on current mini batch.

EWAs of the squares of the derivatives

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw^2 \longleftarrow \text{small}$$

element-wise

$$S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2 \longleftarrow \text{large}$$

$$w := w - \alpha \frac{dw}{\sqrt{S_{dw} + \varepsilon}}, \quad b := b - \alpha \frac{db}{\sqrt{S_{db} + \varepsilon}}$$

dividing by a relatively small number

So that you don't divide by zero

dividing by relatively large value.

common $\varepsilon = 10^{-8}$

- updates in vertical direction are divided by a relatively large number. This damps out the oscilation, and updates in the horizontal direction get divided by a small number and hence accelerated.
  $\Rightarrow$ we can use a much $\boxed{\text{larger learning rate}}$

- RMS prop was first proposed on a Coursera course by Geof Hinton :)

Adam Optimization: RMSprop + Momentum

- Combines the effects of GD with Momentum with effect of GD with RMS prop.
  Very effictive and commonly used across differen NN architectures on applications.

Adam: Adaptive Moment Estimation

$V_{dw} = 0,\ S_{dw} = 0,\ V_{db} = 0,\ S_{db} = 0$

on iteration t:

$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dw,\ V_{db} = \beta_1 V_{db} + (1-\beta_1) db$ ← Momentum β_1

$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dw,\ S_{db} = \beta_2 S_{db} + (1-\beta_2) db$ ← RMS prop β_2

$$\left.\begin{aligned} V_{dw}^{corrected} &= V_{dw}/(1-\beta_1^t),\quad V_{db}^{corr.} = V_{db}/(1-\beta_1^t) \\ S_{dw}^{corr.} &= S_{dw}/(1-\beta_2^t),\quad S_{db}^{corr.} = S_{db}/(1-\beta_2^t) \end{aligned}\right\} \text{bias correction}$$

$$\left.\begin{aligned} W &:= W - \alpha \frac{V_{dw}^{corr.}}{\sqrt{S_{dw}^{corr}} + \varepsilon},\quad b := b - \alpha \frac{V_{db}^{corr.}}{\sqrt{S_{db}^{corr.}} + \varepsilon} \end{aligned}\right\} \text{combined effect of Momentum and RMS prop}$$

Hyperparam. choice:

α: needs to be tuned (try a range of values an see which works)

β_1: 0.9 $\quad (dw)$ ← first moment

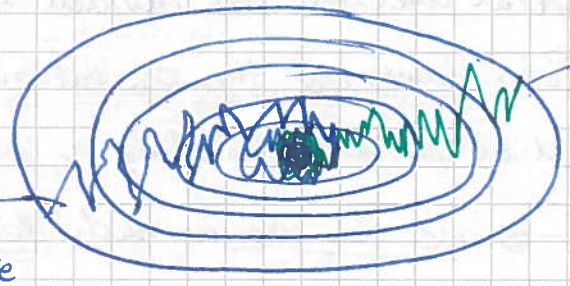
why? β_2: 0.999 $\quad (dw^2)$ ← second moment

ε: 10^{-8}

Learning Rate decay:

- Slowly reduce your learning rate over time.

→

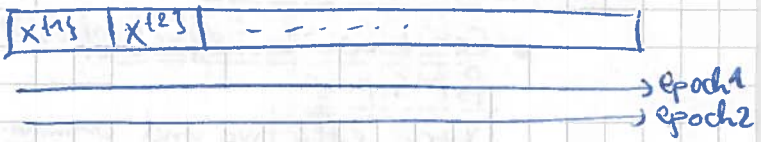


- gradually decreasing learning rate
- smaller step as we near the minimum
- wondering around a "tighter region around minimum"
- Slowly reducing α

- large learning rate
- large steps.
- noisy data
- wanders around minimum

---

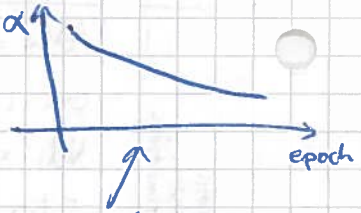
1 epoch = 1 pass throug the data



$$\alpha = \frac{1}{1 + \text{decayrate} * \text{epoch_num}}$$

$\alpha_0 = 0.2$, decayrate = 1

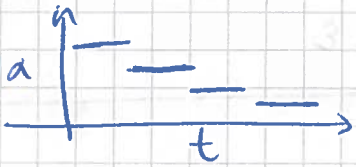
Epoch	α
1	0.1
2	0.67
3	0.5
4	0.4
⋮	⋮



---

another options:

- $\alpha = 0.95^{\text{epochnum}} \alpha_0$ ← exponential decay
- $\alpha = \frac{k}{\sqrt{\text{epochnum}}} \alpha_0$ or $\frac{k}{\sqrt{t}} \alpha_0$ ← mini batch number
- descrete steps



descrete stair case

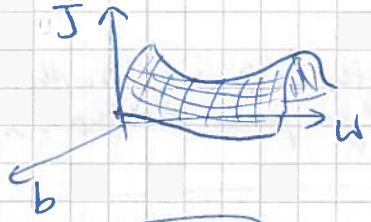
- manual decay: watche the model and tune the α depending on how the training is going on.

- Learning rate decay does help improve training, but to tune it has a lower prio and impact.

Local Optima:

- As the theory of DL advances, our understanding of local optima is changing.
- our intuition of lower dimensional spaces does not transfer well to higher dimensions

- in higher dimensions most of the ~~[illegible]~~ points are gradient = 0 **Saddle points**



- gradient is only zero in some directions, but it is very improbable that ~~[illegible]~~ it is zero in all directions.

eg. 2^{-20000} in

chance of grad is zero in all directions in a 20000 dim space

$\Rightarrow$ Local optima are not a problem.

The **plateus** (where grad is near zero for a long time) are problem. They can make learning very slow $\Rightarrow$ Momentum, RMS prop, Adam $\leftarrow$ help! speed up learning

Course 2 - Improving NNs - Week 2: Optimization, Programming 1

- understand the intuition behind Adam and RMS prop.
- Recognize the importance of mini-batch GD
- Learn the effects of Momentum on the overall performance of model.

Mini-Batch
- shuffling an partitioning are used to create mini batches
- mini-batch sizes are often powers of two: 16, 32, 64, 128, 512, ...

Momentum:

- Momentum takes past gradients into account to smooth out the steps of gradient descent. It can be applied with batch gradient descent, mini Batch GD and stochastic GD.
- We have to tune a momentum Hyperparam β and the learning rate α
- The larger the momentum β is, the smother the update, because the more we take the past gradients into account. But if β is too big it could smooth out the updates too much.
- Common values for β range from 0.8 to 0.999. If you don't want to tune this, then $\beta = 0.9$ seems a good value.
- the velocity is initialized to zero, so the algorithm will take a few iterations to build Velocity and start to take bigger steps.

$\rightarrow$

Adam: Some advantages of Adam are:

- Relatively low memory requirement (though higher than GD and GD with Momentum)
- Usually works well even with little tuning of hyperparameters (except α)

Course 2 - Improving NNs - Week 3 -

Week 3

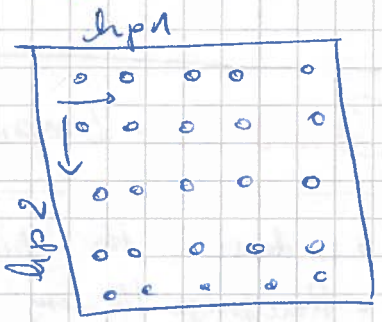
Hyperparameter Tuning, Batch Normalization, TensorFlow

Hyperparameter Tuning:

- α [1]
- β [2] $= 0.9$
- $\beta_1, \beta_2, \varepsilon$ (0.9, 0.999, 10^{-8}) ← Andrew almost never tunes these
- # layers [3]
- # hidden units [2]
- learning rate decay [3]
- mini batch size [2]

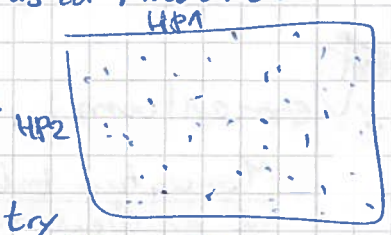
Old common practice:

Grid Search



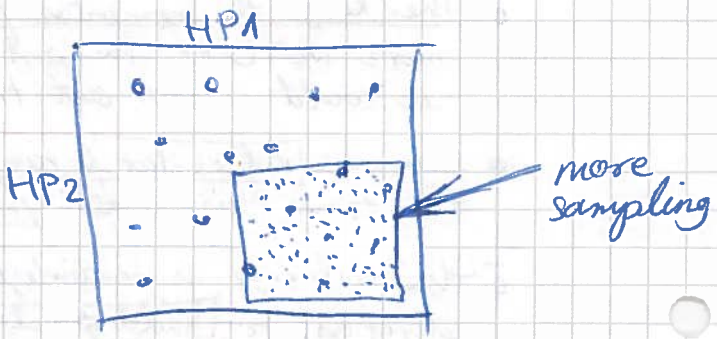
- Works good only if num of HPs is very small

- in D2 it is better to choose the points at random.
  - → difficult to know in advance which HP is the most important for your problem.
  - → more likely to find better values for more important HPs. Because you try more different values and are not bound to the value of other HPs.



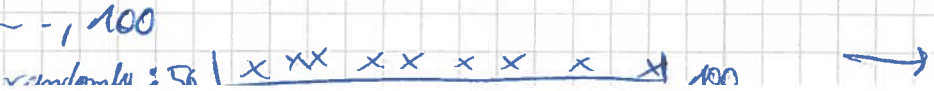
Random Search ensures more richly exploration of HP space

- **Coarse to fine search:**



- **Appropiate scale to pick HPs**

$n^{[l]} = 50, \ldots, 100$



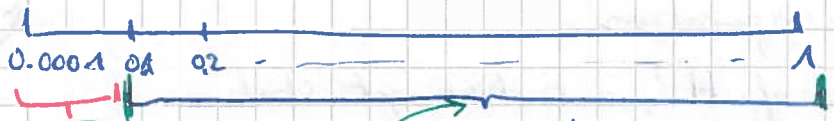
layers L: 2–4, Randomly 2, 3, 4

- Sometimes uniformly random sampling is not good for some HPs

For example α:

range 0.0001 1

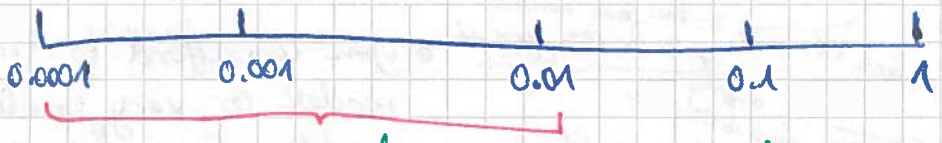
uniform scale:



90% of resources spent for range 0.1 to 1

But what if the range 0.0001 to 0.1 is more important?

instead use logarithmic scaling:



much more resources for range 0.0001 to 0.1.

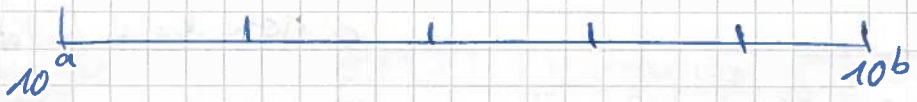
$r = -4 * np.random.rand()$ ← $r \in [-4, 0]$

$\alpha = 10^r$ ← $\alpha \in [10^{-4}, 1]$

---

trying to sample between 10^a to 10^b on the log scale:

$r \in [a, b]$ $\alpha \in [10^a, 10^b]$, $\alpha = 10^r$



HP for EWAs: β

$\beta = 0.9$ 0.999

last 10 values — last 1000 values

uniform sampling on linear scale

is not effective, does not make sense:

Because $\frac{1}{1-\beta}$ is very sensitive to small values of β

$\beta: 0.9 \to 0.9005$ } ~ 10 days

$\beta: 0.999 \to 0.9995$ } from 1000 dys to 2000 days

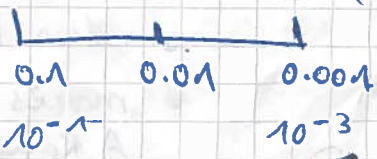
uniformly is not good!

Ok, think we want to sample values for $1-\beta = 0.1$ 0.001

$r \in [-3, -1]$, $1-\beta = 10^r$

$\beta = 1 - 10^r$

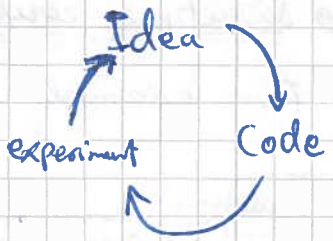
→ efficient in exploring the space of possible outcomes



Sample more density in the regime that β is close to 1. More

HP Tuning in Practice: Pandas vs. Caviar :

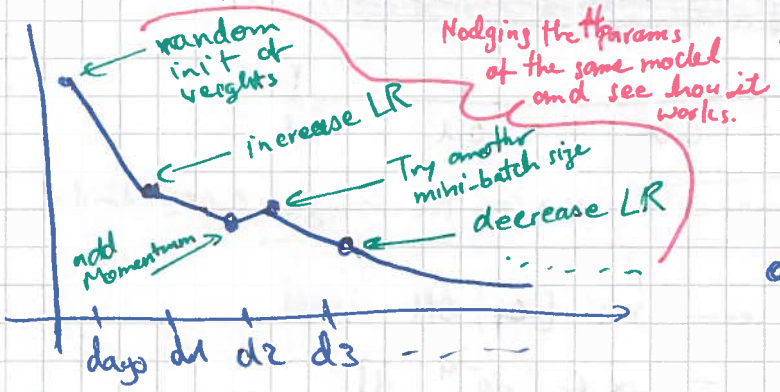
- Intuitions from one domain may or may not transfer to other domains
  - → cross fertalization
  - → Insporation



- In terms of HP, intuitions get stale.
  - ↳ Reevaluate occasionally. At least once in every several months.

- Two Strategies for searching HPs.

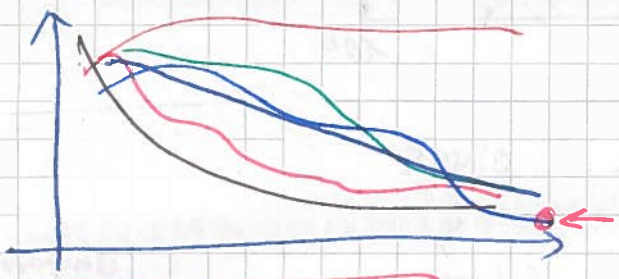
- Baby Sitting One Model:
  - you have a huge DS.
  - but not much CPUs/GPUs
  - you can afford to train one model or very small number of models at a time.
  - And gradually baby-sit, watch that model as it is training.



Panda Strategy: Baby-sitting one Model

Panda

- Training multiple models at the same time:
  - you have a lot of CPU/GPUs
  -



Caviar

Caviar Strategy: Just start many models with many different HPs and see how they work.

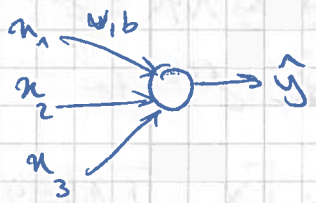
Batch Normalization: Normalizing Activations in a Network

Batch Normalization

- makes the HP search much easier.
- makes the NN much more robust to the choice of HP.
  A Range of HPs work well.

Normalizing the inputs speeds up Learning:

Batch Norm



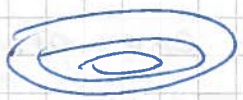
$$\mu = \frac{1}{m}\sum_i x^{(i)}$$

$$X = X - \mu$$

$$\sigma^2 = \frac{1}{m}\sum_i x^{(i)2}$$ element-wise

$$X = X/\sigma$$

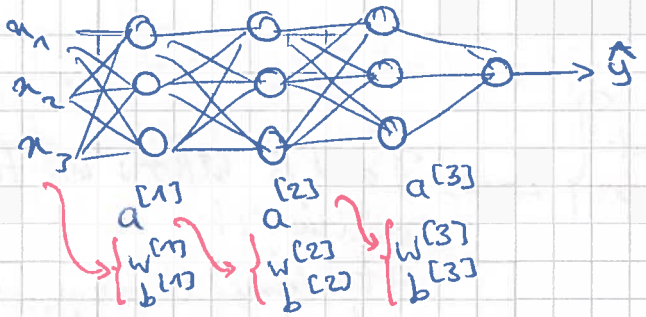
DS before normalization



DS After normalization

- easier for GD
- faster

- So far we had it for a simple logistic Regression model.
- What about a deeper models?



- Can't we normalize $a^{[l]}$ so that the input to the next ~~layer~~ layer is normalized? and training $W^{[l+1]}, b^{[l+1]}$ is faster?
- This is what batch norm does. It ~~also~~ does normalize $\boxed{Z^{[l]}\text{s}}$
- There is a debate whether to normalize $Z^{[l]}$'s or $a^{[l]}$'s?

Recommended: Normalize $Z^{[l]}$

- more often in practice
- use as default choice.

Impl:

Given intermediate values in NN

$Z^{[l](i)}$ for each layer and train sample

$i = 1 \dots m$

* We omit $[l]$ for readability.

$$\mu = \frac{1}{m}\sum_i Z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (Z^{(i)} - \mu)^2$$

$$Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$\Longrightarrow$ Each component of Z has now mean 0 and Standart ~~de~~ unit (1) variance

$$\tilde{Z}^{(i)} = \gamma Z^{(i)}_{norm} + \beta$$

learnable params of the model. Get updated just like the weights.

But don't want the HUs to always have mean zero and var 1. Maybe it makes sense for HUs to have a different distrib

γ and β allow to set the mean and var of $\tilde{Z}$ to whatever we want it to be.

Lets HU values have other means (from 0 and 1) and variance if needed

Now we use $\tilde{Z}^{(i)}$ intead of $Z^{(i)}$ for computations of NN.

In fact:

If $\gamma = \sqrt{\sigma^2 + \varepsilon}$, $\beta = \mu$

Then $\tilde{Z}^{(i)} \to i$

- Using Batch norm we applies normalization not only to X (input layer) but also to values deep in some hidden units in NN : $z^{(i)}$

- But the difference is, you might not want the $z^{(i)}$'s to have forces mean 0 and variance 1.
  So Batch norms allows this by introducing $\boxed{\gamma \text{ and } \beta}$ $\left(\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta\right)$
  For example if we had a Sigmoid activation function, we might not want our values to be always at the area of linear regime of the sigmoid function, in order to take advantage of the non-linearity of Sigmoid function.

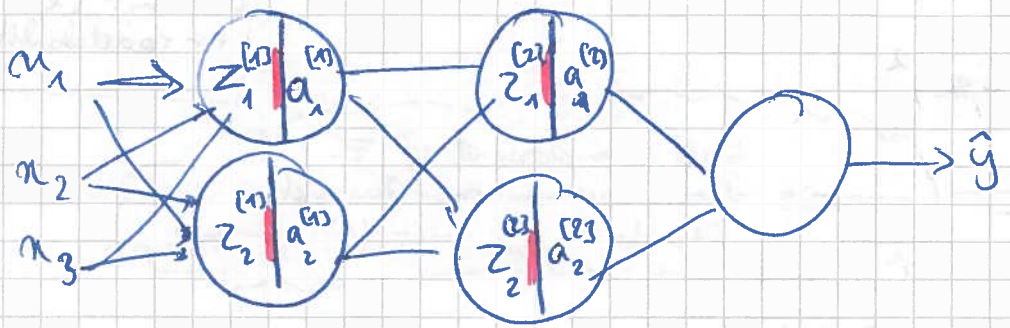
We might not want ~~this~~ our values to always be clustered here.

We may want the values to have a larger varience and a mean $\neq 0$

$\Longrightarrow$ γ and β allow us to achive it.
At same time we can make sure our $z^{(i)}$'s have standardized mean and varience that is controlled by γ and β.

$\Longrightarrow$ Batch norm standardizes the mean and the varience of HU values ($z^{(i)}$'s) to have some fixed mean and variance that is controlled by γ and β and which the algorith can learn the values of γ and β.

Fitting Batch Norm into a NN:



$$X \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[\text{BN}]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \longrightarrow a^{[1]} = g^{[1]}(\tilde{z}^{[1]}) \xrightarrow{W^{[2]}, b^{[2]}} z^{[2]} \xrightarrow[\text{BN}]{\beta^{[2]}, \gamma^{[2]}} \tilde{z}^{[2]} \longrightarrow$$

With BN the model parameters become

$$W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots W^{[L]}, b^{[L]}, \beta^{[1]}, \gamma^{[1]}, \beta^{[2]}, \gamma^{[2]}, \dots \longrightarrow$$

- Now we use whatever optimization that we want, such as GD, in order to ~~update~~ update β's and γs

$$d\beta^{[l]} \rightsquigarrow \beta^{[l]} = \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$d\gamma^{[l]} \rightsquigarrow \gamma^{[l]} = \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

BN is in practice applied with mini-Batches:

$$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \longrightarrow g^{[1]}(\tilde{Z}^{[1]}) = a^{[1]} \xrightarrow{W^{[2]}, b^{[2]}} Z^{[2]} \dots$$

$$X^{\{2\}} \longrightarrow Z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{Z}^{[1]} \dots$$

- For each MB, the $\tilde{Z}^{[l]}$ gets normalized using $\boxed{\text{just the } X^{\{i\}} \text{ mean and variance}}$

- If you are using BN we can $\boxed{\text{eliminate } b^{[l]}\text{'s from model:}}$

  see why: $Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

  but BN first normalizes $Z^{[l]}$'s to have mean zero and standard variance, and then rescale by β and γ.

  $$Z^{(i)}_{norm} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

  But this means that what ever is the value of $b^{[l]}$ it just gets subtracted out.

  Because, we first compute then means of $Z^{[l]}$ and then subtract it ~~from all~~. So adding a constant to all examples of mini batch, does not leave an effect. Because it gets subtracted and canceled out in mean subtraction step.

- The dimensions: $Z^{[l]}$, $\beta^{[l]}$, $\gamma^{[l]}$

  $(n^{[l]}, 1)$ $(n^{[l]}, 1)$ $(n^{[l]}, 1)$?? $\longrightarrow (1, n^{[l]})$??

Impl using GD:

for $t = 1 \dots$ # mini-Batches

compute forward prop on $X^{\{t\}}$

In each hidden layer use BN to replace $Z^{[l]}$ with $\tilde{Z}^{[l]}$

Use backprop to compute $dW^{[l]}, d\beta^{[l]}, d\gamma^{[l]}$

Update params

$$W^{[l]} := W^{[l]} - \alpha\, dW^{[l]}$$
$$\beta^{[l]} := \beta^{[l]} - \alpha\, d\beta^{[l]}$$
$$\gamma^{[l]} := \gamma^{[l]} - \alpha\, d\gamma^{[l]}$$

works with all other optimization algors as well:

RMS prop,
Momentum,
Adam.

Why Does BN work?

- We saw that normalizing input features X to have similar range of values can speed up learning.
- BN is doing a similar thing for the HDs. But there is more to it

Learning on shifting input distributions:

- BN makes weights deeper in the NN more robust to weights changes in earlier layers.

Covariate Shift

- Compensates the problem of "covariate shift" ! for the input to each layer, by making sure that the mean and variance of the input to layer remains the same, and hence the distribution of the input does not change.

- limits the amount that changing the values of the first layer can affect the distribution of the values that the third layer sees, and has to learn on.
- so, BN reduces the problem of the distribut. of input values to layers changing.
- reduces the coupling of the distribution of the inputs to each layer, so each layer can learn a little bit independently of the others.

$\Longrightarrow$ Increases the robustness and speed of learning.

BN as Regularization:

- Each mini-Batch is scaled by the mean/variance compute on just that mini Batch.
- This adds some noise to values $Z^{[l]}$ within that mini batch. So, similar to dropout it adds some noise to each hidden layer's activation
  - $\hookrightarrow$ multiplicative (γ) and additive noise (β)
- This has a slight regularization effect.
  - $\hookrightarrow$ gets reduced with larger batch size, e.g. 512.

Batch Norm at test time:

- we don't have any mini batches in test time. So how to compute μ and σ^2 for BN? $\longrightarrow$

$$\mu = \frac{1}{m}\sum_i z^{(i)}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z^{(i)} - \mu)^2$$

$$z^{(i)}_{norm} = \frac{z^{(i)} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$$

$$\tilde{z}^{(i)} = \gamma z^{(i)}_{norm} + \beta$$

During training:

- keep track of μ's and σ^2's of each mini batch, and create an estimate of μ and σ^2 using a EWA of those values.

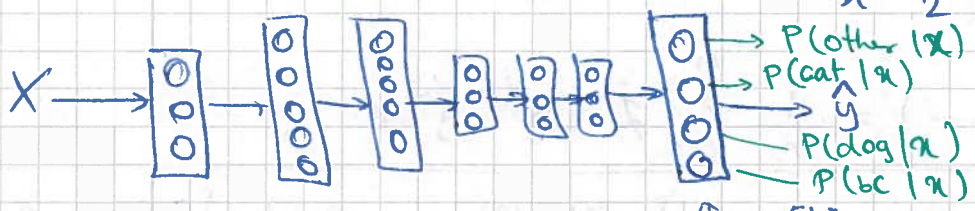
$X^{\{1\}}, X^{\{2\}}, X^{\{3\}}, \ldots$

- $\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]}$ ← do EWA ⇒ μ, σ^2
- $\sigma^{2\{1\}[l]}, \ldots$

Multi-class classification: Softmax Regression

- Softmax Regression: A generalization of logistic Regression for multiple classes.

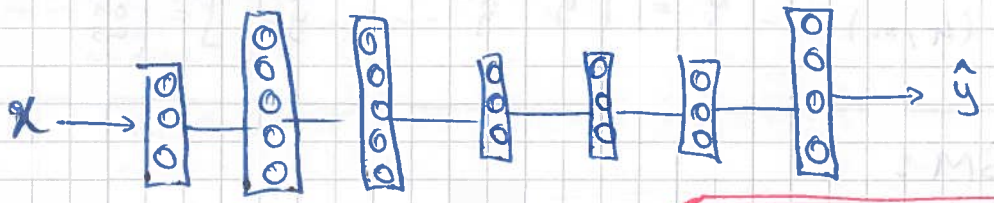
C = #classes, e.g. 4 for {cat, dog, chicken (bc), other}
1 2 3 0



$\hat{y} \rightsquigarrow (4,1)$

$n^{[L]} = 4 = C$

Softmax layer:



$\rightarrow z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$

$a^{[L]} = g^{[L]}(z^{[L]})$

$a^{[L]}$: (4,1) ; $z^{[L]}$: (4,1)

Softmax activation function:

element-wise

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

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_j t_j}$$

→ normalize so that sum = 1

[Because these are Probabilities]

Examples: (not important)

Training a Softmax Classifier:

- Softmax is a contrast to "hard max"
  name (sm) (hm)

$sm = \begin{bmatrix} 0.844 \\ 0.042 \\ 0.002 \\ 0.114 \end{bmatrix}$

$hm = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ →

- SM activation function generalizes the logistic Regression to C classes.
  - ↳ if $C = 2$, then SM results to logistic regression.
    (proof schema: if the sum of both probabilities is one, it is redundant to compute both of them. So only compute one of them => lgReg.)

Loss Function:

$C = 4$

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \text{— cat} \qquad a^{[L]} = \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$$\boxed{\mathcal{L}(\hat{y}, y) = -\sum_{j=1}^{4} y_j \log \hat{y}_j} = -y_2 \log \hat{y}_2 = -\log \hat{y}_2$$

Softmax Loss Function

↳ because $y_1 = y_3 = y_4 = 0$, $y_2 = 1$

MLE => in order to make $\mathcal{L}(\hat{y}, y)$ small, you have to make $\hat{y}_2$ as high as possible

↳ Maximum Likelihood Estimate

Cost: $J(W^{[1]}, b^{[1]}, \dots) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

Vectorized Impl.:

$$\underset{(4,m)}{Y} = \begin{bmatrix} y^{(1)} & y^{(2)} & \cdots & y^{(m)} \end{bmatrix} = \begin{bmatrix} 0 & 0 & & 1 \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & & 0 \\ 0 & 0 & & 0 \end{bmatrix}$$

$$\underset{(4,m)}{\hat{Y}} = \begin{bmatrix} \hat{y}^{(1)} & \hat{y}^{(2)} & \cdots & \hat{y}^{(m)} \end{bmatrix} = \begin{bmatrix} 0.3 & & \\ 0.1 & \cdots & \\ 0.5 & & \\ 0.1 & & \end{bmatrix}$$

GD with SM:

Forward:

$$\underset{(4,1)}{z^{[L]}} \longrightarrow a^{[L]} = \hat{y} \longrightarrow \mathcal{L}(\hat{y}, y)$$

Backward:

$$\frac{\partial J}{\partial z^{[L]}} \quad \underset{(4,1)}{dz^{[L]}} = \hat{y} - y \quad !$$

$$(e^u)' = u' e^u$$

DL Frameworks:

- caffe / caffe 2
- CNTK
- DL4J
- Keras
- Lasagne
- mxnet
- PaddlePaddle
- TF
- Theano
- Torch

→

Choose of a DL Framework:

- Ease of programming (development & deployment)
- Running speed
- Truely open (open source with good governance)

TensorFlow: see the video

tf.Variable, tf.Placeholder, tf.train.GradientDescentOptimizer(--).minimize(cost)
tf.session, session.run.

- The heart of a tensor flow program is something to compute a cost. And tf automatically figures out how to compute the derivatives and how to minimize that cost.
- TF constructs a Computation Graph, and does forward prop through the Comp. Graph.

Course 2 - Improving NN - Week 3 - Programming 1 - TF

- tensors
- operations
- computation graph
- init variables
- start session
- train algorithm
- Implement an NN

```
a = tf.constant(2)
b = tf.constant(10)
c = tf.multiply(a, b)
print(c)
```

↳ Tensor("Mul:0", shape=(), dtype=int32) → "Just put in the computation graph."

TF general steps:

1. Create tensors (variables) that are not executed or evaluated yet.
2. write operations between those tensors
3. intialize your tensors
4. create a session
5. Run the session. This will run the operations you'd written above.

- placeholders, feed-dict variable.
  ↳ pass in values later as you run the session.

```
x = tf.placeholder(tf.int64, name="x")
print(sess.run(2*x, feed_dict={x:3}))
sess.close()
```

Feed Data.

Cost:

Logits

$$-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\sigma(z^{[2](i)})+(1-y^{(i)})\log(1-\sigma(z^{[2](i)}))\right)$$

logits

tf.nn.sigmoid_cross_entropy_with_logits()
↳ logits, lables

1. Create computation grap with place holders
2. Run graph by feeding values for Placeholders

! TF !

One Hot encoding: # classes = C = 4 {0, 1, 2, 3}

$y = [1 \quad 2 \quad 3 \quad 0 \quad 2 \quad 1]$

is converted to

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

Only one is hot

- The two main object classes in TF are Tensors and Operators.
- The backpropagation and optimization is done automatically when running the session on the "optimizer" Object.

Course 4 - CNN - Week 1 -

Week 1 Foundations of CNNs:

Learning Objectives:

- Understand the convolution operation
- Understand the pooling operation
- Vocabulary used in cnns (padding, stride, filter, ...)
- Build a CNN for image multi-class classification.

- Why a course on DL in CV?
  - rapid advances in CV and brand new methods
  - ! - cross fertilization the innovation from CV community to other application areas.

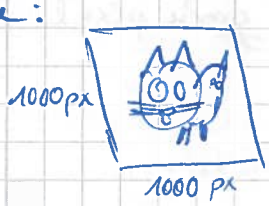
CV problems:

- Image classification / recognition.
- Object detection
- Neural style transfer

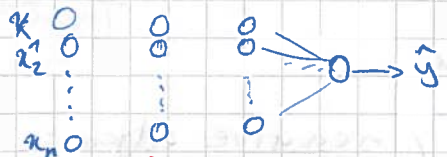
Challenge of DL in CV:

- $\boxed{\text{Very large}}$ number of input features.

for example:



$1000 \times 1000 \times 3 = 3$ Milions



3M

1000

$x \in \mathbb{R}^{3M}$

$\Rightarrow W^{[1]}$ (1000, 3M) Matrix!

$\boxed{3 \times 10^9}$ parameters!

! - very difficult to get enough data to prevent over fitting

- computational req. and memory req. are too high.

$\Rightarrow$ In order to be able to work with large high res images we use Convolution

---

Edge Detection Example:

Edges (earlier layers) → Part of Object (later layers) → Complete Object (last layers)

Convolution: is like scaling (or multiplying) one function using another function at points they overlap !

→ Vertical Edges

→ Horizontal Edges

$3 \times 1 + 1 \times 1 + 2 \times 1 + 0 \times 0 + 5 \times 0 + 7 \times 0 + -1 \times 1 + -1 \times 8 + -1 \times 2 = -5$

3	0	1	2	7	9
1	5	8	9	3	1
2	7	2	5	1	3
0	1	3	1	7	8
9	2	1	6	2	8
2	4	5	2	3	9

6x6

*

1	0	-1
1	0	-1
1	0	-1

3x3 $\boxed{\text{Filter}}$ // $\boxed{\text{Kernel}}$

convolve

→

-5	-4	0	8
-10	-2	2	3
0	-2	-4	-7
-3	-2	-3	-16

4x4

A vertical edge detector :)

$$\begin{bmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{bmatrix}$$

6x6 — The edge — 3x3 — 4x4

Edge

Light to dark

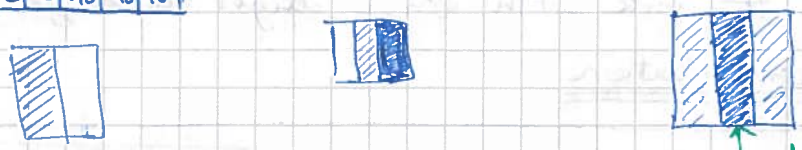
The Edge

light ~~edge~~ to dark transition

An intuitive decription of an edge could be a 3x3 region where there are bright pixels on the left and dark pixels on the right.

- Positive / negative edges:
  - ↳ light to dark
  - ↳ dark to Light

$$\begin{bmatrix} 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \\ 0 & 0 & 0 & 10 & 10 & 10 \end{bmatrix} * \begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \\ 0 & -30 & -30 & 0 \end{bmatrix}$$



dark to light

edge ~~dark~~ dark to light transition.

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

Vertical Edge — Horizontal Edge

Different filters allow you to detect different kind of edges !

~~Question has been for long time~~

Learning to detect Edges:

- for a long time there has been a debate on what is the best set of numbers to use in a filter?

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 3 & 0 & -3 \\ 10 & 0 & -10 \\ 3 & 0 & -3 \end{bmatrix}$$

Sobel filter — Scharr filter

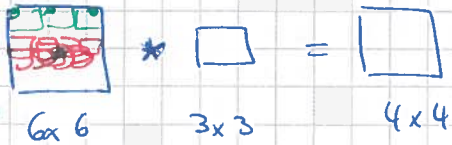
(more robust, because puts more weight to)

- With DL you may no more need to hand-pick the filter. You can learn it.
- It becomes even more robust

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

Padding: Problems of Convolution:

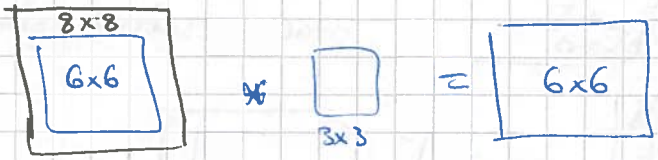
- Shrinking Output



$n \times n \; * \; f \times f \rightarrow (n-f+1) \times (n-f+1)$

- Throwing away a lot of information from edges.
  (The pixels on the edges only contribute few times to the output compared with the pixels in the center.)

$\Longrightarrow$ Pad the image before applying the convolution.



p = padding = 1 px

$\hookrightarrow$ output:

$$(n+2p-f+1) \times (n+2p-f+1)$$

$6 \times 6 \; * \; 3 \times 3 \qquad 4 \times 4$

How much to pad?

- $\hookrightarrow$ Valid Convolution $\longrightarrow$ no padding $\quad n \times n \; * \; f \times f \longrightarrow (n-f+1) \times (n-f+1)$
- $\hookrightarrow$ Same Convolution $\longrightarrow$ output size the same as input size:

$$n+2p-f+1 = n \Rightarrow p = \frac{f-1}{2}$$

filter 3×3
$\Rightarrow p = 1$ px

- By Convention in CV, f is always odd.

Strided Convolutions:

- Jump the filter by s steps (vertically and horizontaly)

$$n \times n \; * \; f \times f \longrightarrow \left\lfloor \left(\frac{n+2p-f}{s}+1\right) \times \left(\frac{n+2p-f}{s}+1\right) \right\rfloor$$

padding p
stride s

floor

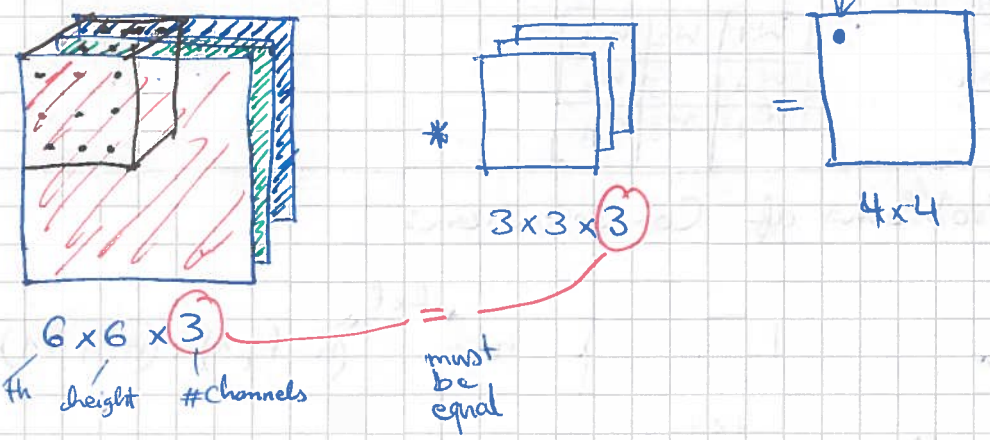
Note on cross-correlation vs. Convolution:

associative operation.

- In math text books convolution is flipping the filter horizontally and vertically before applying the operation.
  - What we do here is technically a cross-

Convolutions over Volumes:

Sum of 27 multiplications each filter channel with the corresponding image channel

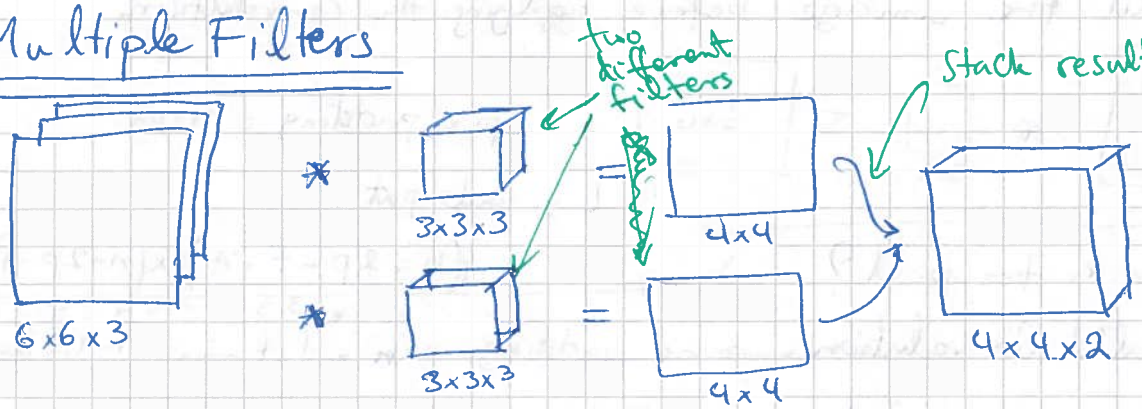


Example: detect Edges

- detect vertical edges in red channel only
- detect vertical edges in all channels

R	G	B
$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$
$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$	$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix}$

Multiple Filters



Summary: $n \times n \times n_c \;*\; f \times f \times n_c \longrightarrow (n-f+1) \times (n-f+1) \times n_c'$

$6 \times 6 \times 3 \quad 3 \times 3 \times 3 \quad 4 \times 4 \times 2$

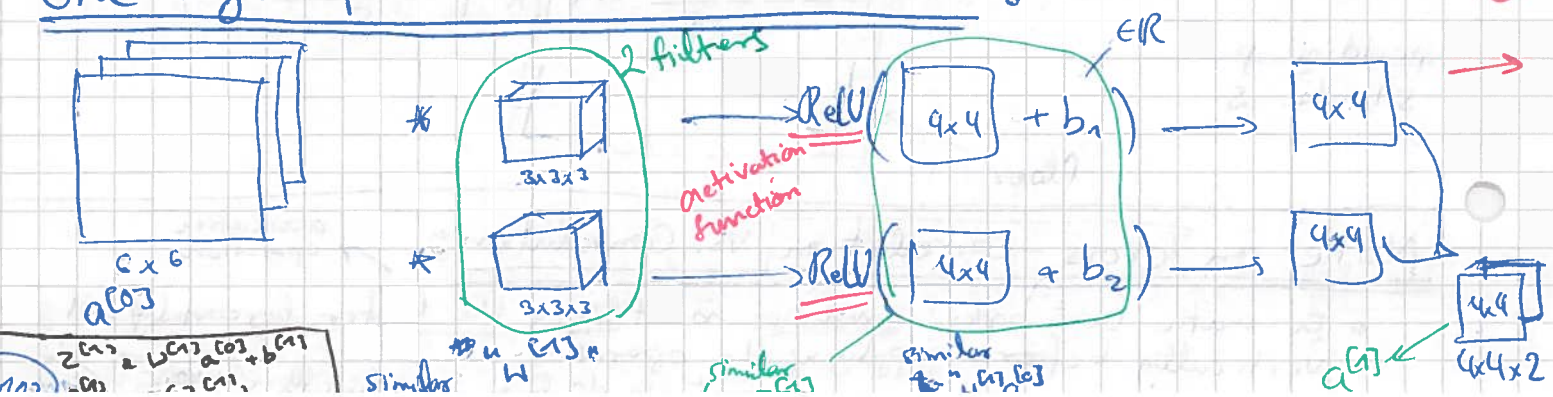
n_c: #channels (also called depth)

n_c': #filters

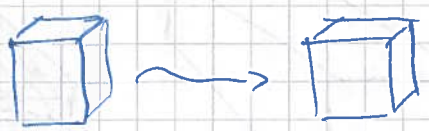
- Convolution on volumes is very powerful.
  - ↳ allows you to detect multiple features at the same time and the output will have the number of channels equals to number of features you are detecting.

One layer of a Convolutional Network:

One Convolutional Layer



One layer of convolutional network:



6x6x3 → 4x4x2

input $x = a^{[0]}$ $\quad a^{[1]}$

- If we had 10 filters we would have an $a^{[1]}$ as $4 \times 4 \times 10$

$\boxed{\text{Output volume}}$

Example: 10 filters in one layer of a Neural Net. How many param?

3x3x3

$3 \times 3 \times 3 = 27$ $\quad 10 \Longrightarrow 280$ parameters

+ bias $\longrightarrow 28$

!! • Nice thing about it:

- no mater how large the input image is (1000x1000; 5000x5000; ...) the number of parameters of the model that we need to learn remains the same
- with 10 filters, we can learn different thing at a time: H_edges, V_edges, diagonal, ...
- This smaller number of params makes our network less prone to overfitting.
- Once you have learned the 10 filters (280 params) you can apply this to very large images.

Notation:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$

Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$$n_{H/W}^{[l]} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

$n_c^{[l]}$ = number of channels = number of filters

Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$ ← the number of channels in filter must match the number of channels in the input.

Activations: $a^{[l]} \longrightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$ | Vectorized: $A^{[l]} \longrightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

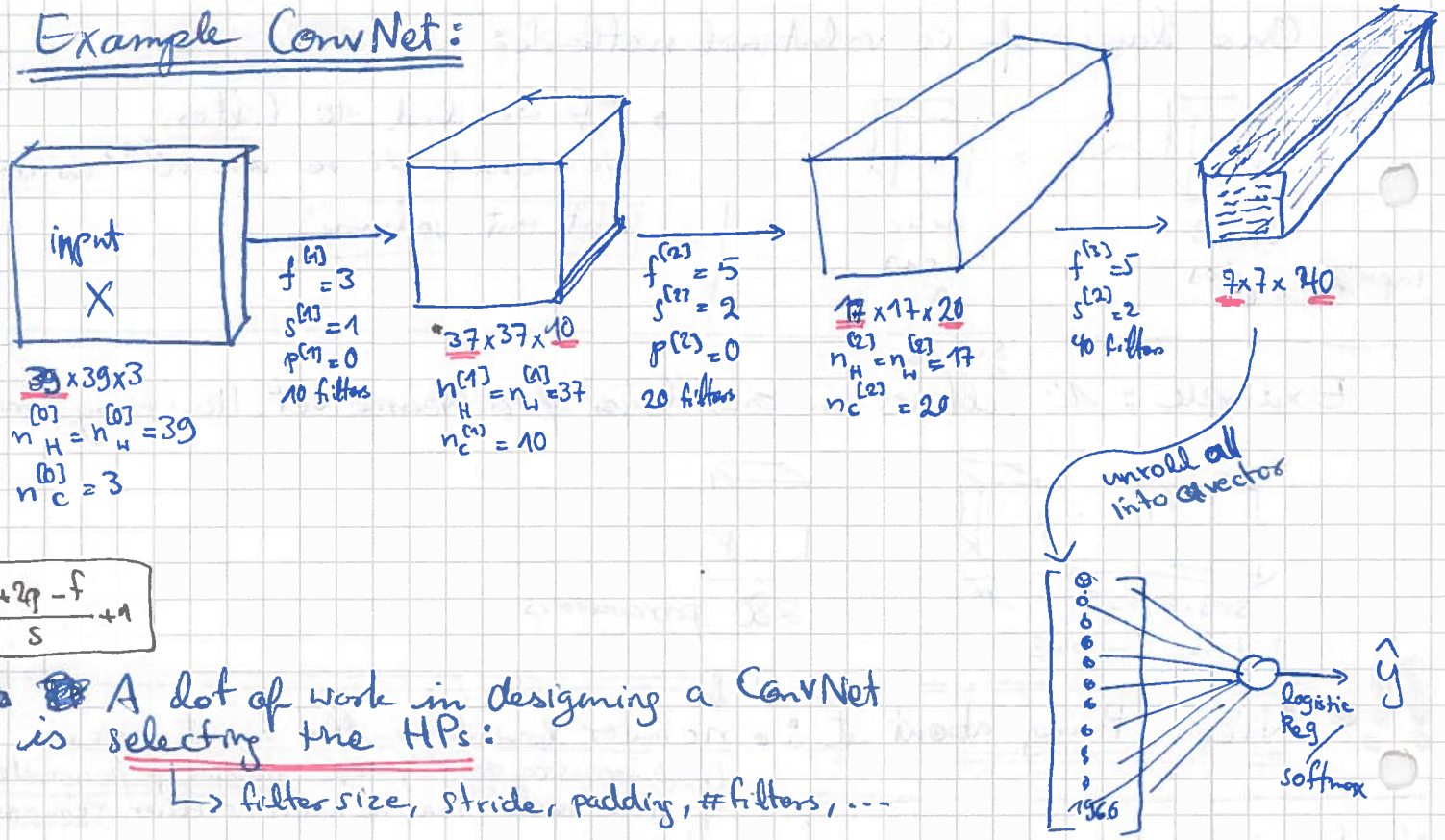
# of training Examples

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$ ← # filters in layer l

bias: $n_c^{[l]}$ (later in code $(1,1,1,n_c^{[l]})$ a four dimensional vector)

$\boxed{\text{A Convolutional layer}}$

Example ConvNet:



$$\boxed{\frac{n+2p-f}{s}+1}$$

- A lot of work in designing a ConvNet is selecting the HPs:
  - ↳ filter size, stride, padding, # filters, ...

- Typical trend:
  - Image size decreases as we go deeper in network 39, 37, 17, 7
  - # channels generally increases. 3, 10, 20, 40

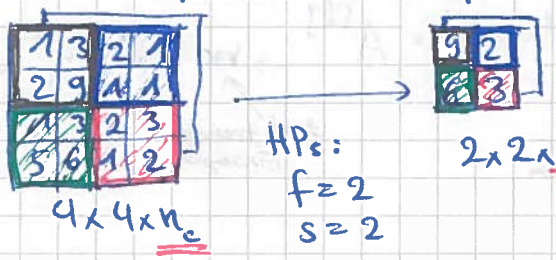
Types of Layers in CNNs:

- Convolution (CONV)
- Pooling (POOL)
- fully connected (FC)

Pooling:

- reduce the size of the representation
- increas speed of computation
- make feature detection more robust.

Example: Max Pooling



HPs:
$f = 2$
$s = 2$

- take ~~Max~~ of each all areas.
- Intuition: Pool the most distinct feature
- No param to learn by GD: fixed computation
- Max pooling is applyed on all layers independently. ⟹ number of channels in input and output are equal. !

Average Pooling

↳ • Same as max pooling, but instead of computing the max of an area, compute its Avg.

↳ used less often than max pooling.
Some times very deep in NN to collapse the representation

HPs for Pooling:

- f: filter size
- s: stride
- max or avg?

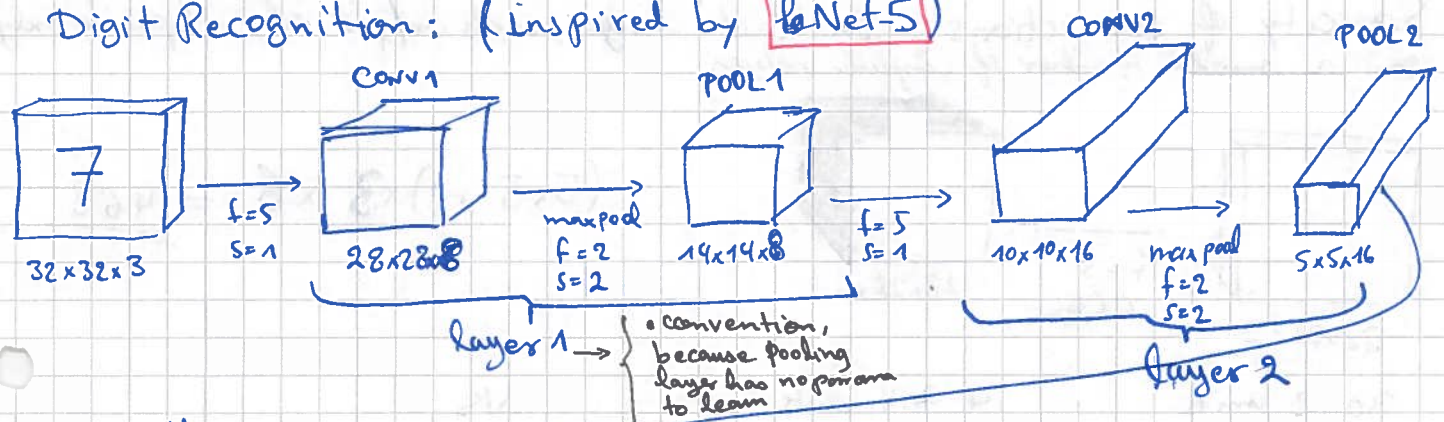
typical values: $f=2, s=2$; $f=3, s=2$

Input: $n_H \times n_W \times n_c \longrightarrow$ Output: $\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_W - f}{s} + 1 \right\rfloor \times n_c$

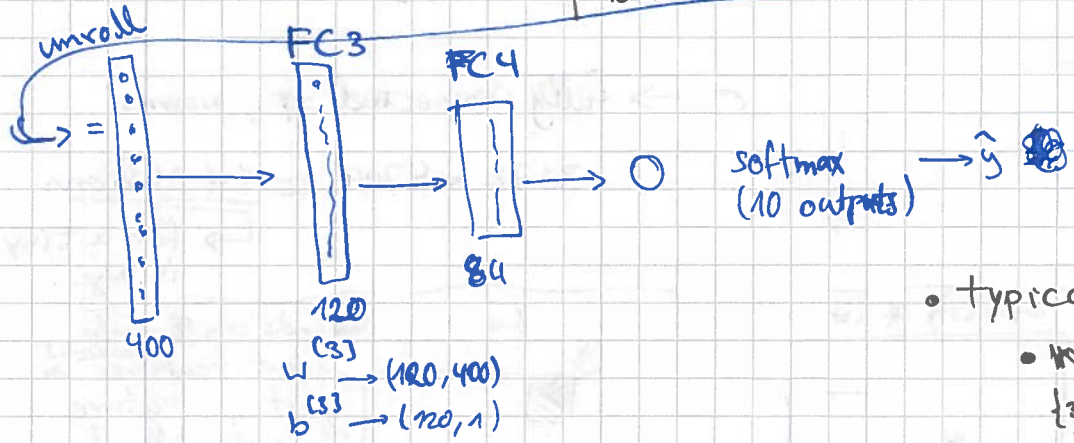
In POOLing # channels in input and output are the same

CNN Example: with FC Layers

Digit Recognition: (inspired by LeNet-5)



• convention, because pooling layer has no param to learn



$W^{[3]} \rightarrow (120, 400)$

$b^{[3]} \rightarrow (120, 1)$

10-digits: 0, 1, 2, 3, 4 ... 9

- typical patterns in CNN
  - $n_H, n_W \downarrow$, $n_c \uparrow$
    $\{32, 28, 14, 10, 5\}$ $\{3, 8, 16\}$
- typical pattern in architecture

CONV – POOL – CONV – POOL .... – FC – FC – FC ... SOFTMAX

	Activation Shape	Activations Size	# Params
Input	(32,32,3)	3072 $a^{[0]}$	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	608
POO1	(14,14,8)	1568	0
CONV2 (f=5, s=1)	(10,10,16)	1600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
SOFTMAX	(10,1)	10	841

decreases

! See Forum

Numbers are misleading!

Params at CONV ℓ

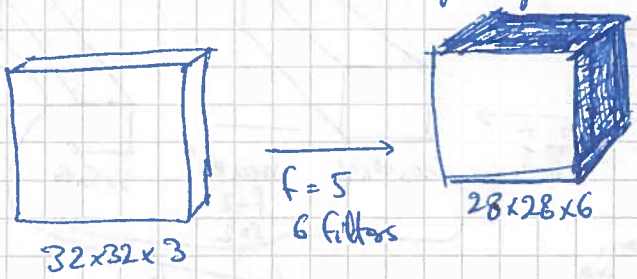
$(f^{[\ell]} \times f^{[\ell]} \times n_c^{[\ell-1]} + 1) \times n_c^{[\ell]}$

Why Convolutions?

Two main advantages: • parameter sharing
• sparcity of connections

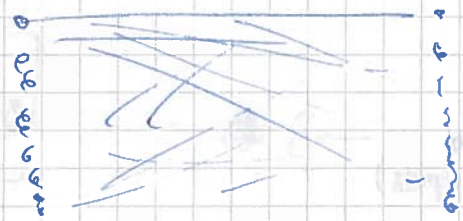
- ! → Parameter Sharing: A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image too.

- Sparcity of Connections: In each layer, each output value depends only on a small number of input values



$(5\times5+1)\times3\times6 = 468$

3072 units 4,704 units vs.

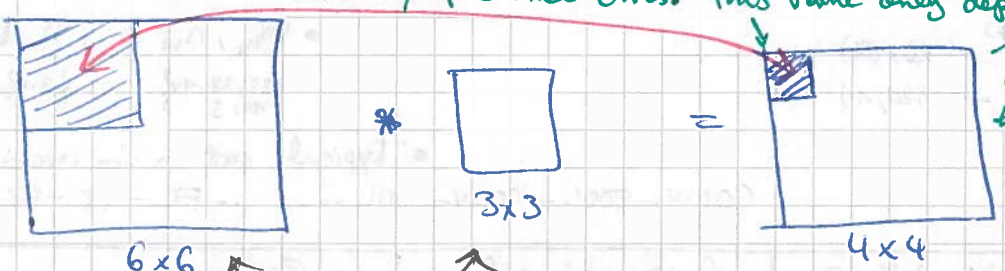


→ fully connected # params

$3072 \times 4704 \approx 14$ Milions

↳ for a tiny 32x32px image!

Sparcity of Connections: this value only depends on only 9 input features and the rest of features does not have an effect on it.



Parameter Sharing: All the output is only computed using the 3x3 filter

That's why

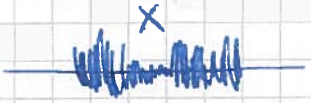
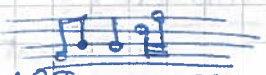
- ⇒ • CNN's are robust to translation and variance.
  • Can be trained without overfitting with fewer examples.

Course 5 - Sequence Models - Week 1

Week 1: Recurrent Neural Networks (RNNs)

- extremely effective on temporal data
- Several variants : LSTM, GRU, Bidirectional RNN, ...

Why sequence Models?

- we need sequence models when either input X or output Y or both are a sequence.
  - Speech Recognition — X:  → Y: "The quic brou fox jumped"
  - Music generation — X: ∅ → Y: 
  - DNA sequence Analysis — X: AGCCTACCCCTAGAB → Y: AGCTACCCCTAGAB
  - Machine translation
  - Video activity recognition — X: → Y: Running
  - Named Entity recognition

Notation : Example : Named Entity Recognition

X : Harry Potter and Hermione Granger invented a new spell.

	$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	.....	$x^{<t>}$	.....			$x^{<9>}$
y :	1	1	0	1	1	0	0	0	0
	$y^{<1>}$	$y^{<2>}$	.....		$y^{<t>}$	—	—		$y^{<9>}$

$x^{<t>}$: t-th element in sequence x

$x^{(i)<t>}$: t-th element in i-th training example

$y^{(i)<t>}$: t-th element in i-th output sequence

T_x : length of sequence x

T_y : length of sequence y

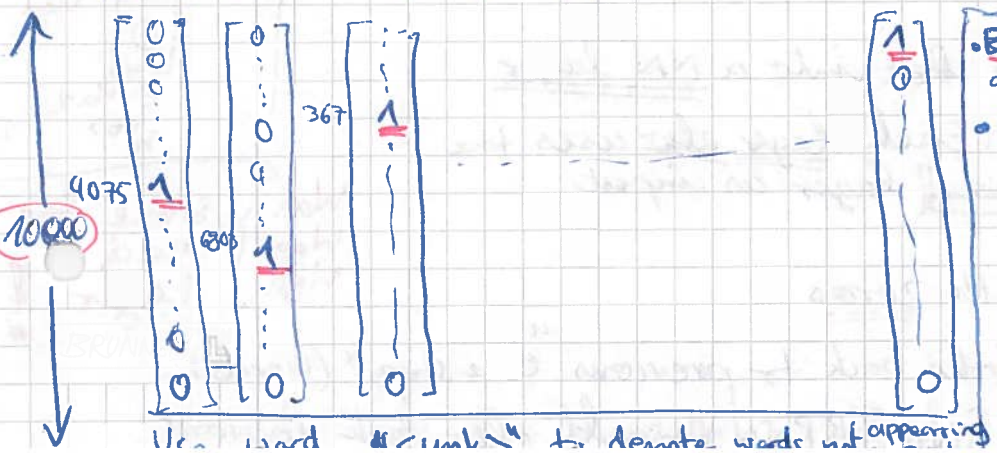
Representing Words in Sentence :

- First we need a Vocabulary : (Dictionary)

a	1
aaron	2
⋮	⋮
and	367
⋮	⋮
harry	4075
⋮	⋮
potter	6803
⋮	⋮
zulu	10000

- now use One Hot representation to represent words in the sentence

X : Harry Potter and Hermione Granger invented a new spell.



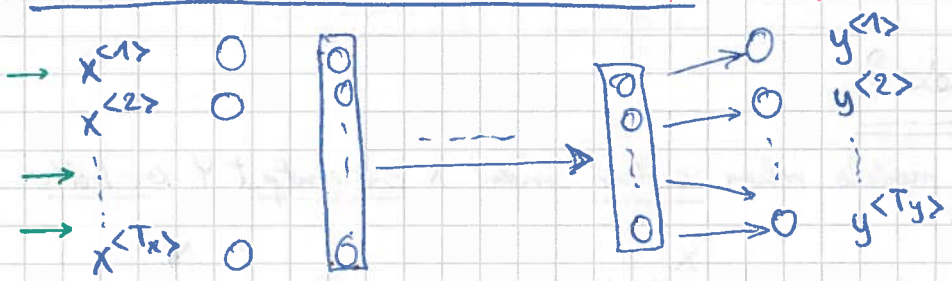
- Each $x^{<t>}$ will be a one hot vector
- X → Y maps the sequence X to target output Y

Dictionary size of 10000 is rather small; 30000 or 50k are common in large companies working on NLP

We need "<unk>" to denote words not appearing

How to learn the mapping $X \to Y$?

Why not a standard NN? ✗



Problems:

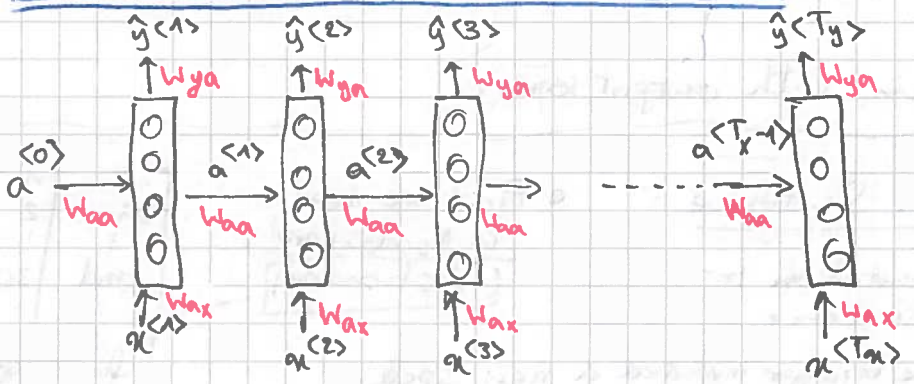
- Input X and outputs Y can have different lengths in different examples.
  - ↳ One solution could be defining a Max_Length for sentences and zero padding inputs. But this wouldn't be nice.
- ! Doesn't share features learned across different positions of text.
  - ↳ Very similar to ConvNets: You want things learnt in one part of the image to generalize to other parts as well!

e.g. Harry at position 1 and Harry at position 200 and Harry at position 352 are all named entity!

- Using a better representation reduces the number of params in the model:
  - ↳ with a dictionary size of 10k, each $x^{<t>}$ is a 10000 dimentional vector!
  - ↳ input size: Max_Length x 10000! ⟹ Huge number of params in the first layer.

Recurrent Neural Network:

Assume $T_x = T_y$



- Each "time step" is fed into a NN layer to predict the output. Each layer also uses the "activation from previous" layer as input
- $a^{<0>}$ is initialized with zeros
- This network only looks back to previous "time steps" (words).

W_{ax}, W_{aa}, W_{ya} } Same for each layer!

108) An Bidirectional RNN (BRNN) would also look forward.

W_{ax}: Parameters (weights) multiplied with an x quantity to produce an a

W_{ya}: Weights multiplied with an a quantity to produce a y quantity.

Forward Propagation: From left to right

$a^{<0>} = 0$

$$a^{<1>} = g(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a) \leftarrow$$ g: often a tanh, sometime ReLU

$$\hat{y}^{<1>} = g(W_{ya} a^{<1>} + b_y) \leftarrow$$ g: sigmoid / softmax } depending on the output y

$$a^{<t>} = g(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y)$$

Simplified Notation:

$$a^{<t>} = g(\underbrace{W_{aa}}_{(100,100)} \underbrace{a^{<t-1>}}_{100} + \underbrace{W_{ax}}_{(100,10000)} \underbrace{x^{<t>}}_{10000} + b_a) \rightsquigarrow$$

$$\boxed{a^{<t>} = g(W_a [a^{<t-1>}, x^{<t>}] + b_a)}$$

$$\underbrace{W_a}_{(100,10100)} = \left[\underbrace{W_{aa}}_{100} \vdots \underbrace{W_{ax}}_{10000}\right] \updownarrow 100$$

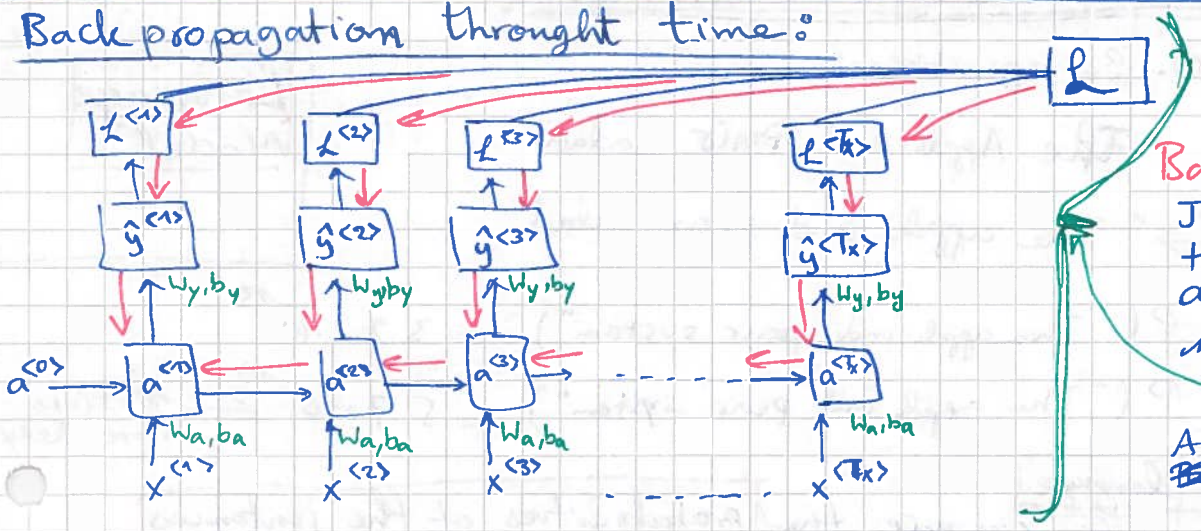
$$\underbrace{[a^{<t-1>}, x^{<t>}]}_{(10100,1)} = \begin{bmatrix} a^{<t-1>} \\ \hline x^{<t>} \end{bmatrix} \begin{matrix} \updownarrow 100 \\ \updownarrow 10000 \end{matrix}$$

$$[W_{aa} \vdots W_{ax}] \begin{bmatrix} a^{<t-1>} \\ \hline x^{<t>} \end{bmatrix} = W_{aa} a^{<t-1>} + W_{ax} x^{<t>}$$

$$\hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y)$$

$$\boxed{\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)}$$

Back propagation throught time:



Back prop.: ←

Just computing the derivatives as we go backward in the computation graph.

As we had it in all other NNs.

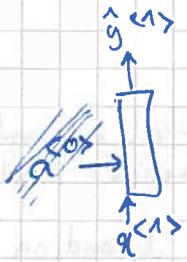
$$\mathcal{L}^{<t>} = (\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>})$$

$\mathcal{L}^{<t>}$: Element-wise loss (for the individual ...)

$$\mathcal{L}(\hat{y}, y) = \sum^{T_x} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

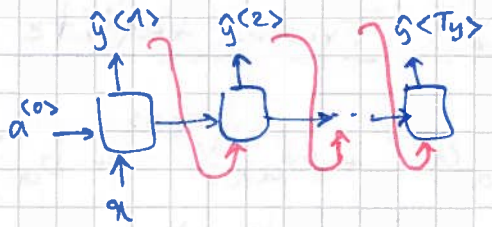
Different types of RNNs: what if $T_x \neq T_y$?

"The unreasonable effectiveness of RNNs" ← Blog Post. Andrej Karpathy



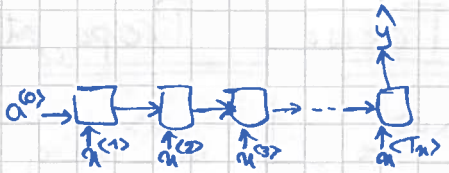
One-to-one

- Just standard generic NN.
- No need for RNN.
- We can get rid of $a^{<0>}$



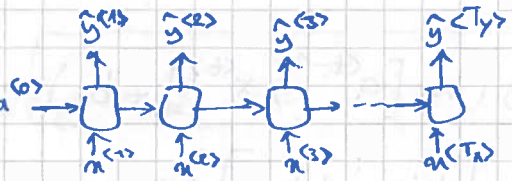
One to many

- Music generation
- Sequence generation



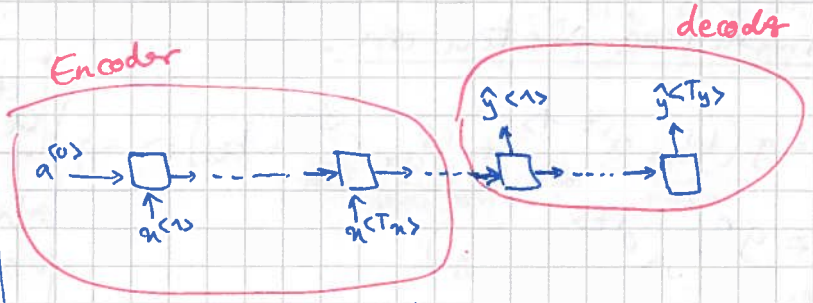
many-to-one

- Sentiment analysis



many to many

- $T_x = T_y$
- named entity recognition



many to many

- $T_x \neq T_y$
- machine translation

- These architectures capture many kinds of applications.
- There is one kind of architecture that is not captured. (Week 4)

Language Model and Sequence Generation:

Speech Recognition:

"The Apple and Pair salad"

outputs vs. "The apple and pear salad"

Speech Rec. system becomes

$P(\text{"The apple and pair system"}) = 3.2 \times 10^{-13}$

$P(\text{"The apple and pear system"}) = 5.7 \times 10^{-10}$ ← 10^3 times more likely

↳ uses a language model to compute the probabilities of the sentences

$P(y^{<1>}, y^{<2>}, \dots y^{<T_y>})$ → $P(\text{sentence})$

↳ sequence of words

- A foundamental component of Speech Recognition and machine translation systems.

Language Model

Language Modelling with RNNs:

Training Set: large $\boxed{\text{corpus}}$ of english text.

Cats average 15 hours of sleep a day.

$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ .....

Tokenize using a dictionary and one hot vectors for each word.

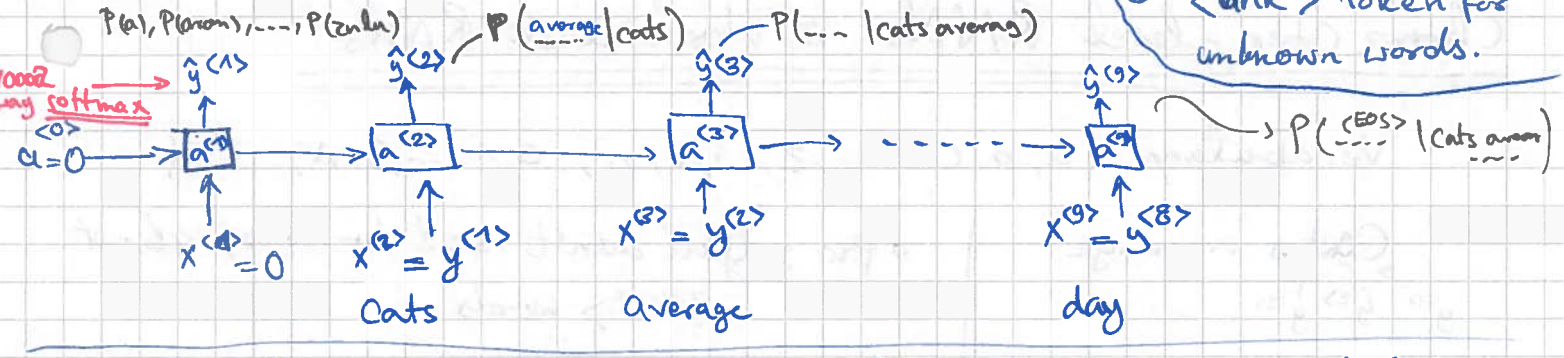
- inputs $x^{<t>}$ will be set to $y^{<t-1>}$
- language model works on the output actually (give probability of the output)

- An extra token for <EOS> (End of sentence)
- Here we ignore punctuation. Otherwise we needed also tokens for punctuation
- <unk> token for unknown words.

- Assume size of dict is 10002

{a, aaron, ..., cats, ..., zulu, <EOS>, <UKN>}

P(a), P(aaron), ..., P(zulu)



- Each step in the RNN will look at the pre-seeding words, to compute what is the distribution over the next word.
- So the RNN learns to predict one word at a time going from left to right.

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$ ← Softmax loss function.

$$\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

- If we train this RNN on a sufficiently large training set, it will be able to "given any initial set of words, such as "Cats average 15" it can predict what is the chance of a next word"

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = \underbrace{P(y^{<1>})}_{\text{first softmax}} \underbrace{P(y^{<2>} | y^{<1>})}_{\text{second softmax}} \underbrace{P(y^{<3>} | y^{<1>} y^{<2>})}_{\text{third softmax}}$$

Sampling Novel Sequences:

- After training a sequence model, let it generate samples to see how it has been trained, and what it has learned.
- The $\boxed{\text{first softmax}}$ gives you the probability of every word in the dictionary: P(a), P(aaron), ..., P(zulu), P(UKN)
- use something like $\boxed{\text{np.random.choice()}}$ to randomly choose one of them.
- Feed it to next layer as $\boxed{x^{\langle 2\rangle} = \hat{y}^{\langle 1\rangle}}$.
- Sample from next softmax distribution $\rightarrow \hat{y}^{\langle 2\rangle}$
- Repeat until you see <EOS> or have reached a certain number of words

Character-level RNNs vs. Word-level RNNs

Vocabulary: {a, b, c, ..., z, ␣, ., ,, ;, 0, 1, ..., 9, A, ... Z}

Cats average

$y^{\langle 1\rangle}$ $y^{\langle 2\rangle}$ $y^{\langle 3\rangle}$ ....

- Pro: You don't need to worry about <UKN> words.
- Con: Much longer sequences. Because it is not as good as word-level language models at capturing dependencies between earlier parts of the sentence and later parts of sentence.
- more computationally expensive to train

- The trend is word-level language models are used much more often.

Trained on Shakespear.

"The mortal moon hath her eclipse in Love"

Vanishing Gradients with RNNs:

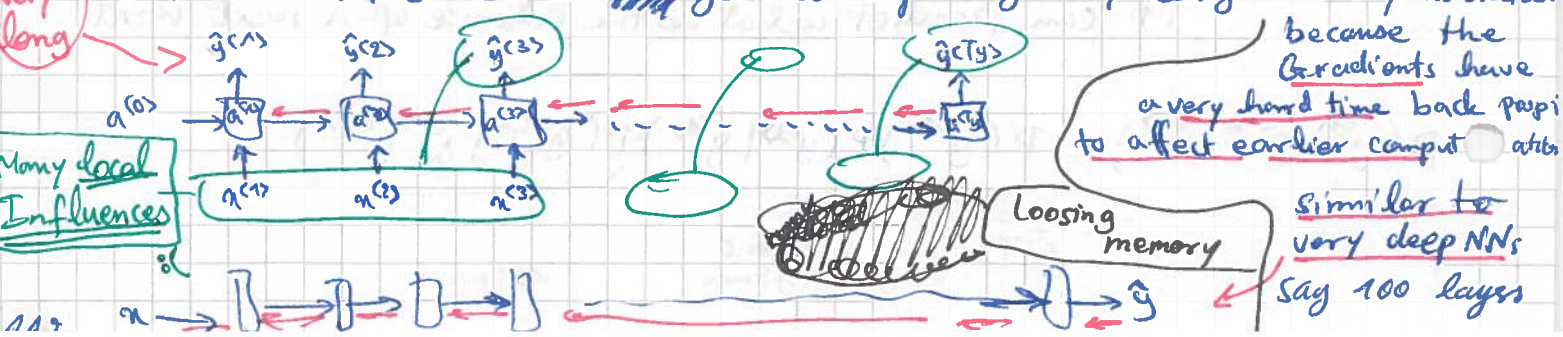
- Basic RNN algorithms have the problem of vanishing Gradients.

"The cat, which ....., was full."

The cats, which ....., were full."

- Language can have very longterm dependencies.
- Basic RNNs are not good at capturing very longterm dependencies.

RNN very long



because the Gradients have a very hard time back propi to affect earlier computations

similar to very deep NNs say 100 layers

- If we don't address the vanishing Gradient problem, RNNs tend not to be good at capturing longterm dependencies.
- RNNs can also ~~sy~~ suffer from exploding gradients! But it is easy to spot (NaNs) $\rightsquigarrow$ Gradient Clipping (rescale the gradients)

- An RNN processing data over a 1000 time steps or over 10000 time steps is in fact a very long NN (1000 layers or 10000 layers)

!! $\Rightarrow$ Vanishing or exploding gradients happen easily.

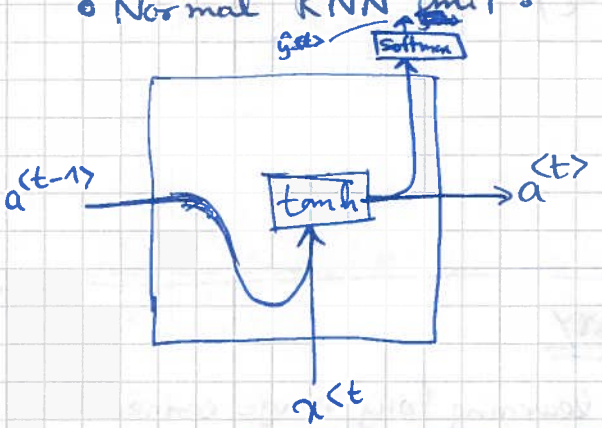
$\Rightarrow$ GRUs to solve the problem of vanishing gradients.

Gated Recurrent Unit

! • Better capturing of longterm relations and solving the vanishing Gradient problem

- Normal RNN unit:

$$a^{\langle t\rangle} = g(W_a[a^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_a)$$



~~The cat, which~~

The cat, which already ate....., was full.

$c^{\langle t\rangle} = 1$ - - -

- Because when $\Gamma_u = 0$, the value of $c^{<t>}$ is maintained ($c^{<t>} = c^{<t-1>}$) the $\boxed{\text{GRN}}$ ~~can~~ very good ~~pas~~ maintain the longterm dependencies.
- Exactly also because of maintaing the value of $c^{<t>}$ over long range, it does not suffer much from vanishing gradient.

across many many timesteps

- $c^{<t>}, a^{<t>}, \tilde{c}^{<t>}, \Gamma_u \rightsquigarrow$ all vectors with the same dimension, e.g. (100,1)
- Γ: you can think of as a vector of 0's and 1's that tell you ~~with~~ which memory cells should be updated.

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

: For example 1 bit to memorize if "cat" or "cats"
1 bit to remeber if "food"
....

Full GRU:

- almost the same.
- has an extra gate Γ_r (for relevance)
- it tells how relevant is $c^{<t-1>}$ for computing the $\tilde{c}^{<t>}$

$$\tilde{c}^{<t>} = \tanh\left(W_c\left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$\Gamma_r = \sigma\left(W_r\left[c^{<t-1>}, x^{<t>}\right] + b_r\right)$$
$$\Gamma_u = \sigma\left(W_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$$

LSTM: Long-Short term memory:

- even more powerful than the GRUs in learning long range connections in a sequence.

GRU

$$\tilde{c}^{<t>} = \tanh\left(W_c\left[\Gamma_r * c^{<t-1>}, x^{<t>}\right] + b_c\right)$$
$$\Gamma_u = \sigma\left(W_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$$
$$\Gamma_r = \sigma\left(W_r\left[c^{<t-1>}, x^{<t>}\right] + b_r\right)$$
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u)\, c^{<t-1>}$$
$$a^{<t>} = c^{<t>}$$

LSTM (more general than GRU)

$$\tilde{c}^{<t>} = \tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right)$$

update $\Gamma_u = \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right)$

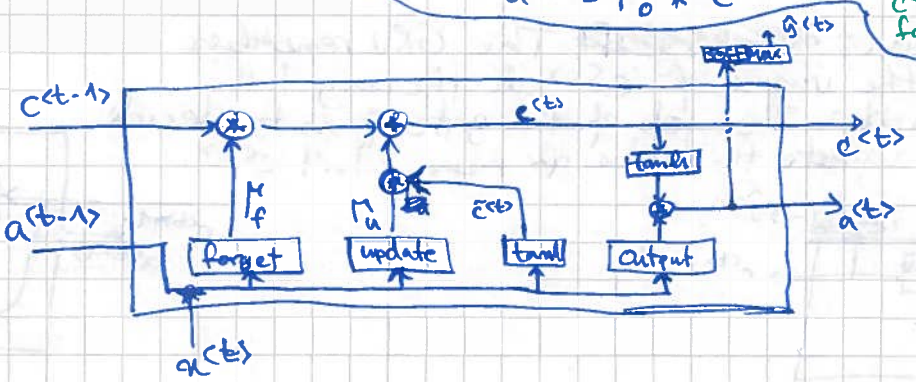
forget $\Gamma_f = \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right)$

output $\Gamma_o = \sigma\left(W_o\left[a^{<t-1>}, x^{<t>}\right] + b_o\right)$

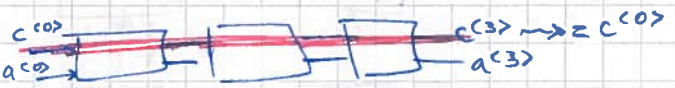
three gates

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$

you can now add $c^{<t-1>}$ to $\tilde{c}^{<t>}$ or forget it.



← • If we connect multiple LSTM units together after eachother, it is relatively easy to have the value of $c^{<0>}$ passed alle the way to right up to $c^{<T_y>}$ unchanged, as long as we choose right values for Γ_f (forget) and Γ_u (update)



Peephole connections

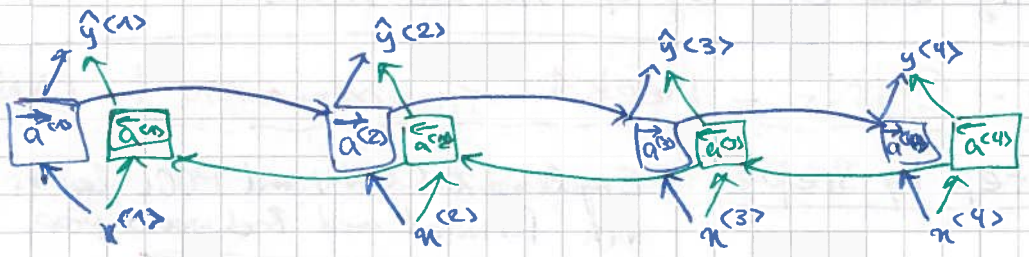
• A common variation is make Γ gate values also depend on $c^{<t-1>}$ and not only on $a^{<t-1>}$ and $x^{<t>}$. It is called "peephole connection"

• Which one is better? GRU or LSTM? when to use one?

↳ • no consensus on that
  • GRUs are even introduced later (2014) than LSTM (1997)
  • GRUs are simpler. Easier to build larger networks.
  • LSTMs are more powerful and more generalizeable.
  • LSTM has more history of working systems. Many people first use LSTMs. But GRUs are gaining popularity.

Bidirectional RNNs:

• Take information at both earlier and later sequence steps.



Asyclic Graph

→ forward recurrent units
← backward recurrent units

} It is all Forward Prop!
The Forward prop has part of the computation from left to right and part of the computation from right to left.

$$\hat{y}^{<t>} = g\left(W_y\left[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}\right] + b_y\right)$$

• At any given time $<t>$ information from the past ($\vec{a}$) and information from future ($\overleftarrow{a}$) are taken into account.

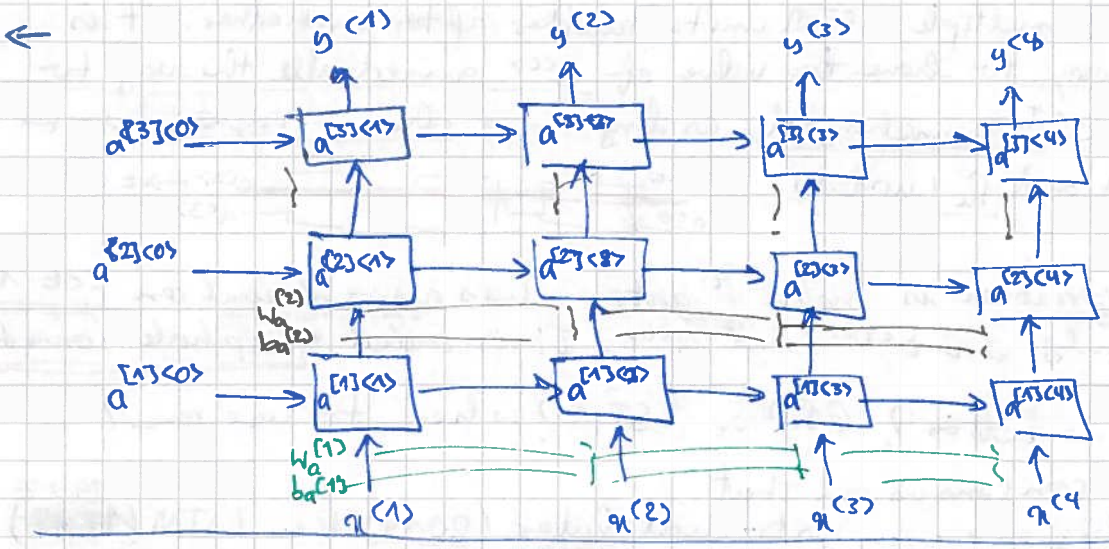
• BRNNs work also with GRU's and LSTM.

• The problem with it is, it $\boxed{\text{needs the entire sequence of}}$ the data before making predictions.

Deep RNNs:

• Stack multiple layers of RNN on top of eachother

← • It is like rolling a deep NN in time →



$a^{[l]<t>}$

↳ activation of layer l at time t

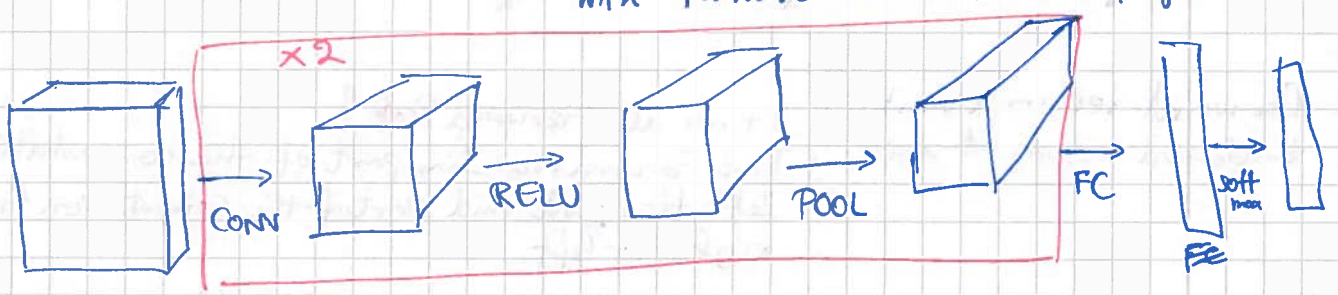
$$a^{[2]<3>} = g\left(W_a^{[2]}\left[a^{[2]<2>}, a^{[1]<3>}\right] + b_a^{[2]}\right)$$

- Typically for RNN having 3 layers is already a lot. Because, because of the temporal relationship, the networks becomes huge. You don't see 100s of temporal layers stacked on top of eachother.
- Sometimes you see deep networks that are not connected temporally on top of stacked RNNs.

Course 4 – CNNs – Week 1 – Programming 1 – CNN Basics.

CNNs Step by Step:

- Implement CONV and POOL layers in numpy with forward and backward prop.



- A convolution layer transforms an input volume into an output volume of different size.
  - volume convolution
  - padding, stride
  - pooling
- Zero padding; Same Convolution;
- POOLing reduces the height and width of the input
  - ↳ helps reduce computation
  - ↳ helps make feature detectors more invariant to its position in input

Back Prop in ConvNets:

$\boxed{dA}$ ← ↳ Z at the h-th row and w-th colum.

dA w.r.t. cost for a certain filter W_c and a given training example:

$$dA += \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} W_c * dZ_{hw}$$

W_c: filter

dZ_{hw}: scalar corresponding to gradient at the conv. layer & to that the conv. layer

← in code: da_prev_pad[vert_start:vert_end, horiz_start:horiz_end, :] +=

dW: dW_c: derivative of one filter w.r.t. loss

$$dW_c \mathrel{+}= \sum_{h=0}^{n_H} \sum_{w=0}^{n_W} a_{slice} \times dZ_{hw}$$

Code:

dW[:,:,:,c] += a_slice * dZ[i,h,w,c]

- a_{slice} corresponds to the slice that was used to generate the activation Z_{ij} and gives us the gradient of W w.r.t. that slice. We just add up all such gradients to get dW

W[:,:,:,c] * dZ[i,h,w,c]

db:
- db w.r.t. to cost for a certain filter W_c: $db = \sum_h \sum_w dZ_{hw}$
- similar to normal NNs, we sum up dZ.

db[:,:,:,c] += dZ[i,h,w,c]

POOLing:
- pooling layer itself does not have any params for backprop to update. But we still need to backprop the gradients through the pooling layer to compute grads of the layer before it.

Max

- We need to keep track of the position of the max. Because it is the input value that ultimately influenced the output and therefore the cost. Backprop computes the gradients w.r.t. cost, so anything that influences the ultimate cost should have a non-zero gradient. So, the backprop will "propagate" the gradient back to this particular input value that had influenced the cost.

Mask: $X = \begin{bmatrix} 1 & 3 \\ 4 & 2 \end{bmatrix} \longrightarrow M = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}$

multiply the mask with the slice from dA to get → the corresponding slice in dA_prev

Avg: In Avg pooling every element of the input window had equal influence on the cost. For example for 2x2 Avg pool filter the mask for backward pass will look like: Avgmask

$dZ = 1 \longrightarrow dZ = \begin{bmatrix} 1/4 & 1/4 \\ 1/4 & 1/4 \end{bmatrix}$ ← Avg Mask

Course 4 - CNNs - Week 1 - Programming 2 - CNN Application.

Tensorflow:

init
- create placeholders
- Initialize weight (filters) W1, W2, using Xavier. We don't need to init the bias, tensorflow does it. TF also inits the fully connected layers.

forward prop ⇒ Z3
- tf.nn.conv2d(X, W1, strides=[1, s, s, 1], padding="SAME")
  tf.nn.max_pool(. . .)
  tf.nn.relu(. . .)
  tf.contrib.layers.flatten(
  tf.contrib.layers.fully_connected(F, num_outputs)

weights get automatically initialized.

CONV2D → RELU → MAXPOOL → CONV2D → RELU → MAXPOOL → FLATTEN → FC

- In tf, softmax and the cost function are lumped together.

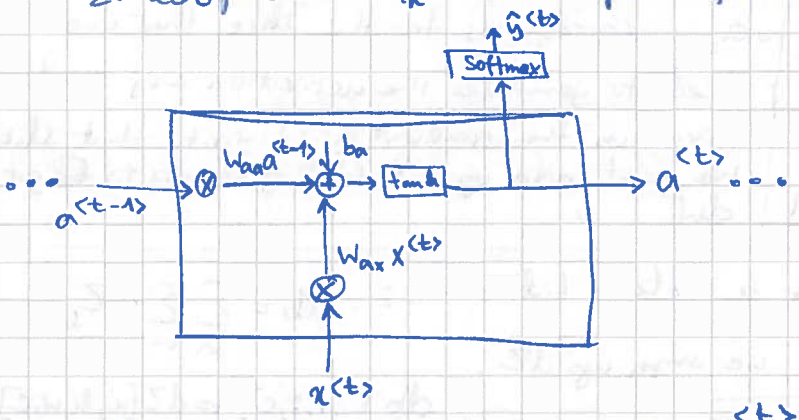
Cost
- tf.nn.softmax_cross_entropy_with_logits(logits=Z3, labels=Y)
- tf.reduce_mean

tf.sess.Run for each mini batch.

Optimizer: tf.train.AdamOptimizer(. . .)

Course 5 – RNNs – Week 1 – Programming 1 – RNN Basics

1. Implement the calculations needed for one time-step of RNN
2. loop over T_x time-steps to process the inputs one at a time.



$$a^{<t>} = \tanh(W_{ax}x^{<t>} + W_{aa}a^{<t-1>} + b_a)$$

$$\hat{y}^{<t>} = \text{softmax}(W_{ya}a^{<t>} + b_y)$$

Dimensions

- Vectorized over m examples: $x^{<t>} \rightsquigarrow (n_x, m)$; $a^{<t>} \rightsquigarrow (n_a, m)$ Vectorized over m examples.
- An RNN is a repetition of the above cell. If a training example has 10 time-steps (for example a sentence with 10 words) we copy this RNN cell 10 times, with the same $W_{ax}, W_{aa}, W_{ya}, b_a, b_y$.

LSTM:

$$\Gamma_f^{<t>} = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$$

$$\hat{y}^{<t>} = \text{softmax}(W_y a^{<t>} + b_y)$$

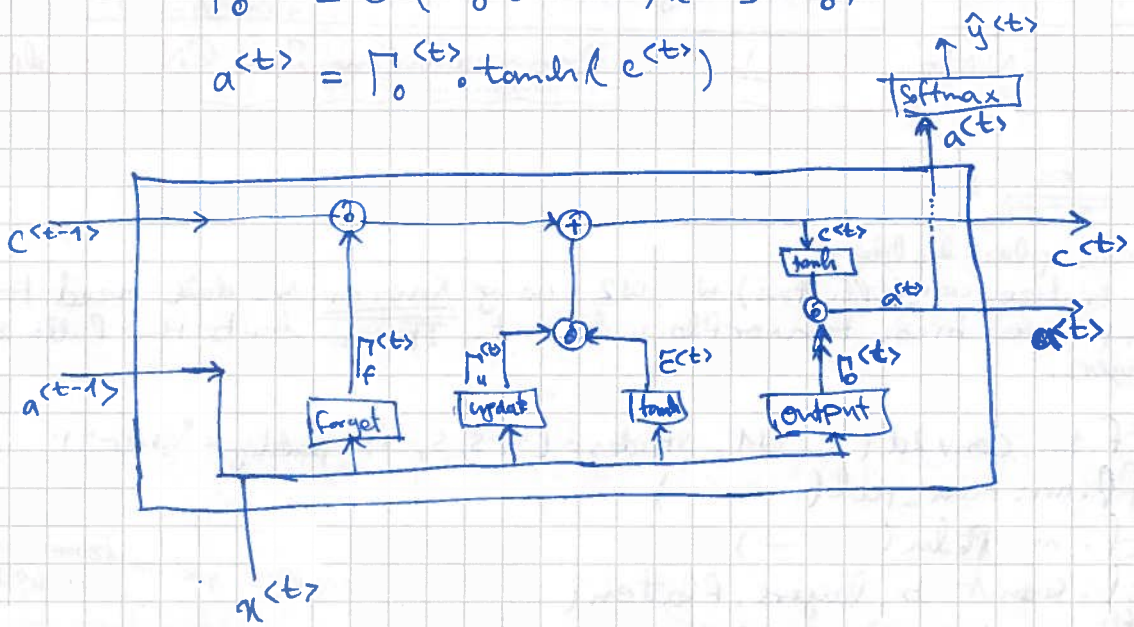
$$\Gamma_u^{<t>} = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$$

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

$$c^{<t>} = \Gamma_f \circ c^{<t-1>} + \Gamma_u \circ \tilde{c}^{<t>}$$

$$\Gamma_o^{<t>} = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$$

$$a^{<t>} = \Gamma_o^{<t>} \circ \tanh(c^{<t>})$$



Back prop in RNNs: Very complicated. See the Jupyter notebook.

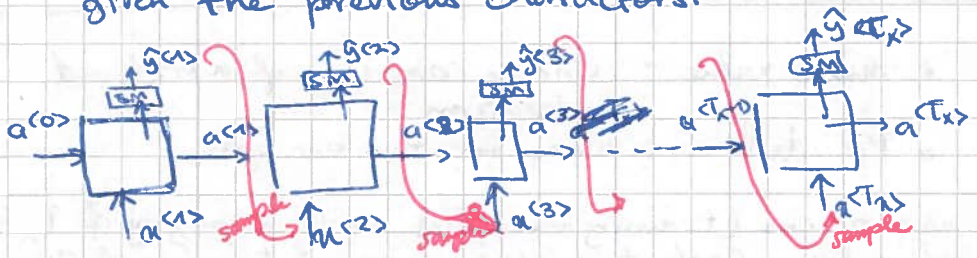
Course 5 – RNN – Week 1 – Programing 2 – Character level language model.

- For dinosaurus: Learn name patterns from a dataset of dinosaurus names and generate new names.
- Learn Objectives:
  - Store text data for processing using RNN
  - Synthesize data by sampling predictions at each →

← time-step and passing it to the next RNN-cell unit.

- How to build a character level text generation RNN.
- Why Gradient clipping is important.

- At each step the RNN tries to predict what is the next character given the previous characters.



dataset: $X = (x^{<1>}, x^{<2>}, \dots, x^{<T_x>})$ is a list of characters in the training set.
while $Y = (y^{<1>}, y^{<2>}, \dots, y^{<T_x>})$ is such that at every timestep t we have $y^{<t>} = x^{<t+1>}$

Building Blocks of the model.

Gradient Clipping: Avoid gradient exploding. np.clip(array, min, max)

Sampling: We assume model is trained. We give in $x^{<1>} = \vec{0}$ and leave the network sample one char at a time.

Step1: pass the network the first dummy input $x^{<1>} = \vec{0}$

Step2: Run one step of forward propagation to get $a^{<1>}$ and $\hat{y}^{<1>}$
Note that $\hat{y}^{<1>}$ is a softmax probability vector, $\hat{y}_i^{<t>}$ represents the probability that the character indexed by i is the next character.

Step3: Carry out sampling: Pick the next character's index according to the probability distribution specified by $\hat{y}^{<t>}$. That is, if $\hat{y}_i^{<t>} = 0.16$ we pick the character at index i with 16% probability.

np.random.choice() ← gets as input an array and a probability distribution of choosing each element of that array

Step 4: Overwrite $x^{<t+1>}$ with the character chosen at step 3.
Repeat until you see an <EOS>

Building the language Model:

- Stochastic GD. (One training example at a time): Steps:

1. Forward propagate through RNN to compute the loss.
2. Backward propagate through time to compute gradients of loss w.r.t to parameters.
3. clip the gradients if necessary.
4. update parameters using GD. Params: $W_{aa}, W_{ax}, W_{ya}, b_y, b_a$

- Every 100 steps of the stochastic GD, we sample 10 randomly chosen names to see how the algorithm is doing.

- For training: Each training Example:

list of $x^{<t>}$ → one hot X: list of integers where each integer is a number that maps to a character in the vocabulary

Y: Exactly the same as X, but shifted one index to the left.

Shakespeare Poem Generator:

- LSTM instead of basic RNN to capture longer-range dependencies
- The model is deeper, stacked LSTM model (2 layers)
- Using Keras instead of Python to simplify the code.

Music Generation:

- music value: a note comprizing of a pitch and a duration
- 78 different values for this example.

X: $(m, T_x, 78)$: m music pieces (training examples). Each comprizing of $T_x = 30$ musical values. Each time-step, a one-hot vector of 78 possible values.

Y: same as X, but shifted one step to the left.

- We are interested in a network that uses the previous values to predict next value: That is to predict $y^{<t>}$ given $x^{<1>}, \dots, x^{<t>}$

Keras: Reshape(), LSTM_cell(), Dense()

It is simply unbelievable!

Course 4 - CNNs - Week 2 - Deep convolutional Models: Case Studies

1h 39m 1/1
3 h program

Week 2

Why case studies?

- Many networks are basically putting together the basic building block of CNNs.
- Gain intuition by seeing how others do it. (Like learning to code by looking at code of others)
- many architectures from one task apply well to other problems.

- Classic Networks:
  - LeNet-5 → 1998s
  - AlexNet
  - VGG
- ResNet (152 layers) → interesting tricks
- Inception

⇒ Much better intuitions in building CNNs

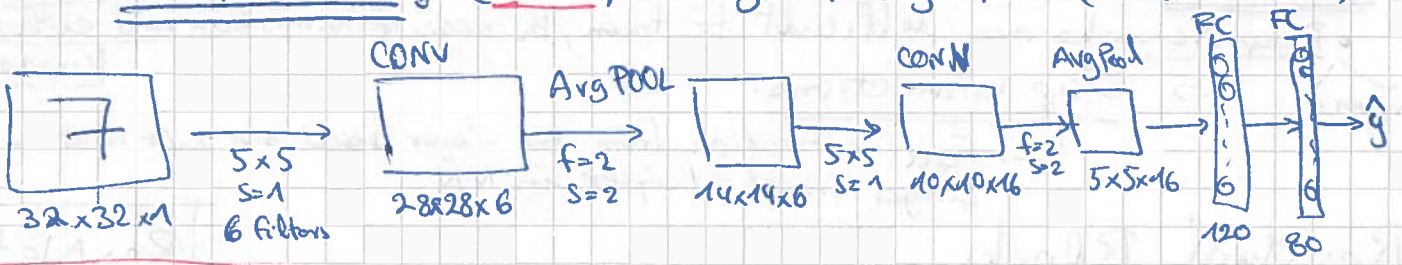
- Many interesting ideas to cross-fertelize to other areas other than computer vision

---

Classic Networks:

→

LeNet-5 : (1998) Digit Recognition (handwritten)

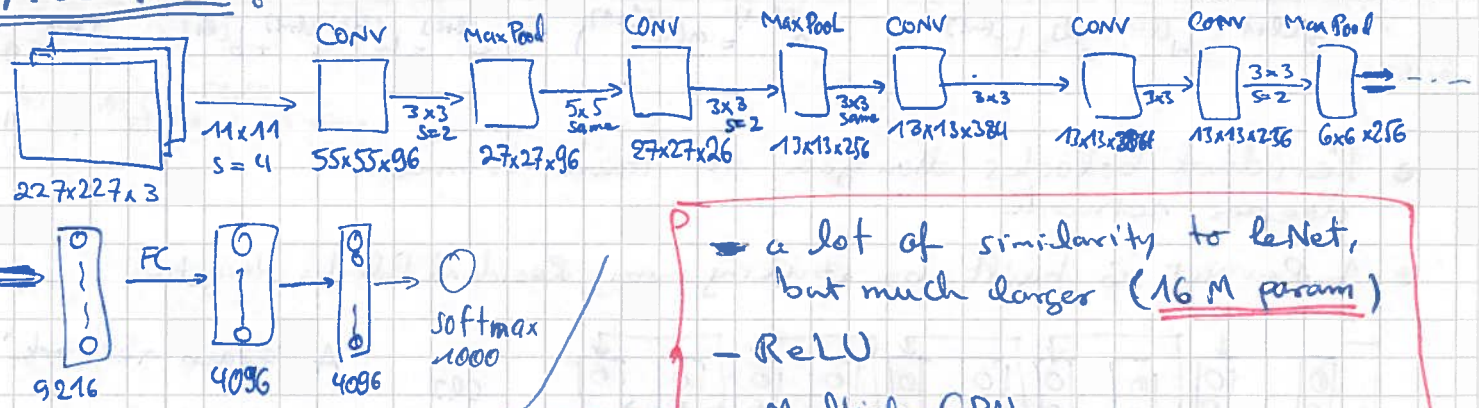


- 60k param
- $n_H, n_W \downarrow \quad n_C \uparrow$ ← !
- conv – pool – conv – pool – fc – fc – output
- sigmoid / tanh.

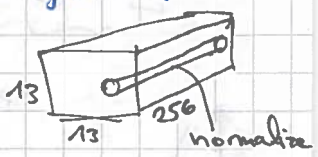
LeNet5

AlexNet : (2012)

Alex-Net



- a lot of similarity to LeNet, but much larger (16 M param)
- ReLU
- Multiple GPUs
- local Response Normalization (LRN) (not used much)



- Milestone paper for deeplearning in CV

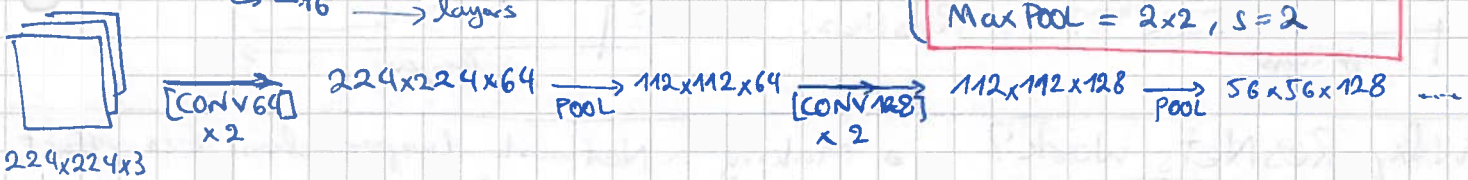
VGG (2015) :

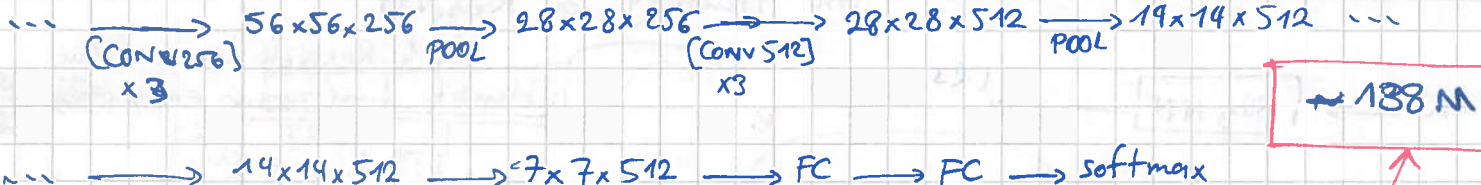
- AlexNet had a very complex architecture with a lot of hyperparameters

⇒ VGG, let's make it simpler :

↳ –16 → layers

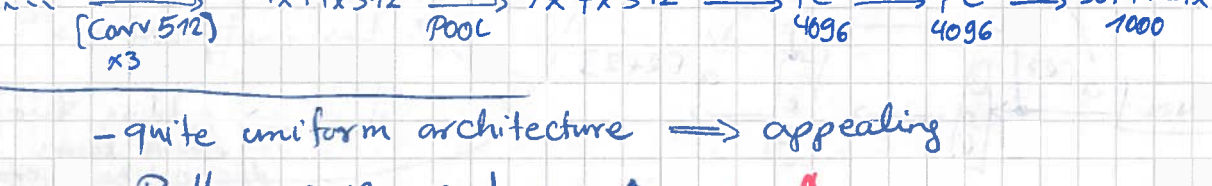
CONV = 3x3, s=1, same
MaxPOOL = 2x2, s=2





≈ 138 M Param

huge even with today's standard.



- quite uniform architecture ⟹ appealing
- Pattern : $n_H, n_W \downarrow, n_C \uparrow$ ← !
  ↳ very systematic.

VGG16 →

ResNets:

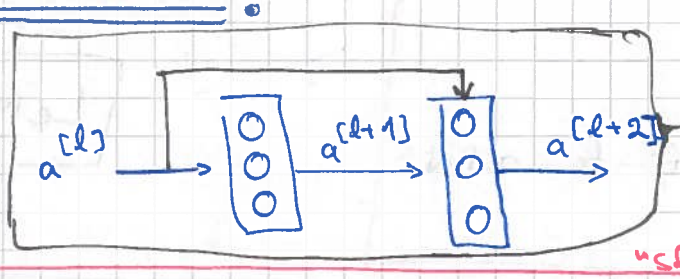
- Deep Networks are difficult to train, because of vanishing and exploding Gradients
  (very very)

  $\Rightarrow$ Skip Connections.

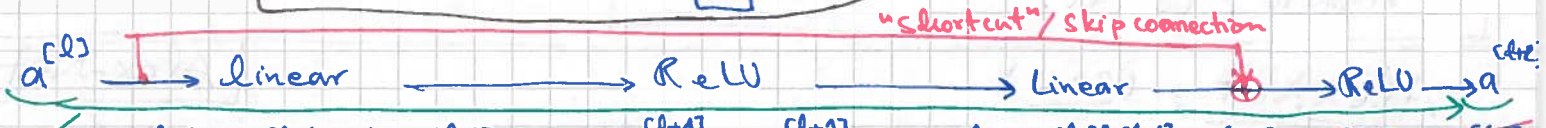
  ↳ Feed activation from one layer suddenly into another Layer much deeper in NN.

Residual Blocks:

ResNets



→ A Residual Block.

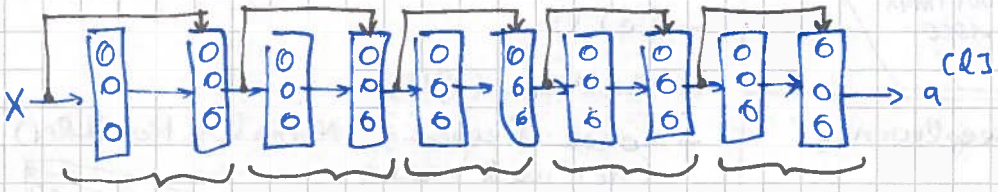


"main path"

$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$ $\quad a^{[l+1]} = g(z^{[l+1]})$ $\quad z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$ $\quad \cancel{a^{[l+2]} = g(z^{[l+2]})}$

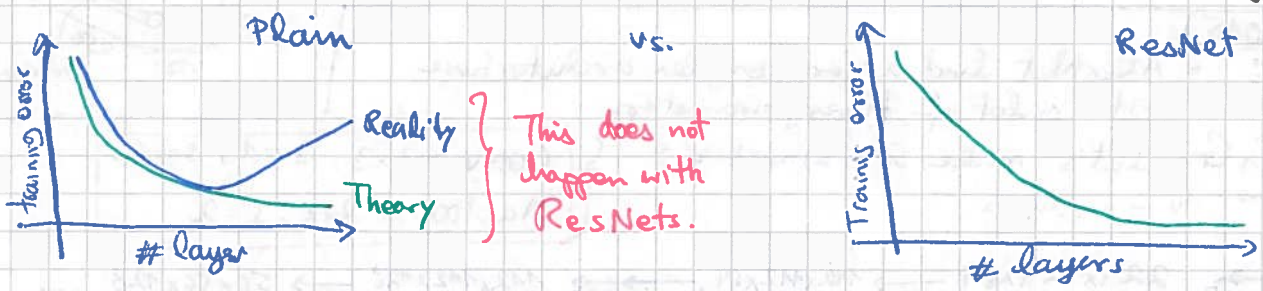
$\rightarrow a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$

- Residual bolocks allow you to train a much deeper network.
- A ResNet is built by stacking many Residual Blocks together.



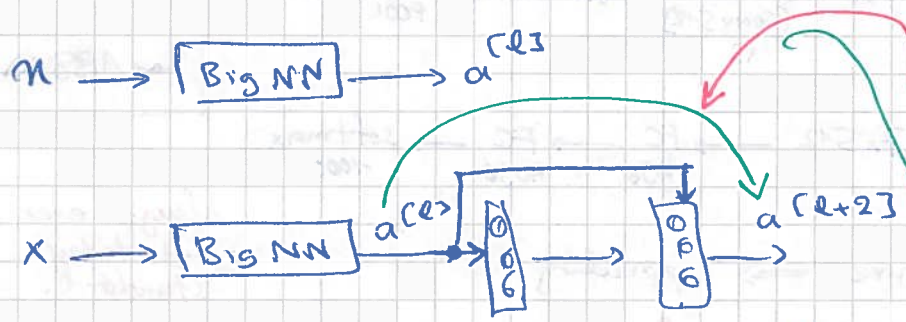
A "plain network"

A ResNet with 5 Residual Blocks.



Why ResNets Work?

- Making a Network deeper has less effect on training a ResNet.



With ResNets Learning the Identity function is easy

↳ $a^{[l+2]} = a^{[l]}$

↳ just copying $a^{[l]}$ to $a^{[l+2]}$

↳ $\Rightarrow$ Adding two extra layers does not hurt performance!

Assume ReLU activation $\Rightarrow$ all $a \geq 0$

$$a^{[l+2]} = g(z^{[l+2]} + a^{[l]}) = g(\underbrace{\cancel{W^{[l+2]}}}_{\to 0} \cancel{a^{[l+1]}} + \underbrace{\cancel{b^{[l+2]}}}_{\to 0} + a^{[l]}) \Rightarrow = g(a^{[l]}) = a^{[l]}$$

If regularization (weight decay) is applied, be careful