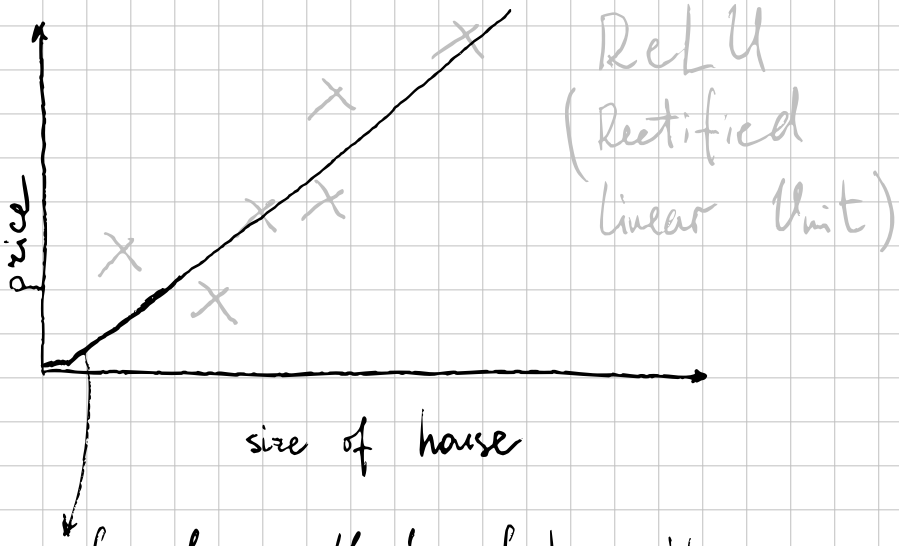
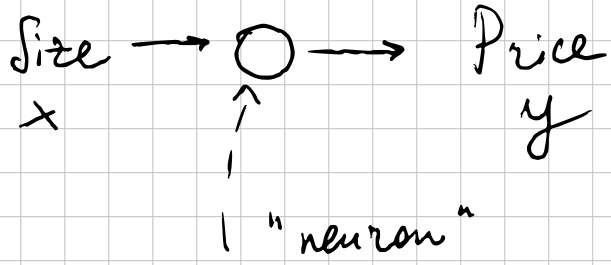
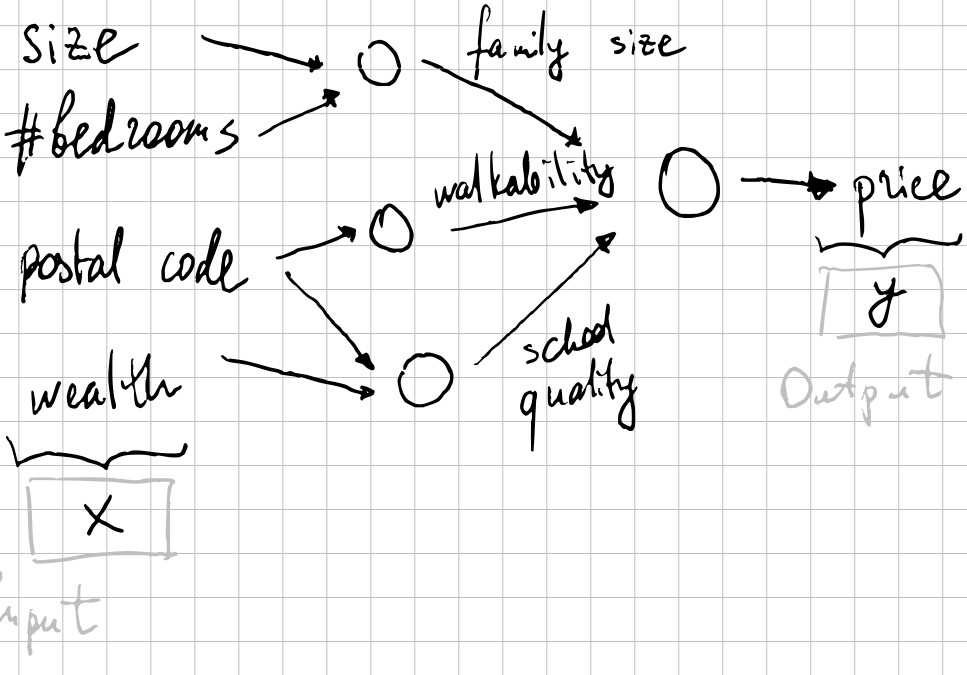
Housing Price Prediction



function that fits the price may be represented as simplest neural network



Another example is



Supervised Learning

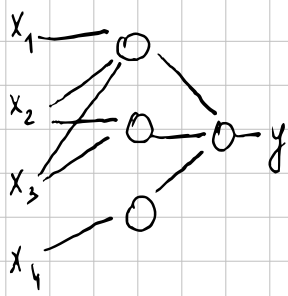
Input (x)	Output (y)	Application
Home features	Price	Real estate
Ad, user info	Click on ad? (0/1)	Online advertising
Image	Object	Photo tagging
Audio	Text transcript	Speech recognition
English	Chinese	Machine translation
Image, Radar info	Position of other cars	Autonomous driving

For ad we may use
- Standard NN

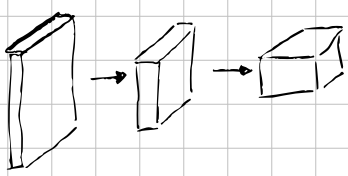
For images
- Convolutional NN (CNN)

For sequential data
- Recurrent NN (RNN)

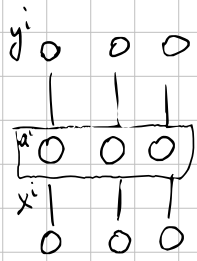
Neural Network Examples



Standard NN



Convolutional NN



Recurrent NN

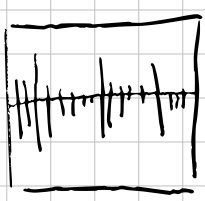
Supervised Learning

Structured Data

(databases of data)
every feature has very well-defined meaning

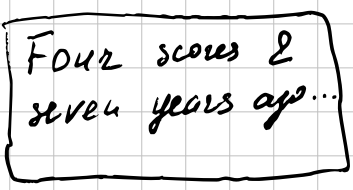
Size	#bedrooms	...	Price
100	2		1000
150	1		1200

Unstructured Data



Audio

Image

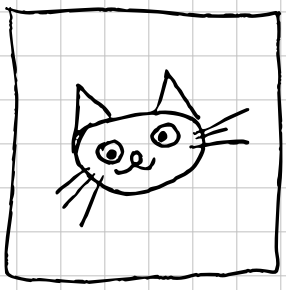


Text

Drivers of Deep Learning progress

- Data
- Computation
- Algorithms

Binary classification



→ 1 (cat) vs 0 (non-cat)

y

In binary classification our goal is to learn a classifier that can input an image, represented by a feature vector X, and predict whether the corresponded label y is 0 or 1.

Notation

- (x, y) where $x \in \mathbb{R}^{n_x}$ feature vector
  $y \in \{0, 1\}$ label
- m training examples: $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}) \dots (x^{(m)}, y^{(m)})$,
- m_{train} training samples
- m_{test} test examples
- $X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ | & | & & | \end{bmatrix}$ (n_x ↕, m ↔)

  $X \in \mathbb{R}^{n_x \cdot m}$
- $y = [y^{(1)}, y^{(2)} \dots y^{(m)}]$

  $y \in \mathbb{R}^{1 \cdot m}$

Logistic Regression

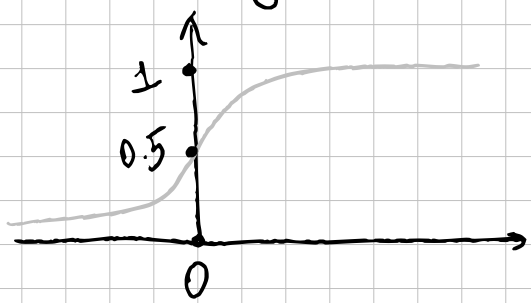
Given x, we want to predict $\hat{y}$ which is an estimate of y,

i.e. $\hat{y} = P(y=1 \mid x)$

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

Parameters: $w \in \mathbb{R}^{n_x}$, $b \in \mathbb{R}$

Output: $\hat{y} = \sigma(w^T x + b)$



$$\sigma(z) = \frac{1}{1+e^{-z}}$$

- If z large $\quad \sigma(z) \approx \frac{1}{1+0} \approx 1$

If z small or large neg. number

$$\sigma(z) \approx \frac{1}{1+e^{-z}} = \frac{1}{1+\text{Big}} \approx 0$$

Logistic Regression Cost Function

We need to find

$\hat{y} = \sigma(w^T x + b)$,

where $\sigma(z) = \frac{1}{1+e^{-z}}$

Given $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$,

want $\hat{y}^{(i)} \approx y^{(i)}$

Loss (error) function:

$$\mathcal{L}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Example

Function we need to define that shows how good predicted output ($\hat{y}$) compared to the ground truth (y)

In logistic regression usually uses:

$$\mathcal{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y) \log(1-\hat{y}))$$

If
- $y=1$ $\quad \mathcal{L}(\hat{y}, y) = -\log \hat{y}$
- $y=0$ $\quad \mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$

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

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

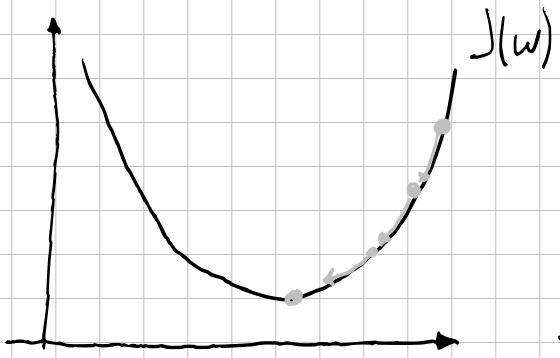
Gradient Descent

Want to find w, b that minimize $J(w, b)$

$J(w, b)$ is a multi-dimensional surface

Gradient descent starts at some initial point and takes a step towards the steepest direction

For 1-dimensional case



Repeat {

$$w := w - \alpha \frac{dJ(w)}{dw}$$

derivative

}

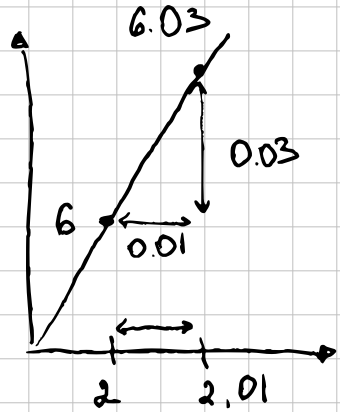
Learning rate

$$w := w - \alpha \frac{dJ(w,b)}{dw}$$

$$b := b - \alpha \frac{dJ(w,b)}{db}$$

$J(w, b)$

Derivatives



$f(a) = 3a$

$a = 2 \quad f(a) = 6$

$a = 2 + 0.01 \quad f(a) = 6.03$

Slope (derivative) of $f(a)$ at $a = 2$ is 3

Slope is defined as $\frac{\text{height}}{\text{width}}$

$a = 5 \quad f(a) = 15$

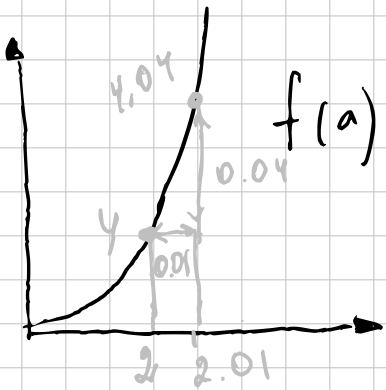
nudge $a = 5 + 0.01 \quad f(a) = 15.03$

Slope at $a = 5$ is 3

$$\frac{df(a)}{da} = 3 = \frac{d}{da} f(a)$$

Derivate is operating with infinitely small nudges

More derivatives



$f(a) = a^2$

$a = 2 \quad f(a) = 4$

$a = 2 + 0.01 \quad f(a) \Leftarrow$

$\Leftarrow \quad 4.0401 \approx 4.04$

$$\frac{d}{da} f(a) = 4 \quad \text{when } a = 2$$

$a = 5^- \qquad f(a) = 25^-$

$a = 5 + 0.001 \qquad f(a) \approx 25.010$

$$\frac{d}{da} f(a) = 10 \quad \text{when} \quad a = 5^-$$

$$\frac{d}{da} a^2 = 2a$$

Additional examples

$$f(a) = a^2 \qquad \frac{d}{da} f(a) = 2a$$

$$f(a) = a^3 \qquad \frac{d}{da} f(a) = 3a^2$$

$$f(a) = \log_e(a) = \ln(a) \qquad \frac{d}{da} f(a) = \frac{1}{x}$$

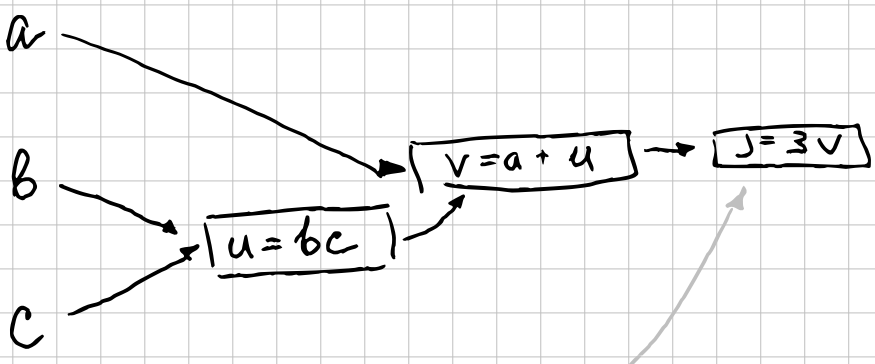
Computation graph

Let's say we want to compute $J(a,b,c) = 3(a+bc)$

We can compute this function in a few steps:

1. $u = bc$
2. $v = a + u$
3. $J = 3v$

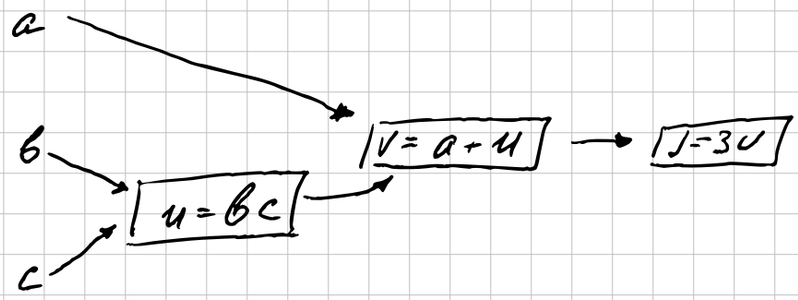
We can take this 3 steps and draw them in a computational graph



In case of Logistic regression J is a cost function

Computing derivatives with a computation graph

The computation graph



Let's compute $\frac{dJ}{dv}$?

$J = 3v$

$\frac{d}{dv} J = 3$

What is $\frac{dJ}{da}$?

$$\frac{dJ}{da} = \frac{dJ}{dv} \cdot \frac{dv}{da} = 3 \cdot 1 = 3$$

Chain rule

Final output variable is a variable that we want to optimise

$$\frac{dJ}{du} = \frac{dJ}{dv} \frac{dv}{du} = 3 \cdot 1 = 3$$

$$\frac{dJ}{db} = \frac{dJ}{dv} \frac{dv}{du} \frac{du}{db} = 3c$$

$$\frac{dJ}{dc} = \frac{dJ}{dv} \frac{dv}{du} \frac{du}{dc} = 3b$$

Back Propagation

We are trying to recognise hand-written number

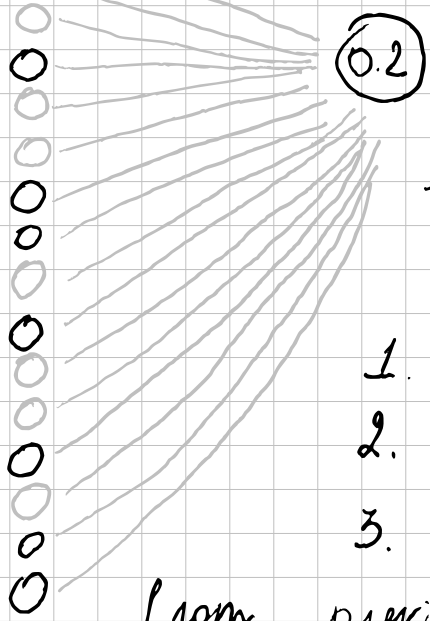
activation

bias

$$(0.2) = \sigma(w_0 a_0 + w_1 a_1 + \dots + w_{n-1} a_{n-1} + b)$$

sigmoid

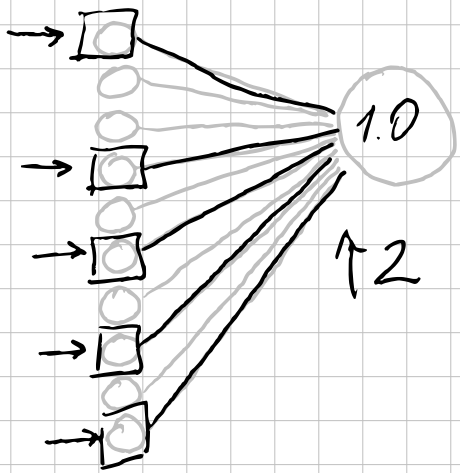
weighted sum of all activations from prev. layer



To increase the activation:

1. Increase bias (b)
2. Increase weights (w_i)
3. Change activation from previous layer (a_i)

Increasing neurons weight for neurons with bigger activation value will bigger affect the resulting activation



neurons in □ have bigger a_i

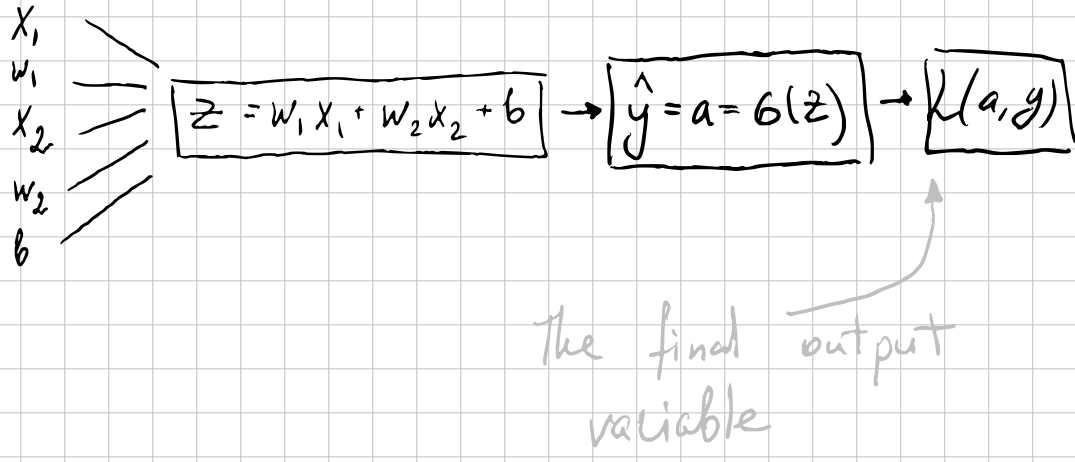
- Increase w_i in proportion to a_i
- Change a_i in proportion to w_i

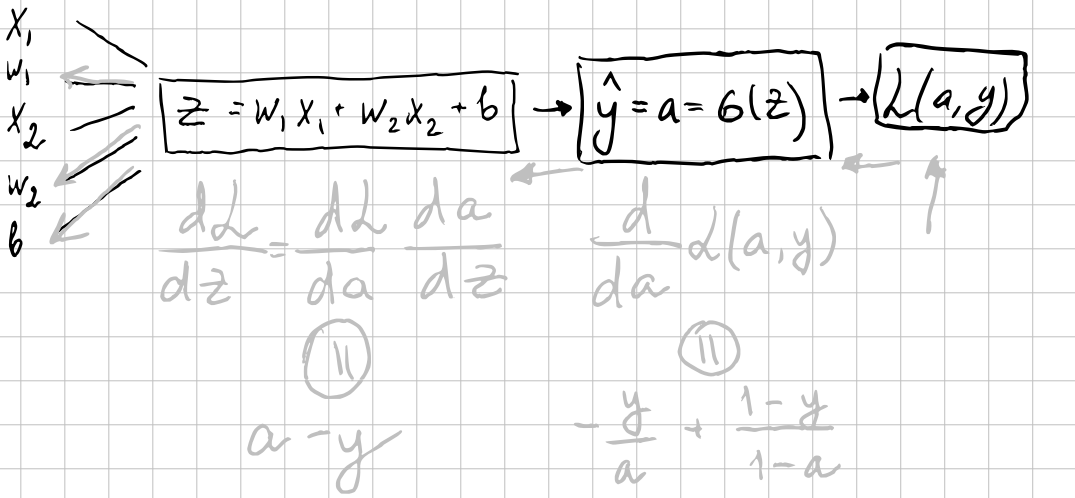
Logistic Regression Gradient Descent

$$z = w^T x + b$$

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

$$\mathcal{L}(a, y) = -(y \log(a) + (1-y) \log(1-a))$$





$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dz} x_1 \qquad \frac{dh}{db} = \frac{d\mathcal{L}}{dz}$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dz} x_2$$

$$w_1 = w_1 - \alpha \, dw_1$$

$$w_2 = w_2 - \alpha \, dw_2$$

$$b = b - \alpha \, db$$

Gradient descent for logistic regression

One step of Gradient Descent:

$J=0;\ dw_1=0;\ dw_2=0;\ db=0$

For $i=1$ to m:

$$z^{(i)} = w^T x^{(i)} + b$$

$$a^{(i)} = \sigma(z^{(i)})$$

$$J \mathrel{+}= -\left[y^{(i)} \log a^{(i)} + (1-y^{(i)}) \log(1-a^{(i)})\right]$$

$$dz^{(i)} = a^{(i)} - y^{(i)}$$

$$dw_1 \mathrel{+}= x_1^{(i)} dz^{(i)}$$

$$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$$

$$db \mathrel{+}= dz^{(i)}$$

$n = 2$

$J \mathrel{/}= m$

$dw_1 \mathrel{/}= m;\ dw_2 \mathrel{/}= m;\ db \mathrel{/}= m.$

$$w_1 := w_1 - \alpha\, dw_1$$

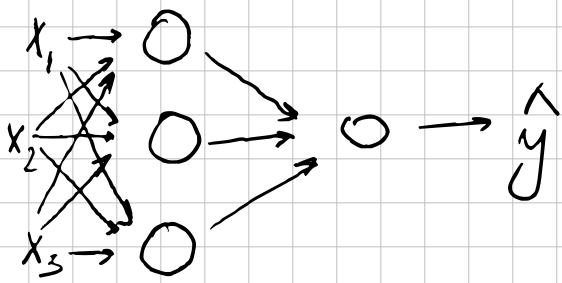
$$w_2 := w_2 - \alpha\, dw_2$$

$$b := b - \alpha\, db$$

Problem with the approach:

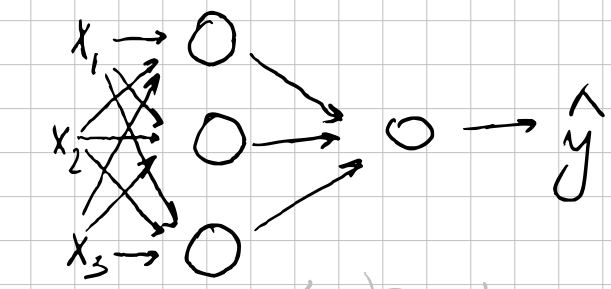
- multiple for loops, i.e. complexity is $O(n \times m)$

Neural Network



Notation

$w^{[1]}$, $b^{[1]}$ — first layer

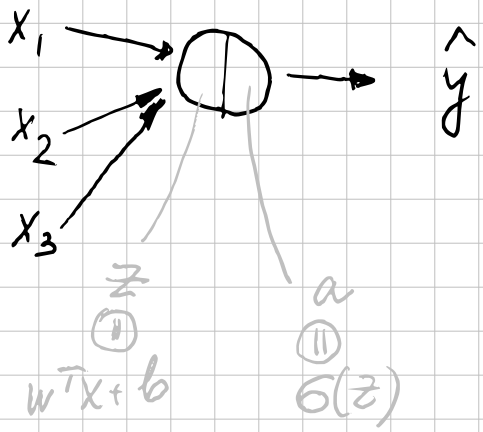


input layer — hidden layer — output layer

2 layers neural network (we don't count input layer)

Alternative notation for input layer is $a^{[0]}$

Neuron

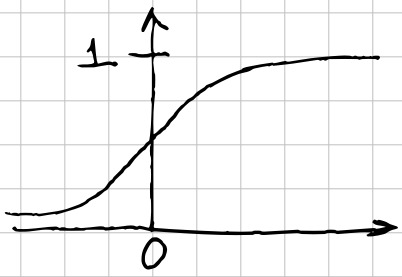


$a_i^{[l]}$ ← layer

← node in layer

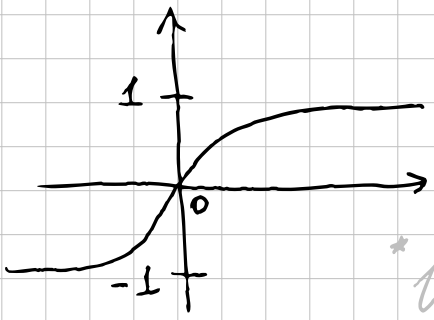
Activation functions

Examples:



Sigmoid

$$a(z) = \frac{1}{1+e^{-z}}$$



Hyperbolic tangent

$$a(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

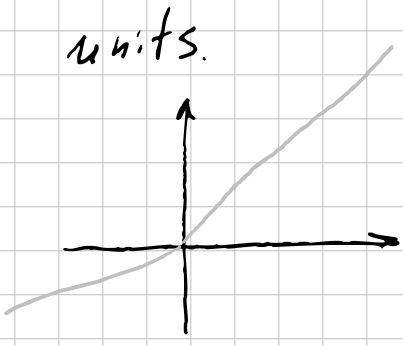
* Usually works better than sigmoid as shifts mean to 0. One possible exception for it is the last layer if output is binary classification $\in \{0, 1\}$

Problem of sigmoid and tank is gradient vanishing: values that are really big or really small usually are close to each other after transformation.

ReLU

$$a(z) = \max(0, z)$$

Sigmoid is very natural for the output layer for binary classific. $\{0, 1\}$. ReLU is increasingly default choice for other units.



Leaky ReLU

$$a(z) = \max(0.01z, z)$$

it can also be a hyperparameter

Why do you need Non-Linear Activation Functions?

If we replace non-linear transformations with linear functions, we would be able to substitute all layers with the only layer.

No matter how many layers NN has, if it only has linear activations, it will only output linear function.

Let's

$$a^{[i]} = z$$

- $a^{[1]} = z^{[1]} = W^{[1]}X + b^{[1]}$
- $a^{[2]} = z^{[2]} = W^{[2]}X + b^{[2]}$ (=)

(=) $W^{[2]}\left(W^{[1]}X + b^{[1]}\right) + b^{[2]}$ (=)

(=) $\underbrace{W^{[2]}W^{[1]}}_{W'}X + \underbrace{W^{[2]}b^{[1]} + b^{[2]}}_{b'}$ (=)

(=) $W'X + b'$

Derivatives of activation functions

- Sigmoid

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\frac{d}{dz} g(z) = \frac{-1}{(1+e^{-z})^2} \cdot e^{-z} \cdot (-1) \Leftrightarrow$$

$$\Leftrightarrow \frac{e^{-z}}{(1+e^{-z})^2} = \frac{e^{-z}}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} = \frac{e^{-z}+1-1}{1+e^{-z}} \odot$$

$$\odot \frac{1}{1+e^{-z}} = \left(1 - \frac{1}{1+e^{-z}}\right) \cdot \frac{1}{1+e^{-z}} = (1-g(z))\,g(z)$$

- Hyperbolic tangent

$$g(z) = \tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$$

$$\frac{d}{dz} g(z) = \left(\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}\right)' \Leftrightarrow$$

$$\Leftrightarrow \frac{(e^{z}-e^{-z})'(e^{z}+e^{-z}) - (e^{z}-e^{-z})(e^{z}+e^{-z})'}{(e^{z}+e^{-z})^2}$$

$$\Leftrightarrow \frac{(e^{z}+e^{-z})^2 - (e^{z}-e^{-z})^2}{(e^{z}+e^{-z})^2} = 1 - \left(\frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}\right)^2 \Leftrightarrow$$

$$\Leftrightarrow 1 - (\tanh(z))^2$$

- ReLU

$$g(z) = \max(0, z)$$

$$\frac{d}{dz} g(z) = \begin{cases} 0, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

- Leaky ReLU

$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01, & z \leq 0 \\ 1, & z > 0 \end{cases}$$

Gradient Descent for neural networks

NN with a single hidden layer

Parameters

$w^{[1]}$, $(n^{[1]}, n^{[0]})$

$b^{[1]}$, $(n^{[1]}, 1)$

$w^{[2]}$, $(n^{[2]}, n^{[1]})$

$b^{[2]}$, $(n^{[2]}, 1)$

Cost function

$$J(w^{[1]}, b^{[1]}, w^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{n} \mathcal{L}(\hat{y}, y)$$

Gradient descent:

repeat {

forward prop.

$$\text{compute predicts } (\hat{y}^{(i)}, i = 1 \dots m)$$

backward prop.

$$dw^{[1]} = \frac{dJ}{dw^{[1]}},$$

$$db^{[1]} = \frac{dJ}{db^{[1]}}$$

$$\vdots$$

$$w^{[1]} = w^{[1]} - \alpha\, dw^{[1]}$$

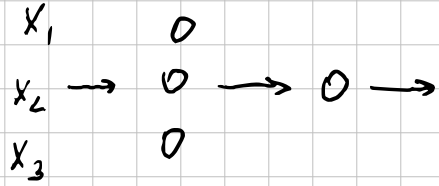
$$\vdots$$

}

Logistic regression

x, w, b → $z = w^T x + b$ → $a = \sigma(z)$ → $\mathcal{L}(a, y)$

NN



x, $W^{[1]}$, $b^{[1]}$ → $z^{[1]} = W^{[1]} x + b^{[1]}$ → $a = \sigma(z)$ → $z^{[2]} = W^{[2]} x + b^{[2]}$ → ⊖

$W^{[2]}$, $b^{[2]}$ → $z^{[2]} = W^{[2]} x + b^{[2]}$

⊖ $a = \sigma(z^{[2]})$ → $\mathcal{L}(a^{[2]}, y)$

$$\frac{d\mathcal{L}}{dz^{[2]}} = a^{[2]} - y$$

$$\frac{d\mathcal{L}}{dw^{[2]}} = dz^{[2]} a^{[1]T} \qquad \frac{d\mathcal{L}}{db^{[2]}} = dz^{[2]}$$

$$dz^{[1]} = w^{[2]T} dz^{[2]} * g^{[1]'}(z^{[1]})$$

Random initialization

Let's initialize everything with 0. In that case all derivatives are identical. All units in the same layer compute the same function. (symmetry breaking prob.)

The solution to this is initialize every parameter at random.

Deep L-layer Neural Network

Number of hidden layers can be considered as another hyperparameter

L (number of layers in the network)

$n^{[l]}$ (number of nodes/units in layer l)

$a^{[l]}$ (activations in layer l)

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

$w^{[l]}$ (weights for $z^{[l]}$)

Forward propagation in a deep network

X: $z^{[1]} = W^{[1]} x + b^{[1]}$

$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$

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

... until you get to the output layer

$z^{[n]} = W^{[n]} a^{[n-1]} + b^{[n]}$

$a^{[n]} = g^{[n]} a^{[n]}$

General rule is

$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$

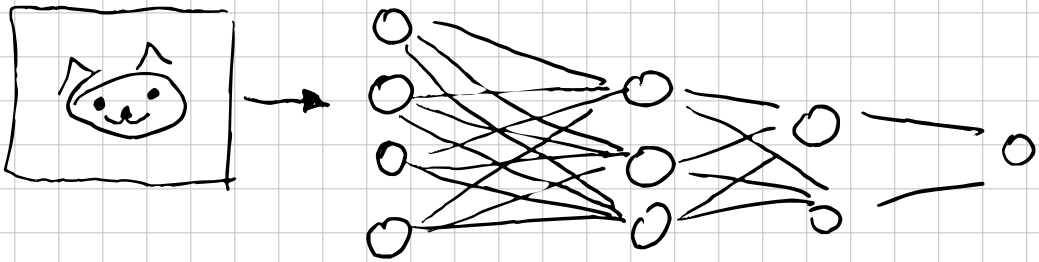
$a^{[l]} = g^{[l]}(z^{[l]})$

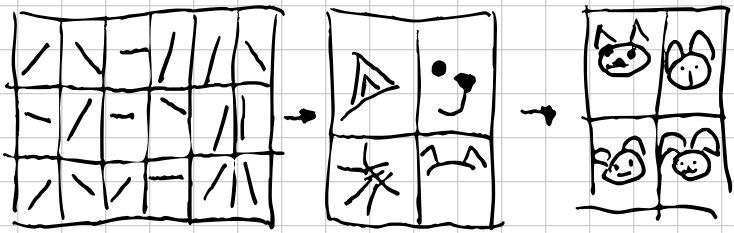
$W^{[l]}$: $(n^{[l]}, n^{[l-1]})$

$a^{[l]}$ (x is $a^{[0]}$): $(n^{[l]}, 1)$

$b^{[l]}$: $(n^{[l]}, 1)$

Why Deep Representations?





Allows to compose more simple features into more complicated ones

Building blocks of Deep Neural networks

layer l: $W^{[l]}$, $b^{[l]}$

forward: input $a^{[l-1]}$ output $a^{[l]}$

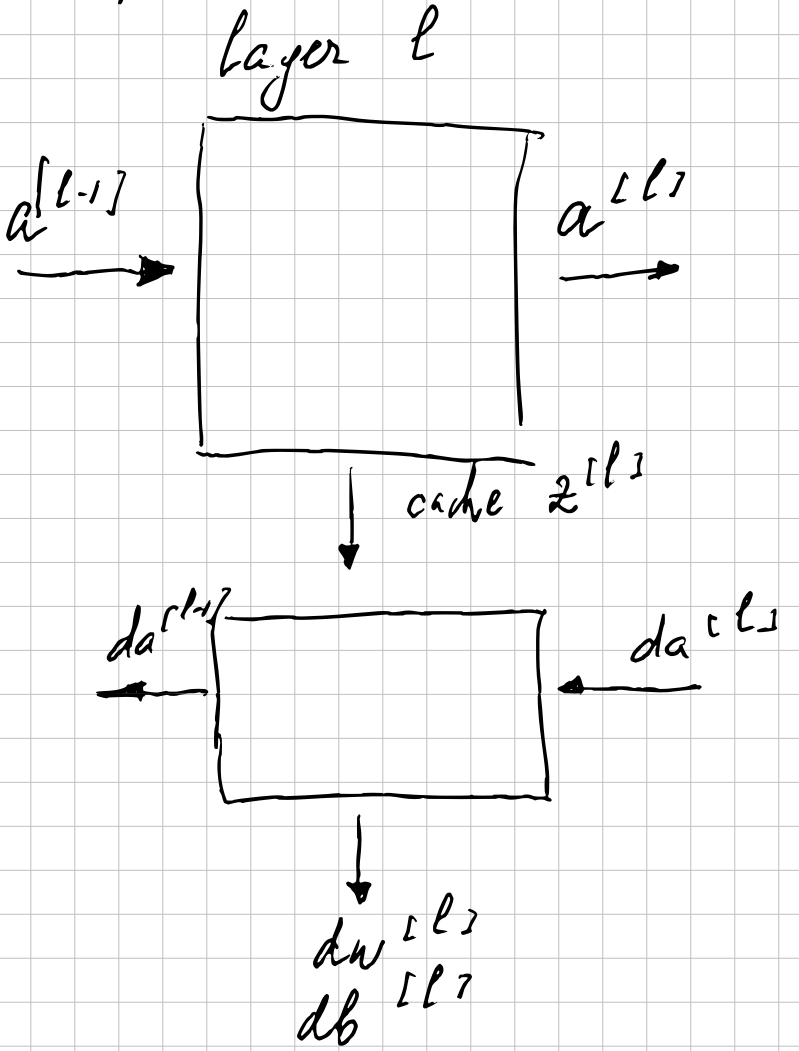
$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$,

cache $z^{[l]}$

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

backward: input $da^{[l]}$,

output $da^{[l-1]}$



Forward propagation for layer l

Input $a^{[l-1]}$

Output $a^{[l]}$, cache $(z^{[l]})$

$$z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$$

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

Backward propagation for layer l

Input $da^{[l]}$

Output $da^{[l-1]}$, $dW^{[l]}$, $db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$$

prime, i.e. derivative

$$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]T}$$

$$db^{[l]} = dz^{[l]}$$

$$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$$

Parameters vs. hyperparameters

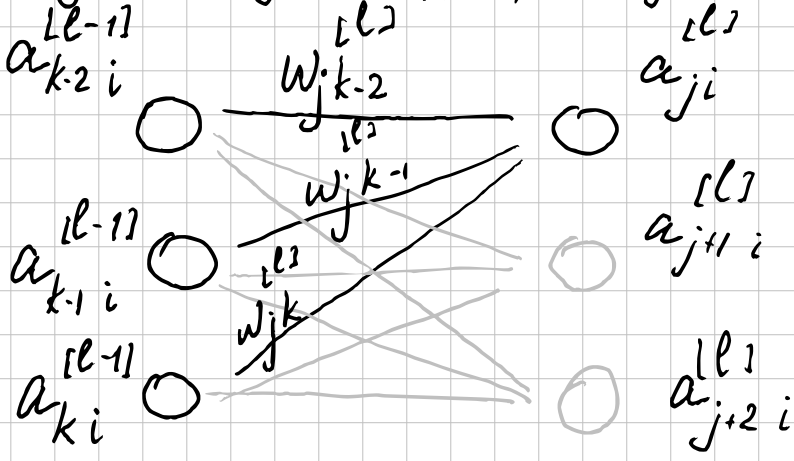
Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \ldots$

Hyperparameters: learning rate α, # iterations, # hidden layers, # hidden units, choice of activation function

Forward propagation

$$z_{ji}^{[l]} = \sum_k w_{jk}^{[l]} a_{ki}^{[l-1]} + b_j^{[l]}$$

$$a_{ji}^{[l]} = g_j^{[l]}\left(z_{1i}^{[l]}, \dots z_{ji}^{[l]} \dots z_{n^{[l]}i}^{[l]}\right)$$



Vectorized version (for one example i)

$$\begin{bmatrix} z_{1i}^{[l]} \\ \vdots \\ z_{ji}^{[l]} \\ \vdots \\ z_{n^{[l]}i}^{[l]} \end{bmatrix} = \begin{bmatrix} w_{11}^{[l]} & \cdots & w_{1k}^{[l]} & \cdots & w_{1n^{[l-1]}}^{[l]} \\ \vdots & & \vdots & & \vdots \\ w_{j1}^{[l]} & \cdots & w_{jk}^{[l]} & \cdots & w_{jn^{[l-1]}}^{[l]} \\ \vdots & & \vdots & & \vdots \\ w_{n^{[l]}1}^{[l]} & \cdots & w_{n^{[l]}k}^{[l]} & \cdots & w_{n^{[l]}n^{[l-1]}}^{[l]} \end{bmatrix} \begin{bmatrix} a_{1i}^{[l-1]} \\ \vdots \\ a_{ki}^{[l-1]} \\ \vdots \\ a_{n^{[l-1]}i}^{[l-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[l]} \\ \vdots \\ b_j^{[l]} \\ \vdots \\ b_{n^{[l]}}^{[l]} \end{bmatrix}$$

$$\begin{bmatrix} a_{1i}^{[l]} \\ \vdots \\ a_{ji}^{[l]} \\ \vdots \\ a_{n^{[l]}i}^{[l]} \end{bmatrix} = \begin{bmatrix} g_1^{[l]}\left(z_{1i}^{[l]} \cdots z_{ji}^{[l]} \cdots z_{n^{[l]}i}^{[l]}\right) \\ \vdots \\ g_{n^{[l]}}^{[l]}\left(\dots\right) \end{bmatrix}$$

or

$$Z_{:,i}^{[l]} = W^{[l]} a_{:,i}^{[l-1]} + b^{[l]}$$

$$a_{:,i}^{[l]} = g^{[l]}\left(Z_{:,i}^{[l]}\right)$$

Vectorizing over all examples

$$Z^{[l]} = \left[Z_{:,1}^{[l]} \cdots Z_{:,i}^{[l]} \cdots Z_{:,m}^{[l]} \right] (=)$$

$$(=)\ W^{[l]} \left[a_{:,1}^{[l-1]} \cdots a_{:,i}^{[l-1]} \cdots a_{:,m}^{[l-1]} \right] +$$

$$+ \left[b^{[l]} \cdots b^{[l]} \cdots b^{[l]} \right] (=)$$

$$(=)\ W^{[l]} A^{[l-1]} + \text{broadcast}(b^{[l]})$$

Cost function

$$J = f(\hat{y}, y) = f(A^{[l]}, y)$$

Backward propagation

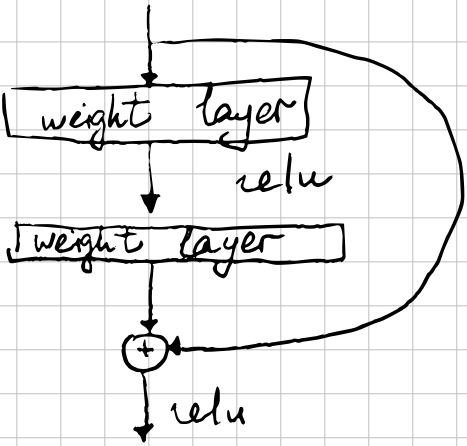
$$\frac{\partial J}{\partial w_{jk}^{[l]}} = \sum_i \frac{\partial J}{\partial z_{ji}^{[l]}} \frac{\partial z_{ji}^{[l]}}{\partial w_{jk}^{[l]}} = \sum_i \frac{\partial J}{\partial z_{ji}^{[l]}} \cdot a_{ki}^{[l-1]}$$

$$\frac{\partial J}{\partial b_j^{[l]}} = \sum_i \frac{\partial J}{\partial z_{ji}^{[l]}} \frac{\partial z_{ji}^{[l]}}{\partial b_j^{[l]}} = \sum \frac{\partial J}{\partial z_{ji}^{[l]}}$$

$$\frac{\partial J}{\partial z_{ji}^{[l]}} = \sum_p \frac{\partial J}{\partial a_{pi}^{[l]}} \frac{\partial a_{pi}^{[l]}}{\partial z_{ji}^{[l]}}$$

$$\frac{\partial J}{\partial a_{ki}^{[l-1]}} = \sum_j \frac{\partial J}{\partial z_{ji}^{[l]}} \frac{\partial z_{ji}^{[l]}}{\partial a_{ki}^{[l-1]}} = \sum_j \frac{\partial J}{\partial z_{ji}^{[l]}} w_{jk}^{[l]}$$

Deep Residual Learning



$H(x)$ is the true function we want to learn

Lets pretend we want to learn $F(x) = H(x) - x$ instead

The original function is then $F(x) + x$

Depthwise Separable Convolution

A standard convolution is an operator of size

$$D_k * D_k * C_{in}$$

where D_k - convolutional kernel size, C_{in} - input number of channels

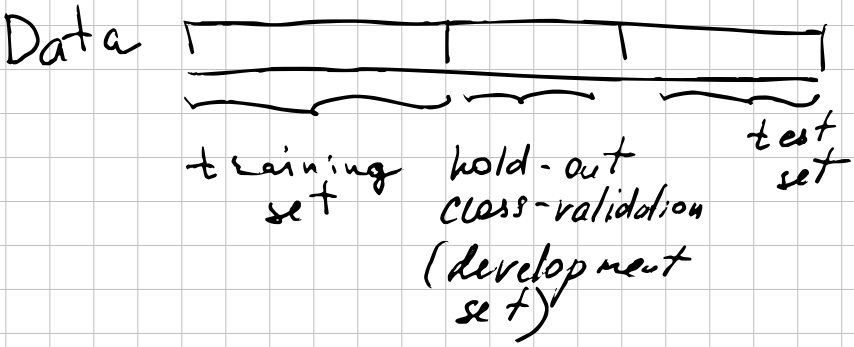
Computational complexity of the convolutional layer is $D_k * D_k * C_{in} * D_f * D_f * C_{out}$

Practical aspects of deep learning

You may want to decide on how many of each you want:

- # layers
- # hidden units
- learning rates
- activation functions

...



Usually people do: 60/20/20 (%) split

Dev. set helps to evaluate multiple algorithm choices

Mismatch train/test distribution

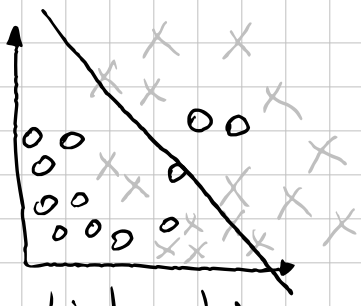
Training set: cat pictures from webpages

Dev/test sets: cat pictures from users using your app

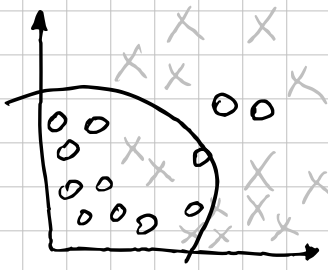
Rule of thumb: make sure dev and test come from the same distribution.

Not having a test set might be okay. (Only dev set)

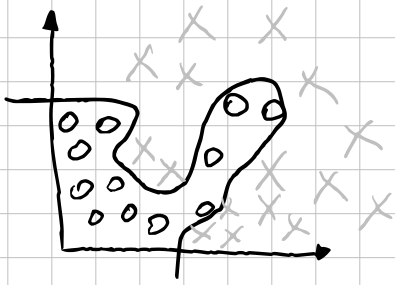
Bias / Variance



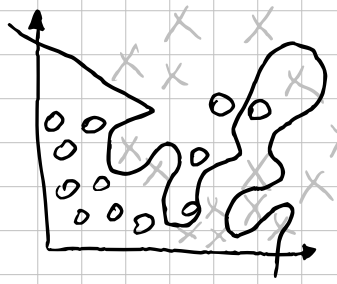
high bias
(underfit)



"just right"



high variance
(overfitting)



high bias &
high variance

Cat classification

y=1

y=0

The key metrics for high-dimens. data are:

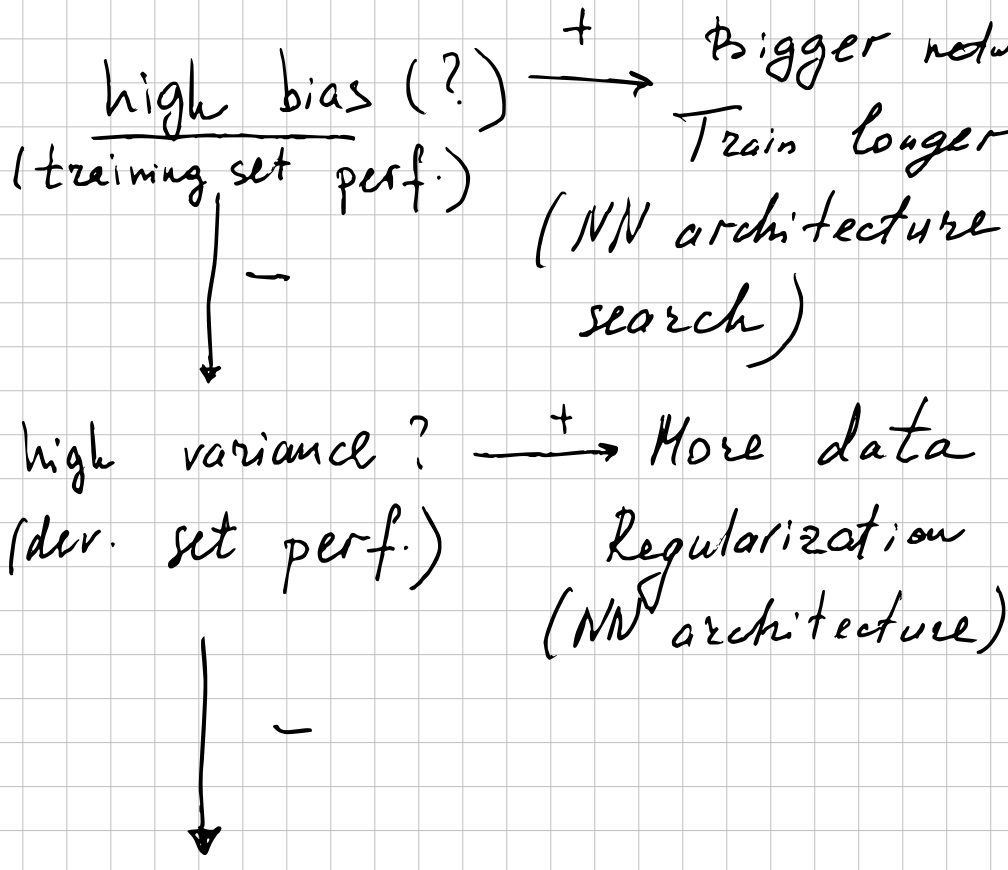
train set error: 1%

dev set error: 11%

may signalise of overfitting as good on train and bad on dev (high variance)

15%	15%
16%	30%
not doing well on both sets, so underfitting (high bias)	high bias & high variance

Basic recipe for Machine Learning



"bias variance tradeoff"

Regularization

$$\min_{w,b} J(w,b) \quad , \quad w \in \mathbb{R}^{n_x} \quad , \quad b \in \mathbb{R}$$

For logistic regression:

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

regularization factor

$$\|w\|^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$$

L2 regular.

$$\frac{\lambda}{2m} \sum_{i=1}^{n_x} |w| = \frac{\lambda}{2m} \|w\|_1$$

L1 regular.

λ - regularization parameter

For neural network:

$$J\left(w^{[1]}, b^{[1]}, \dots, w^{[l]}, b^{[l]}\right) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}\left(\hat{y}^{(i)}, y^{(i)}\right) +$$

$$+ \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|^2$$

$$\|w^{[l]}\|^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} \left(w_{ij}^{[l]}\right)^2$$

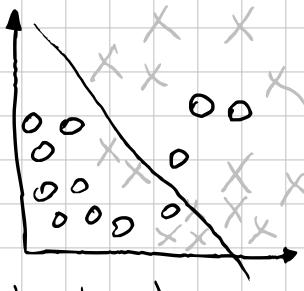
Frobenius norm

Gradient descent:

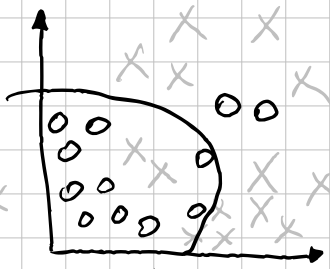
$$dw^{[l]} = (\text{from backprop}) + \frac{\lambda}{m} w^{[l]}$$

$$w^{[l]} := w^{[l]} - \alpha \, dw^{[l]}$$

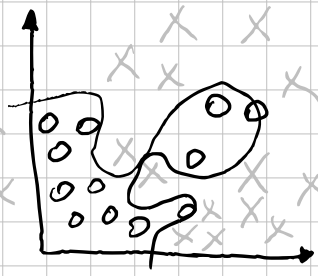
Why regularization reduces overfitting?



high bias



just right

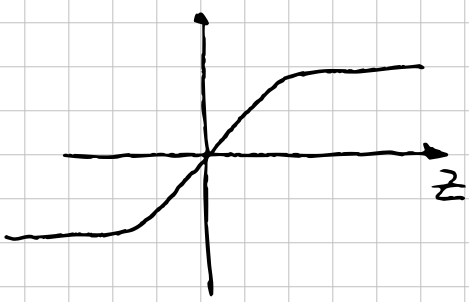


high variance

$$J(w^{[l]}, b^{[l]}) = \frac{1}{m} \sum_{i}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) \quad (+)$$

$$(+) \quad \frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|_F^2$$

Another intuition



$g(z) = \tanh(z)$

z taking values from this interval

If λ is large ($\lambda \uparrow$)
my parameters are small ($w^{[l]} \downarrow$)
because they are penalized
by the cost function

$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$

Every layer will be roughly ≈ linear

If w is small $z^{[l]}$ takes values from small limited range

Dropout regularization

You eliminate some nodes (randomly) from the network and train this reduced network.

- Inverted dropout

Illustrate with $L=3$

```
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0],
a3.shape[1]) < keep_prob

a3 = np.multiply(a3, d3)
```

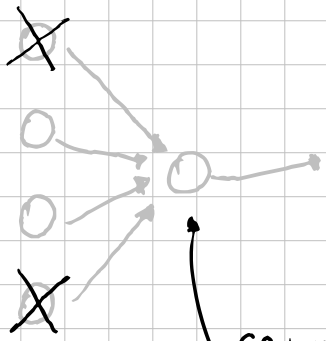
zeroing-out elements

```
# Scaling a3 distribution back
a3 /= keep_prob
```

At test time we don't add any drop out.

Understanding dropout

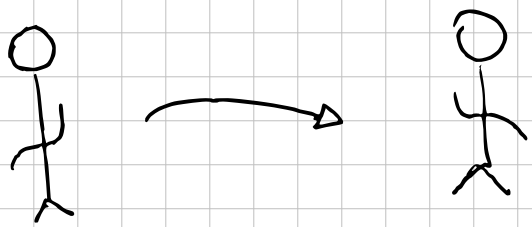
Intuition: can't rely on any one feature, so have to spread out weights.



cannot rely on any prev. nodes feature as the feature may go away

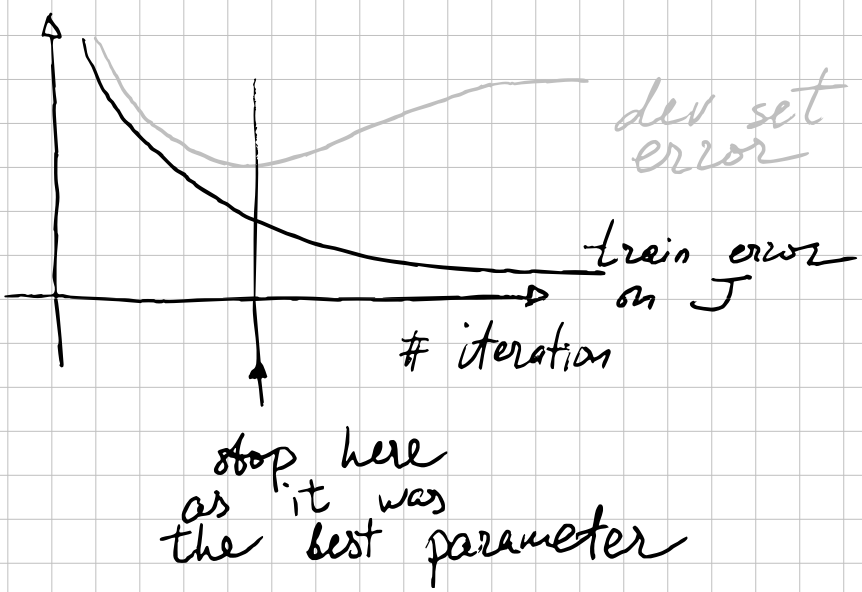
Other regularisation methods

Data augmentation



- Image flip
- Randomly crop the image
- Random distortions / transformations

Early stopping



Normalizing training set

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Normalizing consists of several steps:

1. Subtract mean

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

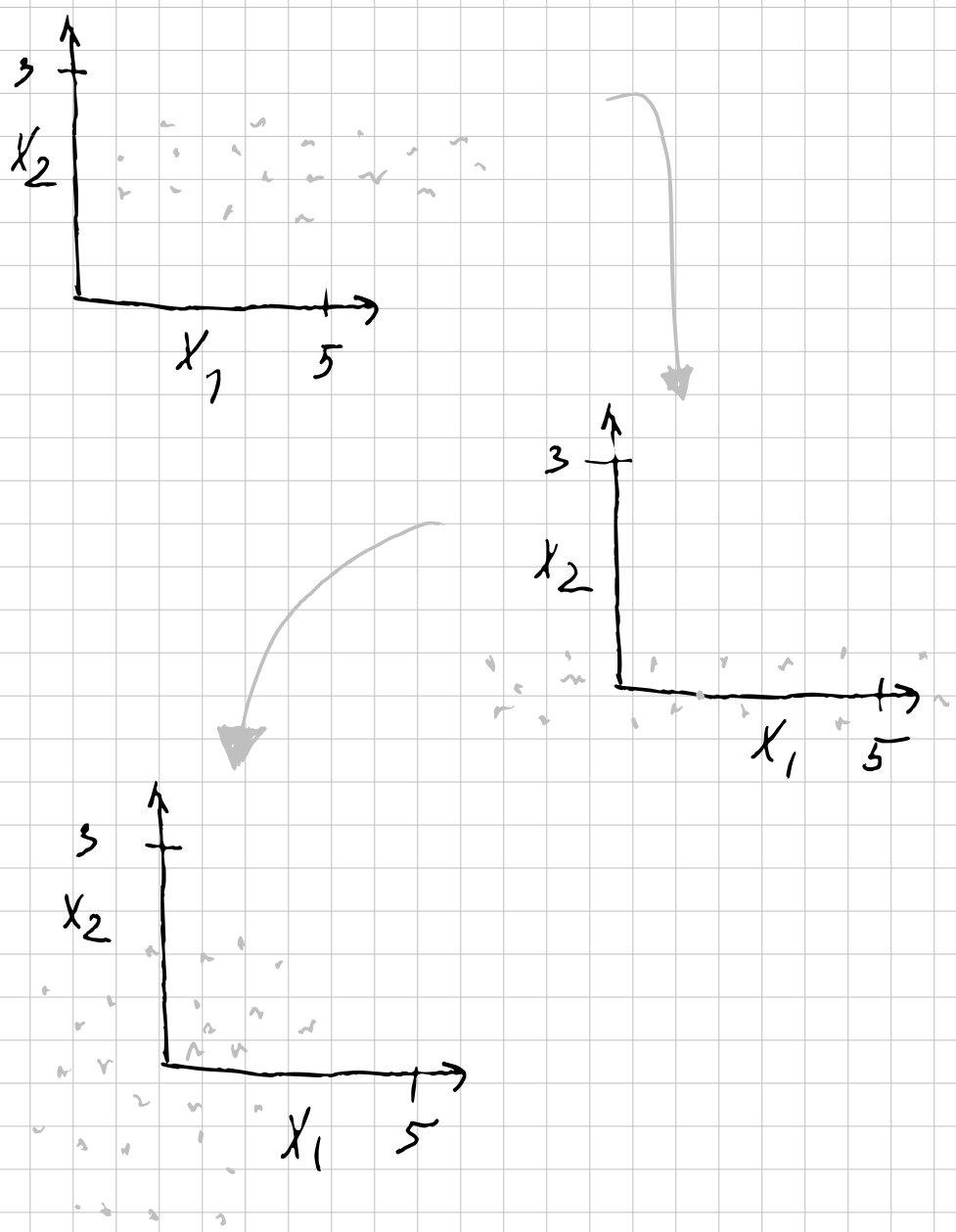
$$X = X - \mu$$

2. Normalize variance

$$\sigma = \frac{1}{m} \sum_{i=1}^{m} \left(x^{(i)}\right)^2$$

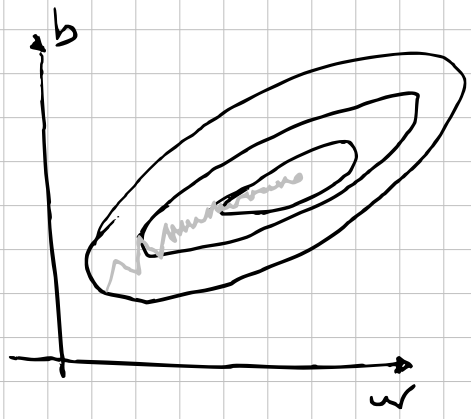
element wise square

$$X = X / \sigma$$

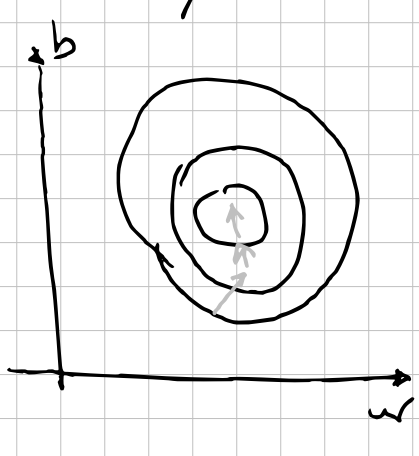


Use same μ and σ to normalize both test and training sets

Non-normalized loss function



Normalized loss function

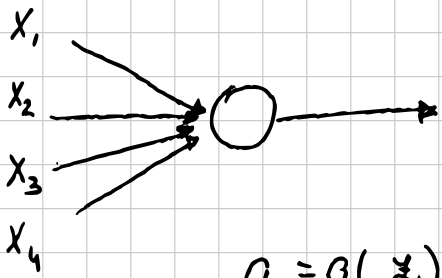


Vanishing / Exploding Gradients

- $w^{[l]} > I$ the activations can explode in a really deep network
- $w^{[l]} < I$ the activations will decrease exponentially in deep networks

The same happens with derivatives.

Weights Initialization for Deep Networks



$$a = g(z)$$

$$z = w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots + w_n x_n + b$$

$b = 0$ in this example

Large n $\rightarrow$ Smaller w_i

$$Var(w) = \frac{1}{n}$$

$$w^{[l]} = np.random.rand(shape) * np.sqrt\left(\frac{1}{n^{[l-1]}}\right)$$

ReLU $g^{[l]}(z) = ReLU(z)$

Other variance:

$$ReLU = \frac{2}{n^{[l-1]}}$$

$$tanh = \sqrt{\frac{1}{n^{[l-1]}}}$$ } Xavier initialisation

$$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$$ - another popular

Numeric Approximation of Gradients

Checking your derivative computation

$f(\theta) = \theta^3$

$\varepsilon \to 0$

$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{(\theta+\varepsilon)-(\theta-\varepsilon)} = \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx$$

$\approx$ derivative

Gradient check for a neural network

Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ.

Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

Q: Is $d\theta$ a gradient of $J(\theta)$?

for each i:

$$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2 \dots \theta_i + \varepsilon, \dots) - J(\theta_1, \theta_2 \dots \theta_i - \varepsilon, \dots)}{2\varepsilon} \approx d\theta[i]$$

$$d\theta_{approx} \overset{?}{\approx} d\theta$$

$$\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} < \text{some small epsilon, like } 10^{-7}$$

← The actual check

error:	
10^{-7}	great
10^{-5}	so-so
10^{-3}	worry

Gradient Checking tricks & hints

- Don't use in training - only to debug;
- If algorithm fails grad check, look at components to try to identify bugs;

For example db may be far off, but dW may be just fine

- Remember regularization;
- Does not work with dropout (implement check without dropout)
- Run at random initialisation; perhaps again after some training.

Mini-batch Gradient Descent

Batch vs. mini-batch gradient descent

Vectorization allows you to efficiently compute on m examples.

$$X = [\, x^{(1)} \; x^{(2)} \; \dots \; x^{(m)} \,]$$

$$Y = [\, y^{(1)} \; y^{(2)} \; \dots \; y^{(m)} \,]$$

X dimension is (n_x, m)

Y dimension is $(1, m)$

What if $m = 5 \cdot 10^7$?

mini-batches are small (tiny, baby) split ups of the training set

$X^{\{1\}}$ - is the first mini-batch split

For example, if mini-batch size is 10^3

$$X = [\, \underbrace{x^{(1)} \dots x^{(1000)}}_{X^{\{1\}}} \,|\, \underbrace{x^{(1001)} \dots x^{(2000)}}_{X^{\{2\}}} \,|\, \dots x^{(m)} \,]$$

Y set is split up in the same manner.

> Notation recap.
>
> $x^{(i)}$ - i-th example
>
> $z^{[l]}$ - l-th layer
>
> $X^{\{t\}}$ - t-th mini-batch

! Let's consider

$m = 5 \cdot 10^7$ mini-batch size $= 10^3$

$X^{\{1\}}$ dimension is $(n_x, 1000)$

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

vectorized implementation

1. forward prop on $X^{\{t\}}$

$$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$$

$$A^{[1]} = g^{[1]}(Z^{[1]})$$

2. Compute cost function

$$J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

One step of Gradient Descent using $X^{\{t\}}, Y^{\{t\}}$ (as if $m = 1000$)

3. Backprop. to compute gradients

4. Update

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

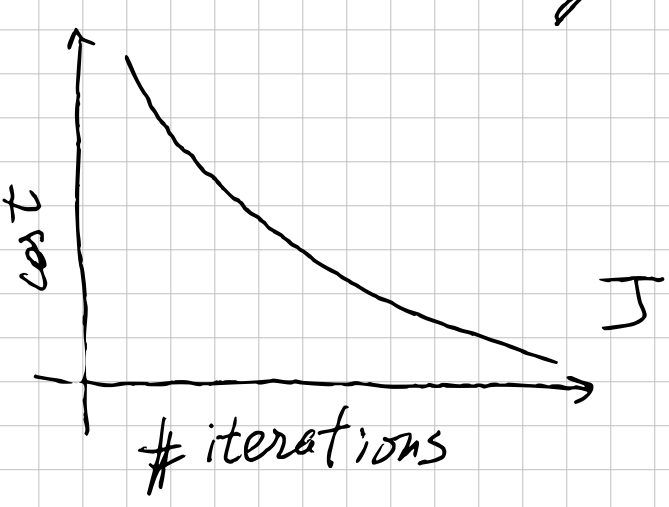
$$b^{[l]} := b^{[l]} - \alpha db^{[l]}$$

It is called "1 epoch", i.e. one pass through the training set.

Understanding Mini-batch Gradient Descent

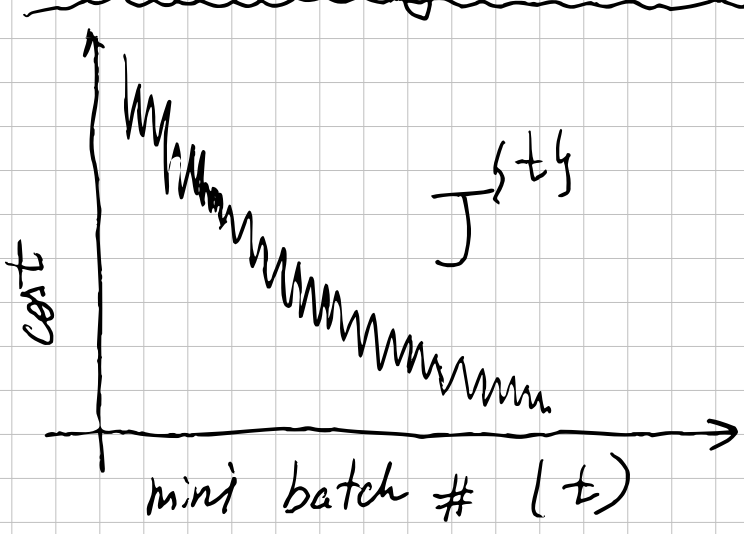
Batch gradient descent

You go through the entire training set



J should decrease over iterations (if you see opposite then something is wrong)

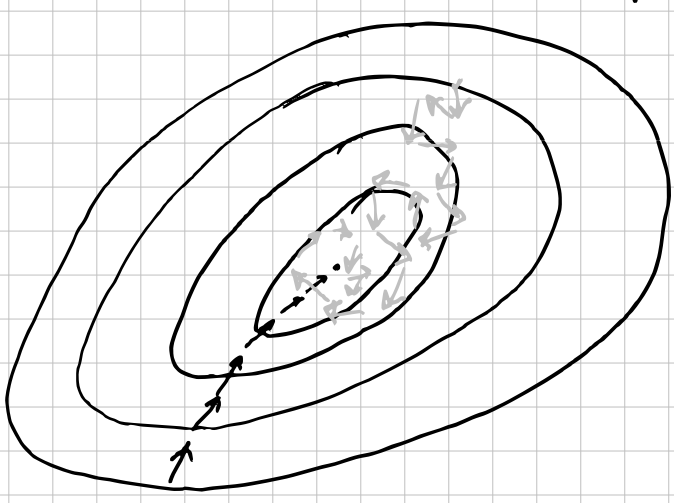
Mini-batch gradient descent



Its okay if it does not go down every iteration.

Choosing your mini-batch size

- If mini-batch size = m : Batch gradient descent, in this extreme case $(X^{\{t\}}, Y^{\{t\}}) = (X, Y)$
- If mini-batch size = 1 : Stochastic gradient descent, every example is its own mini-batch.



Batch gradient descent

Stochastic gradient descent

In practice mini-batch gradient descent is in-between

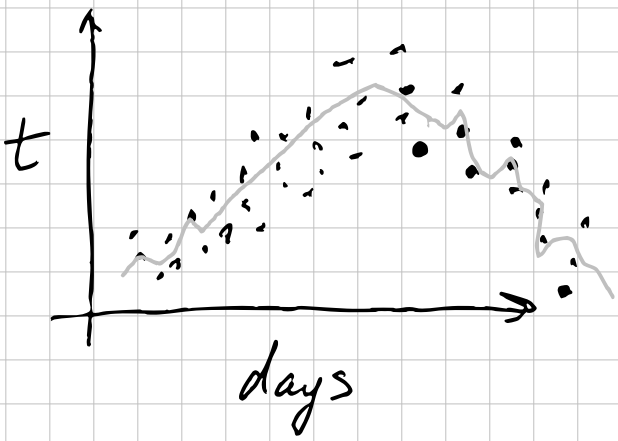
Batch grad. desc	Mini-batch grad. desc	Stochastic grad. desc.
↓	↓	↓
• Too long per iter.	• Fast learning • Vectorization • Make progress w/o processing the entire training set	• lose speedup from vectorization

- If small training set: use batch gradient descent ($m \leq 2 \cdot 10^3$)
- Otherwise typical mini-batch sizes: 64, 128, 256, 512, 1024
  (more common values)

! Make sure mini-batch fits in CPU / GPU memory

Exponentially weighted moving averages

Temperature in London



$\theta_1 = 4\,^\circ C$

$\theta_2 = 9\,^\circ C$

$\vdots$

$\theta_{180} = 15\,^\circ C$

$V_0 = 0$

$V_1 = 0.9 V_0 + 0.1 \theta_1$

$V_2 = 0.9 V_1 + 0.1 \theta_2$

$\vdots$

$V_t = 0.9 V_{t-1} + 0.1 \theta_t$

Exponentially weighted average (— on the plot)

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

V_t is an approximate average over $\left(\frac{1}{1-\beta}\right)$ days' temperature

For example,

$\beta = 0.9$ is ≈ 10 days' t°

$\beta = 0.98$ is ≈ 50 days' t°

$\beta = 0.5$ is ≈ 2 days' t°

Understanding exponentially weighted average

$$v_t = \beta v_{t-1} + (1-\beta)\theta_t$$

$$\begin{cases} V_{100} = 0.9 V_{99} + 0.1 \theta_{100} \\ V_{99} = 0.9 V_{98} + 0.1 \theta_{99} \\ V_{98} = 0.9 V_{97} + 0.1 \theta_{98} \\ \dots \end{cases}$$

$$V_{100} = 0.1\theta_{100} + 0.9 V_{99} \quad (=)$$

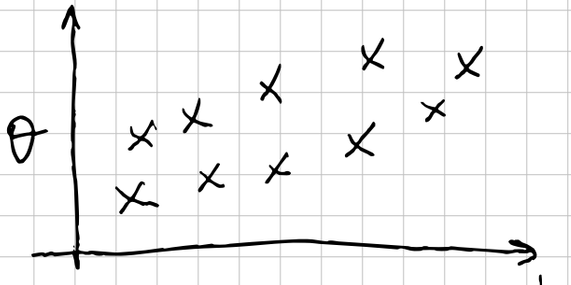
$$(=)\ 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9 V_{98}) \quad (=)$$

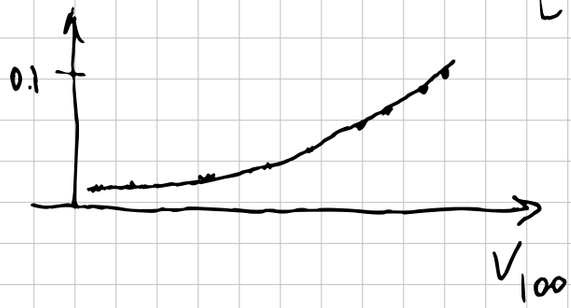
$$(=) 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9 V_{97}))$$

and so on.

$$V_{100} = 0.1\theta_{100} + 0.09\theta_{99} + 0.081\theta_{98} \quad (+)$$

$$(+)\ 0.1 \cdot (0.9)^3 \theta_{97} + \dots$$





$$(1-\varepsilon)^{1/\varepsilon} \approx \frac{1}{e}$$

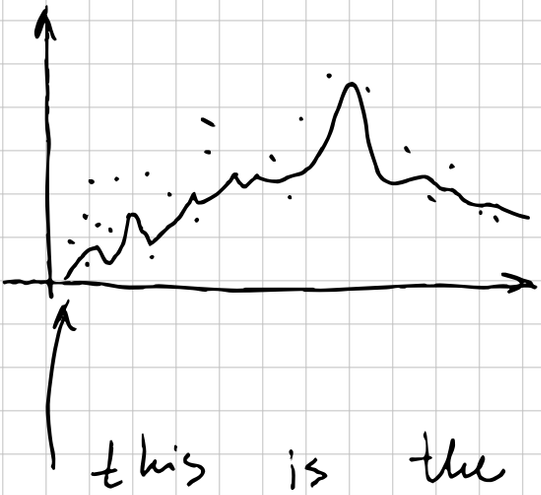
$$0.9^{10} \approx \frac{1}{e}$$

$$0.98^{50} \approx \frac{1}{e}$$

Bias correction

Using $v_t = \beta v_t + (1-\beta)\theta_t$

we get curve that is low



this is the problem

Original algorithm

$v_0 = 0$

$v_1 = \underbrace{0.9 v_0}_{0} + 0.1\theta_1$

...

and so on.

Instead of taking v_t

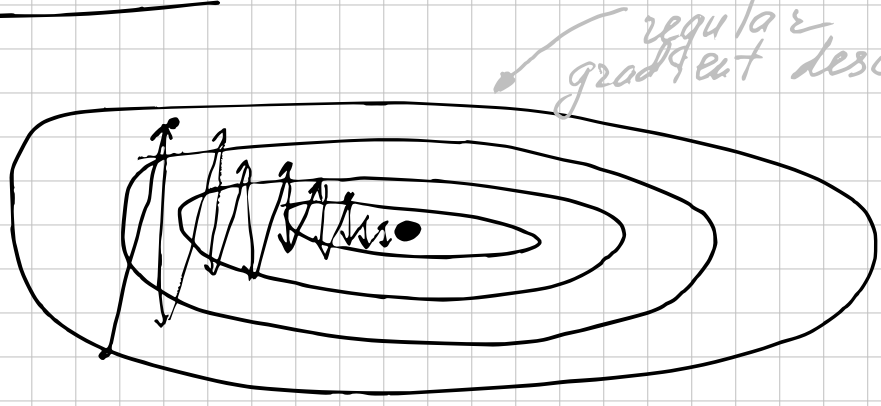
take $\frac{v_t}{1-\beta^t}$ — bias correction

$$v_t = \beta \frac{v_{t-1}}{1-\beta^t} + (1-\beta)\theta_t$$

$\frac{v_t}{1-\beta^t}$ becomes closer to v_t

when t is big

Gradient descent with momentum



You want:

↕ slower learning ⟷ faster learning

Momentum:

On iteration t:

Compute dW, db on current batch / mini-batch

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

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

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

$$b := b - \alpha V_{db}$$

Hyperparameters: α, β

In practice people don't bother with bias correction.

RMSprop

(root mean square)

On iteration t:

Compute dW, db on current batch (mini-batch)

element-wise

$$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$$

$$S_{db} = \beta S_{db} + (1-\beta) db^2$$

$$W := W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$

$$b := b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Adam optimization algorithm

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

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

On iteration t:

Compute dW, db using current mini-batch

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1) dW$$

$$V_{db} = \beta_1 V_{db} + (1-\beta_1) db$$

} momentum grad. desc.

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2) dW^2$$

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

} RMSprop

$$V_{dw}^c = \frac{V_{dw}}{(1-\beta_1^t)}$$

$$V_{db}^c = \frac{V_{db}}{(1-\beta_1^t)}$$

$$S_{dw}^c = \frac{S_{dw}}{(1-\beta_2^t)}$$

$$S_{db}^c = \frac{S_{db}}{(1-\beta_2^t)}$$

} bias correction

$$W = W - \alpha \frac{V_{dw}^c}{\sqrt{S_{dw}^c} + \varepsilon}$$

$$b = b - \alpha \frac{V_{db}^c}{\sqrt{S_{db}^c} + \varepsilon}$$

to avoid devision by 0

Hyperparameters:

α : needs to be tuned

β_1 : 0.9 (first moment)

β_2 : 0.999 (second moment)

ε : 10^{-8}

Adam stands for adaptive moment estimation

Learning rate decay

1 epoch = 1 pass through the data

$$\alpha = \frac{1}{1 + \underbrace{\text{decay\_rate}}_{\text{another hyper parameter}} \times \text{epoch\_number}} \alpha_0$$

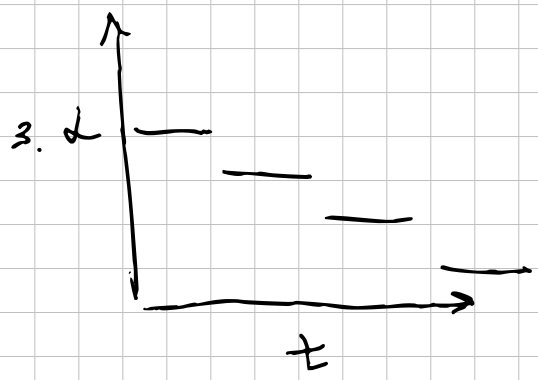
Other learning rate decay methods

1. $\alpha = 0.95^{\text{epoch\_number}} \cdot \alpha_0$ —

— exponential decay

2. $\alpha = \frac{k}{\sqrt{\text{epoch\_number}}} \cdot \alpha_0$ or

$\frac{k}{\sqrt{t}} \alpha_0$

3. 

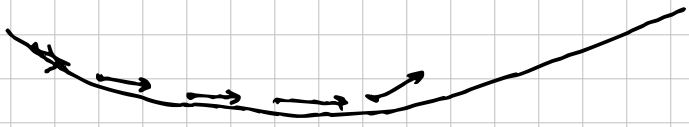
discrete staircase

4. Manual decay

The problem of local optima

Most likely local points are saddle points

Problem of plateaus



plateaus are parts of functions where the gradient is close to 0 for a long time

Notes:

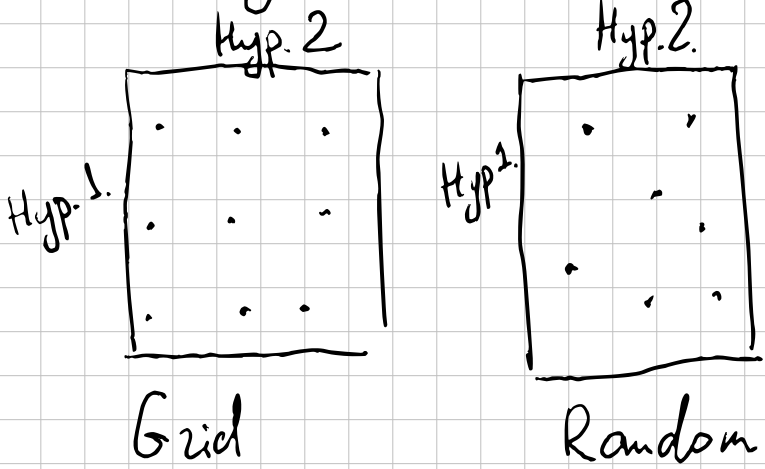
- Unlikely to get stuck in a bad local optima
- Plateaus can make learning slow

Hyperparameter Tuning

Hyperparameters

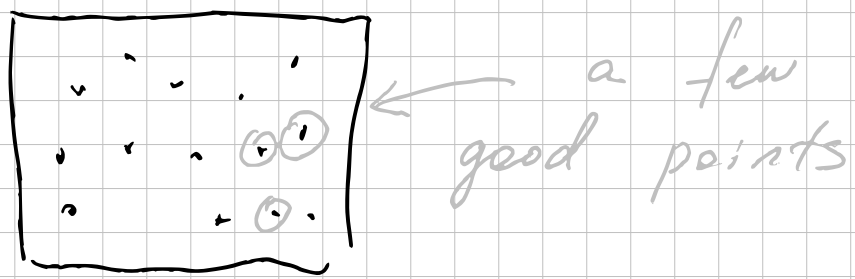
- learning rate (α)
- # hidden-unit
- mini-batch size
- momentum (β)
- learning-rate decay

- Try random values: don't use a grid



Grid Random

Coarse to fine

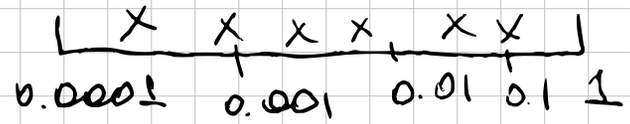


Zoom-in "good" region and perform thorough search there

Using an Appropriate Scale to pick hyperparameters

1. Picking hyperparameters at random

$\alpha = 0.0001$



$r = -4 \times np.random.rand()$

$\alpha = 10^r$

Sampling on a log scale

2. Hyperparameters for exponentially weighted averages

$\beta = 0.9 \ldots 0.999$

avg. over 10 el-ts ↓ 1000 el-ts ↓

We want to explore

$1-\beta = 0.1 \ldots 0.001$

uniformly random sampling

$r \in [-3, -1]$

$1-\beta = 10^r$

$\beta = 1 - 10^r$

Tips & tricks

2 ways to tune hyperparameters

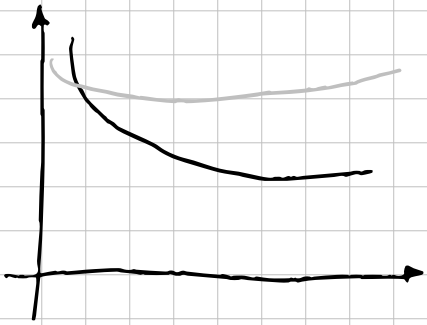
Babysitting one model

day 0 day 10 day 20

"Panda" approach

* One kid (trial)

Train many models in parallel



Model 1

Model 2

"Caviar" approach

* A lot of kids (trials)

** requires a lot of computational power

Normalizing activations in a network

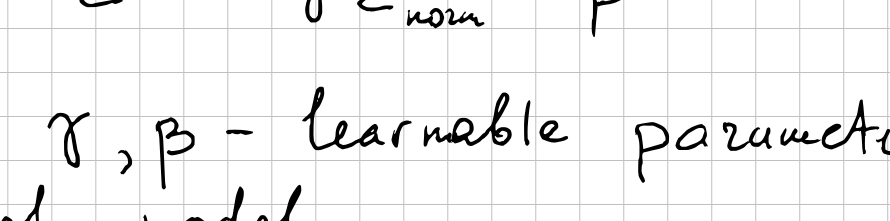
Regular normalization

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

$$X = X - \mu$$

$$G^2 = \frac{1}{m} \sum_i X^{(i)2}$$

$$X = X / G$$



Can we normalize $a^{[i]}$ so to train $w^{[i]}, b^{[i]}$ faster

Normalize $z^{[i]}$

(in practice this is done often)

Given some intermediate values in NN $\underbrace{z^{(1)}, \dots, z^{(m)}}_{z^{[l](i)}}$

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

$$G^2 = \frac{1}{m} \sum_i (z_i - \mu)^2$$

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

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

γ, β - learnable parameters of model

If $\gamma = \sqrt{G^2 + \varepsilon}$

$\beta = \mu$

then $\tilde{z}^{(i)} = z^{(i)}$

Use $\tilde{z}^{(i)}$ instead of $z^{(i)}$

Fitting Batch Norm to a network

Node computes 2 functions:

z, a.

$$X \xrightarrow{W^{[i]}, b^{[i]}} z \xrightarrow[\text{batch norm}]{\beta^{[i]}, \gamma^{[i]}} \tilde{z}^{[i]} \longrightarrow a^{[i]}$$

Normalised value

Parameters

$W^{[i]}, b^{[i]}$, $\beta^{[i]}$, $\gamma^{[i]}$

Working with mini-batches

$$X^{\{1\}} \xrightarrow{W^{[1]}, b^{[1]}} z^{[1]} \xrightarrow[BN]{\beta^{[1]}, \gamma^{[1]}} \tilde{z}^{[1]} \rightarrow a^{[1]} \rightarrow \dots$$

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

↙

Normalization will disregard any constant added to $z^{[l]}$ when will subtract mean, therefore if using Batch Norm parameter $b^{[l]}$ can be dropped

$$z^{[l]} = w^{[l]} a^{[l-1]}$$

$z^{[l]}$

$z^{[l]}_{norm}$

$$\tilde{z}^{[l]} = \gamma^{[l]} z^{[l]}_{norm} + \beta^{[l]}$$

$\beta^{[l]}$ $\gamma^{[l]}$

dims $(n^{[l]}, 1)$ $(n^{[l]}, 1)$

Implementing gradient descent

for $t = 1 \dots$ # mini-batches:

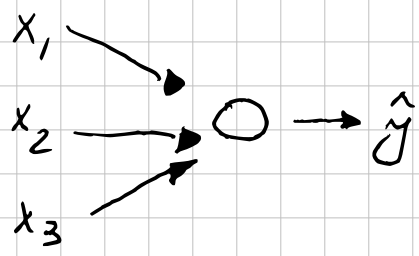
- Compute forward prop on $X^{\{t\}}$
  - In each hidden layer, use BN to replace $z^{[l]}$ with $\tilde{z}^{[l]}$
- Use backprop to compute $dW^{[l]}$, $d\beta^{[l]}$, $d\gamma^{[l]}$
- Update params

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

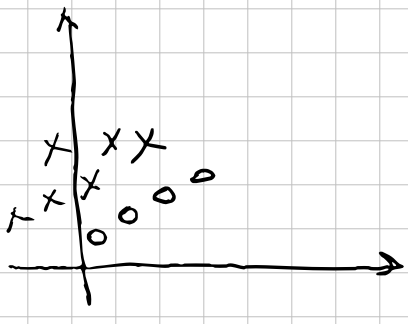
$$\beta^{[l]} := \beta^{[l]} - \alpha d\beta^{[l]}$$

$\gamma^{[l]}$ ...

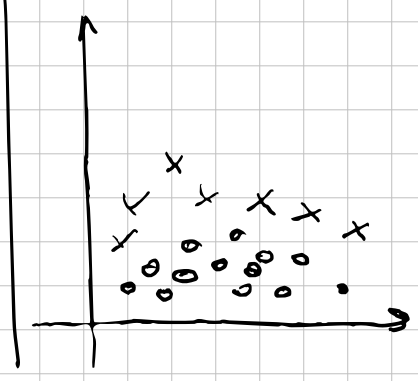
Why does Batch Norm work?



If you training set is



But dev set



You may generalize badly

This change is called "covariate shift"

* Consider a deeper network

Batch norm reduces the destribution: no mater how hidden layers parameters are changing the mean and variance stays the same.

Later layers of the NN to have more "stable ground"

- Each mini-batch is scaled by the mean/variance computed on just that mini-batch
- This adds some noise to the values $z^{[l]}$ within that mini-batch. So, similar to dropout, it adds some noise to each hidden layer's activations.
- This has a slight regularization effect.

Batch Norm at Test time

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

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

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

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

μ, σ^2 : estimate using exponentially weighted average (across mini-batches)

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

$\mu^{\{1\}[l]}, \mu^{\{2\}[l]}, \mu^{\{3\}[l]} \rightarrow \mu^{[l]}$

$\theta_1 \quad \theta_2 \quad \theta_3$

$\sigma^{2\{1\}[l]}, \sigma^{2\{2\}[l]}, \sigma^{2\{3\}[l]} \rightarrow \sigma^{2}$

*

$$z_{norm} = \frac{z - \mu}{\sqrt{\sigma^2 + \varepsilon}} \qquad \tilde{z} = \gamma z_{norm} + \beta$$

Softmax regression

Helps to detect multiple classes

$\hat{y}$ dim is $(4, 1)$

Softmax layer

In the last layer $\underline{\underline{l}}$

$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

- Activation function:

$$t = e^{(z^{[l]})} \quad (\text{dim is } (4,1))$$

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_i^4 t_i} = \frac{t_i}{\sum_i^4 t_i}$$

Understanding softmax

"hard max" gets max and considers it as 1 and everything else 0.

Softmax regression generalizes logistic regression to C classes.

* If $C = 2$, softmax reduces to logistic regression.

Loss function

$$y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \qquad \hat{y} = \begin{bmatrix} 0.3 \\ 0.2 \\ 0.1 \\ 0.4 \end{bmatrix}$$

$\hat{y}$ is probability

$$\mathcal{L}(\hat{y}, y) = -\sum_{i}^{4} y_i \log \hat{y}_i$$

to make it small is to make $\hat{y}_2$ bigger.

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

Backprop:

$$dz^{[1]} = \hat{y} - y$$

Orthogonalisation

Orthogonalisation referrs to a process of making hyperparameters independent from each other (i.e. change of one does not affect the other)

Chain of assumptions in ML

- Fit training set well on cost function
  ↓
- Fit dev set well on cost function
  ↓
- Fit test set well on cost function
  ↓
- Performs well in real world

Singhe number evaluation metric

Ex.

Classifier	Precision	Recall
A	95%	90%
B	98%	85%

Is B better than A?

It is not clear what of them is better

Posible solution: use F1-score as a combined metric for precision and recall

Classifier	F1
A	92.4%
B	91.%

* F1 is a harmonic mean $\left(\frac{2}{\frac{1}{P}+\frac{1}{R}}\right)$

Satisficing and Optimising metric

Ex.

Classifier	Accuracy	Running time
A	90%	80 ms
B	92%	95 ms
C	95%	1500 ms

One thing to do is to combine accuracy and running time in one metric

Another option: maximize accuracy as a subject to running time ($\leq$ 100 ms)

optimising — satisficing

Pick one as optimising and other metrics as satisficing

Train / Dev / Test distributions

dev - development / hold out cross-validation set

dev and test sets should come from the same distribution

You spend a lot of time optimising dev set (development time), but when it comes to test it may not generalise good (if dev and test are from different dist.)

Size of train/dev/test

98% 1% 1%
train dev test

for 10^6 examples

* Reasonable to use smaller sets for larger data sets.

Set your test set to be big enough to give high confidence in the overall performance of your system.

It is fine to do not have test set. (it is not recommended though)

Convolutional Neural Networks

Computer Vision Problems

- Image Classification

→ Cat? (0|1)

64x64

- Object detection

- Neural Style Transfer

Input in computer vision, usually, is <u>really big</u>.

Convolution operation

Egde detection

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

$6 \times 6 \times 1$ (grayscale)

$*$ "convolution"

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

3×3 filter/kernel

$=$

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

4×4

$3 \cdot 1 + 0 \cdot 0 + 1 \cdot -1$

$1 \cdot 1 + 5 \cdot 0 + 8 \cdot -1$

$2 \cdot 1 + 7 \cdot 0 + 2 \cdot -1$

python: conv-forward

tensorflow: tf.nn.conv2d

keras: Conv2D

Vertical edge detection examples

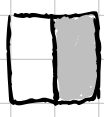
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0

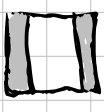
$*$

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

$=$

0	30	30	0
0	30	30	0
0	30	30	0
0	30	30	0





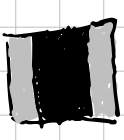
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

$*$

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

$=$

0	-30	-30	0
0	-30	-30	0
0	-30	-30	0
0	-30	-30	0



In this filter there is a difference between light-to-dark and dark-to-light transitions

If you dont care you may use absolute values.

Filters

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

Vertical

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

Horizontal

10	10	10	0	0	0
10	10	10	0	0	0
10	10	10	0	0	0
0	0	0	10	10	10
0	0	0	10	10	10
0	0	0	10	10	10

$*$

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

$=$

0	0	0	0
30	10	-10	-30
30	10	-10	-30
0	0	0	0

Learning to detect edges

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

Vertical filter

1	0	-1
2	0	-2
1	0	-1

Sobel filter

3	0	-3
10	0	-10
3	0	-3

Scharr filter

w_1	w_2	w_3
w_4	w_5	w_6
w_7	w_8	w_9

} We can treat filter values as parameters and learn them at backpropagation step

Generally learning a filter is more robust approach.

Padding

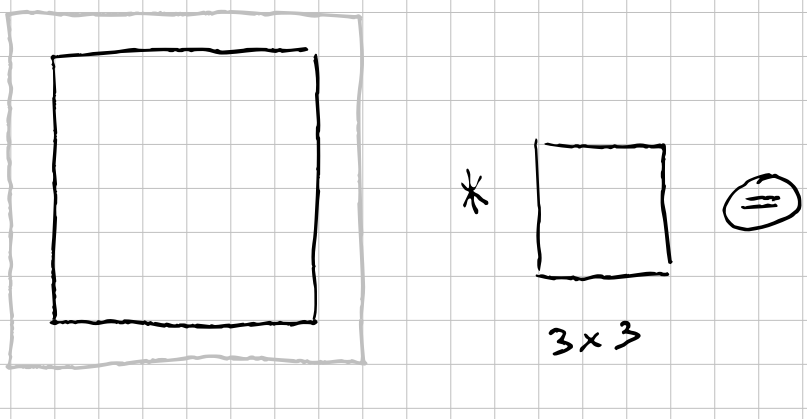
$n \times n \quad * \quad f \times f \quad = (n-f+1) \times (n-f+1)$

(image) (filter)

Problems:

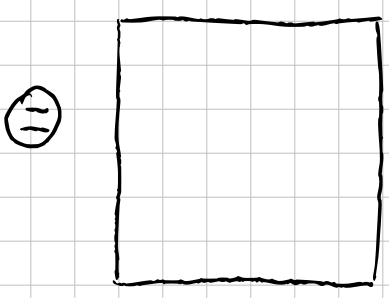
- Shrinking output
- Throwing away a lot of information

To solve it we can pad image



3×3

6×6 → 8×8



6×6 (instead of 4×4 for un-padded image)

by convention:

- padded values are 0
- p = padding = 1 (in this case)

Valid and Same Convolutions

"Valid" means no padding

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

$6 \times 6 \quad * \quad 3 \times 3 \rightarrow 4 \times 4$

"Same" means to pad so that output size is the same as the input size.

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

$n+2p-f+1 = n$

$2p = f-1$

$$p = \frac{f-1}{2}$$

By convention f is usually odd.

There are 2 reasons:

- If f is even you need an assymetric padding
- Even filters don't have central position

Strided convolution

Stride = 2

Step 1

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

7x7

$*$

3	4	4
-1	0	2
-1	0	3

3x3

$=$ 91

Step 2

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

7x7

$*$

3	4	4
-1	0	2
-1	0	3

3x3

$=$ 91 100

Step 3

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

7x7

$*$

3	4	4
-1	0	2
-1	0	3

3x3

$=$ 91 100 83

Step 4

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

7x7

$*$

3	4	4
-1	0	2
-1	0	3

3x3

$=$ 91 100 83 69

and so on

$n \times n \quad * \quad f \times f$

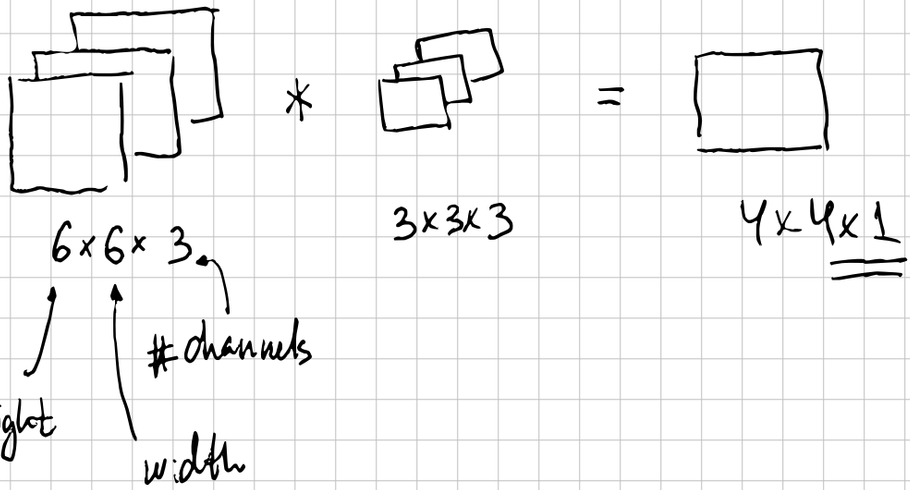
padding p stride s

$$\left\lfloor \frac{n+2p-f}{s} \right\rfloor + 1 \quad \otimes$$

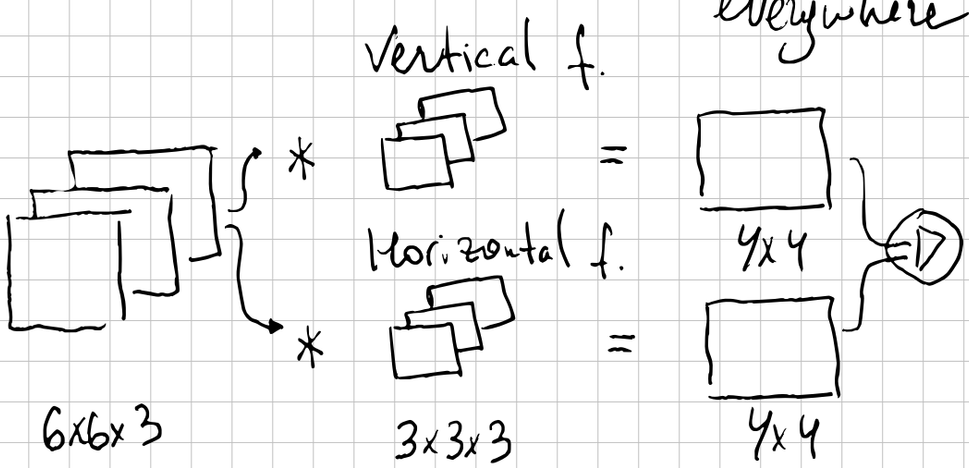
$$\otimes \left\lfloor \frac{n+2p-f}{s} \right\rfloor + 1$$

Convolution over volumes

Convolutions on RGB images

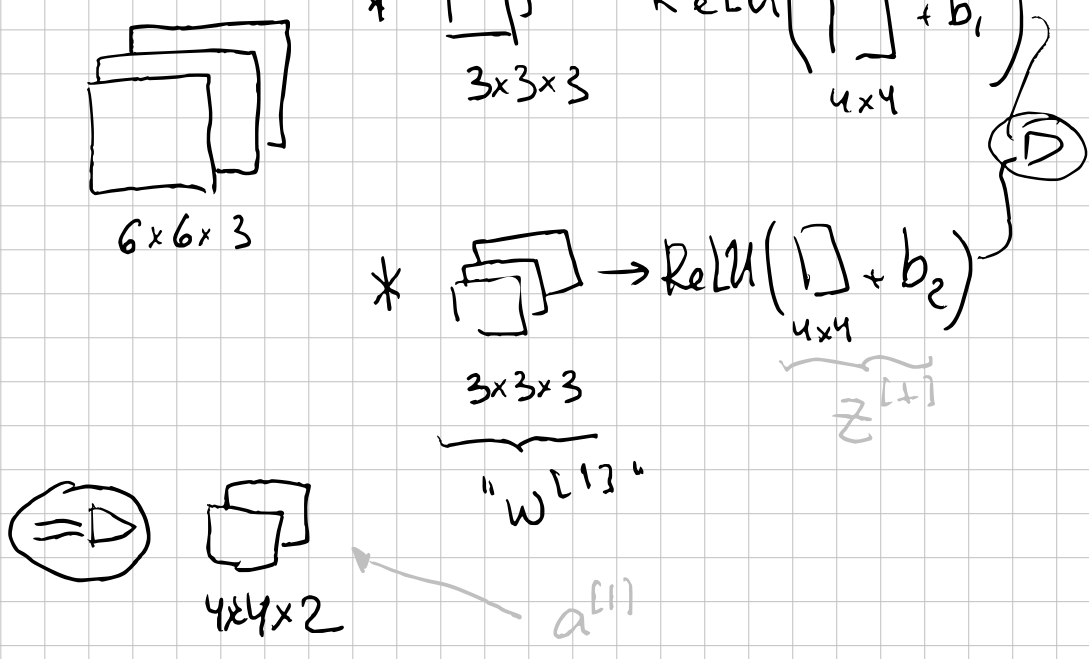


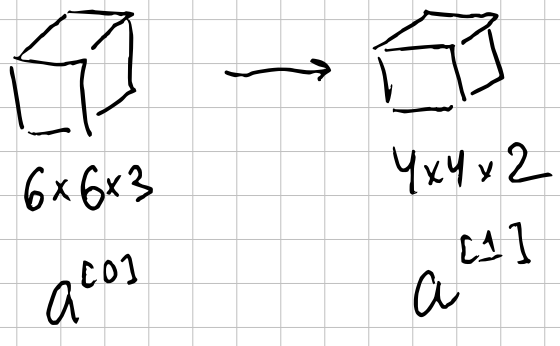
R	G	B	
1 0 -1 1 0 -1 1 0 -1	0 0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0 0	3×3×3 filter detects edges in red ch.
1 0 -1 1 0 -1 1 0 -1	1 0 -1 1 0 -1 1 0 -1	1 0 -1 1 0 -1 1 0 -1	3×3×3 filter detects edges everywhere



Sometimes people say depth instead of channels.

Example of a layer





Q: If you have 10 filters that are 3×3×3 in one layer of a neural network, how many parameters does that layer have?

3×3×3 + bias — × 10 times

28

A: 280 parameters

If layer l is a convolutional layer:

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

$p^{[l]}$ = padding

$s^{[l]}$ = stride

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

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

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

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

$n_C^{[l]}$ = number of filters

Each filter is: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]}$

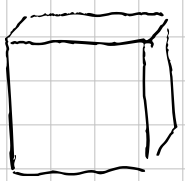
Activations: $a^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_C^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_C^{[l-1]} \times n_C^{[l]}$

($n_C^{[l]}$: number of filters in layer l)

Bias: $n_C^{[l]} - (1,1,1,n_C^{[l]})$

Simple Convolutional Network Examples



$39 \times 39 \times 3$

$n_H^{[0]} = n_W^{[0]} = 39$

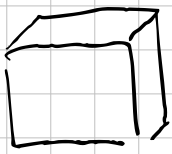
$n_C^{[0]} = 3$

$f^{[1]} = 3$

$s^{[1]} = 1$

$p^{[1]} = 0$

10 filters



$37 \times 37 \times 10$

$n_H^{[1]} = n_W^{[1]} = 37$

$n_C^{[1]} = 10$

$f^{[2]} = 5$

$s^{[2]} = 2$

$p^{[2]} = 0$

20 filters

$17 \times 17 \times 20$

$n_H^{[2]} = n_W^{[2]} = 17$

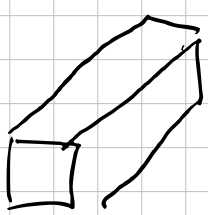
$n_C^{[2]} = 20$

$f^{[3]} = 5$

$s^{[3]} = 2$

$p^{[3]} = 0$

40 filters



$7 \times 7 \times 40$

unflatten into vector

1960 elements

logit / softmax → $\hat{y}$

Types of layers in CNN:

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

Pooling

Max pooling

1	3	2	1
2	9	1	1
1	3	2	3
5	6	1	2

$\longrightarrow$

9	2
6	3

4 x 4

Hyperparameters:

$f = 2$

$s = 2$

Take maximum in the given region.

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

5x5

$\longrightarrow$

$f = 3$

$s = 1$

9	9	5
9	9	5
8	6	9

3x3

If 3d then pooling preserves number of channels, i.e. pooling computation is done separately for every channel.

Average pooling

1	3	2	1
2	9	1	1
1	4	2	3
5	6	1	2

$\longrightarrow$

3.75	1.25
4	2

Hyperparameters:

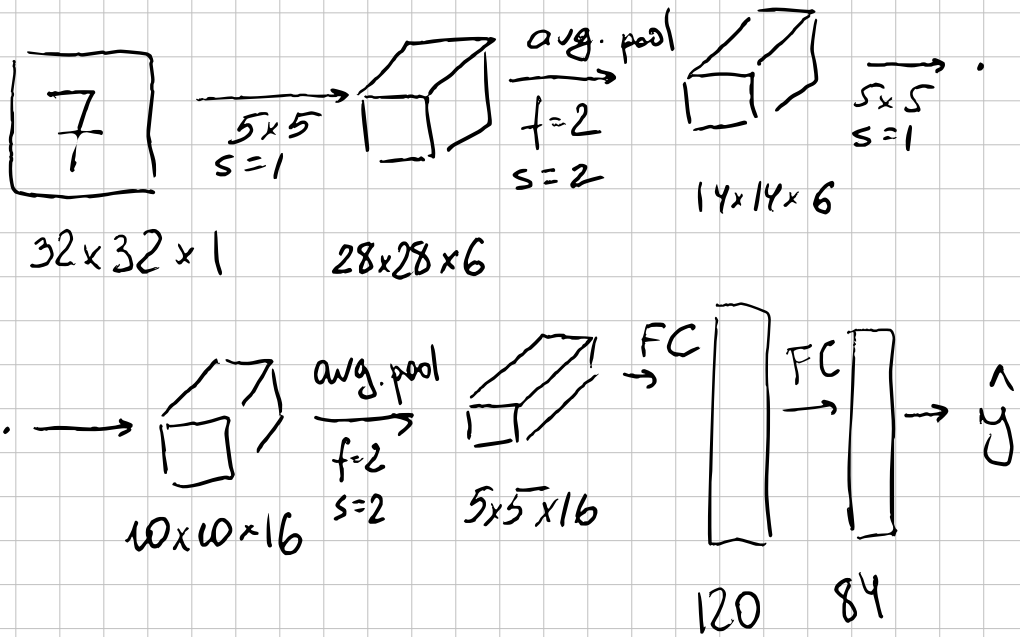
- f: filter size
- s: stride
- Max or average pooling
- p: padding (usually rarely used)

Input $n_H \times n_w \times n_C$ $\longrightarrow$ $\left\lfloor \frac{n_H - f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n_w - f}{s} + 1 \right\rfloor + n_C$

(assuming there is no padding)

Classic CNNs

LeNet-5

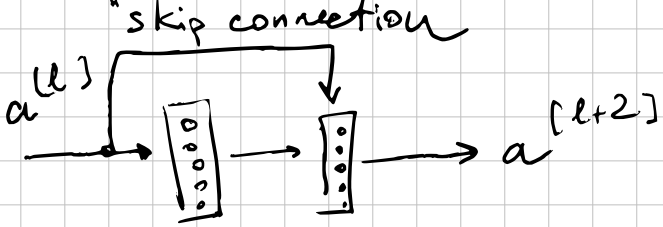


60k parameters

AlexNet

VGG-16

Residual network



All $a \geq 0$ and use ReLU

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

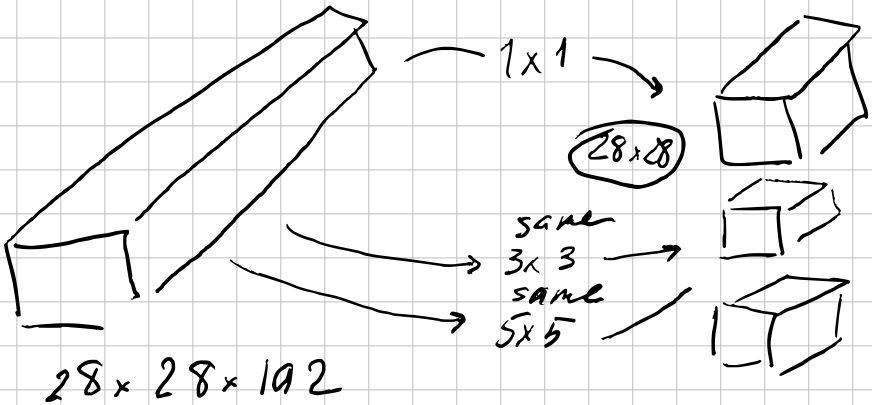
$$= g\left(w^{[l+2]}\, a^{[l+1]} + b^{[l+2]} + a^{[l]}\right)$$

If $w^{[l+2]} = 0$, $b^{[l+2]} = 0$

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

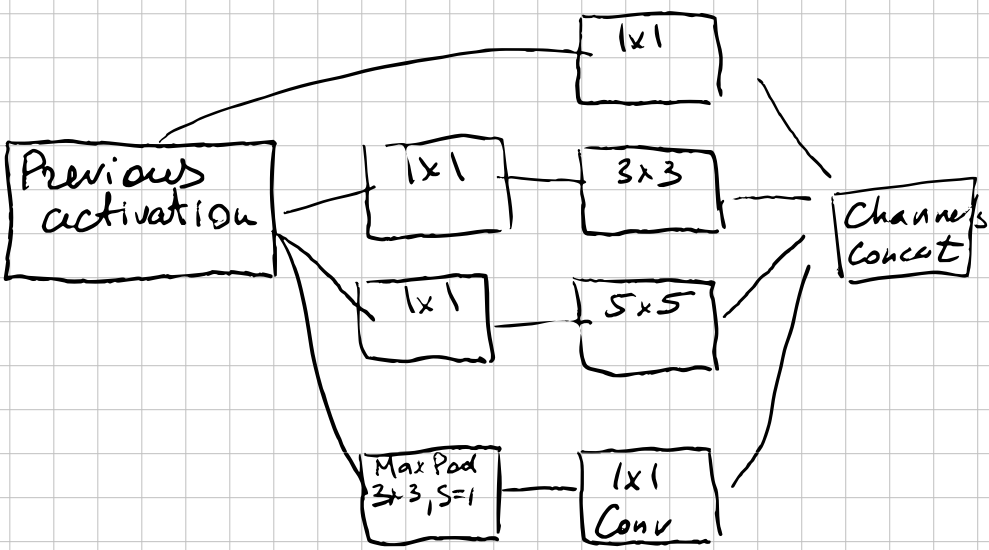
i.e. residual networks can learn to ignore the residual block and it does not hurt the network.

Inception network



Stack all of them together

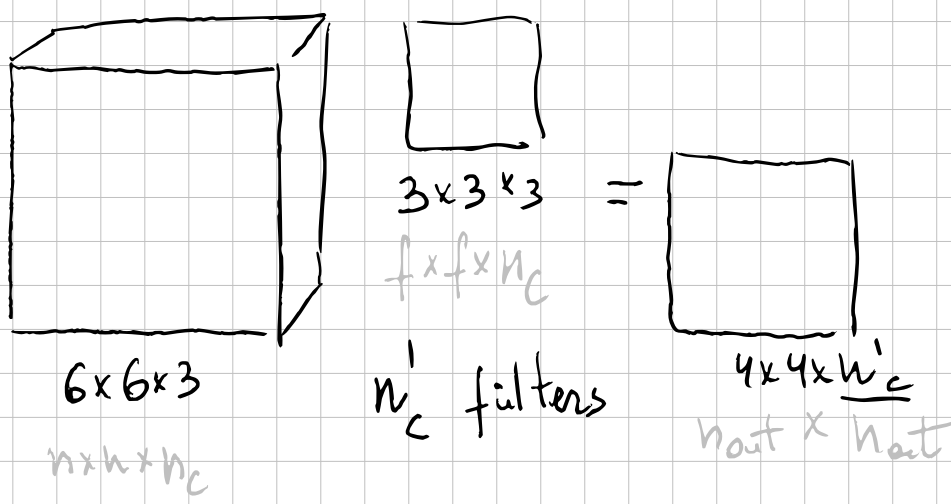
Inception module



MobileNet

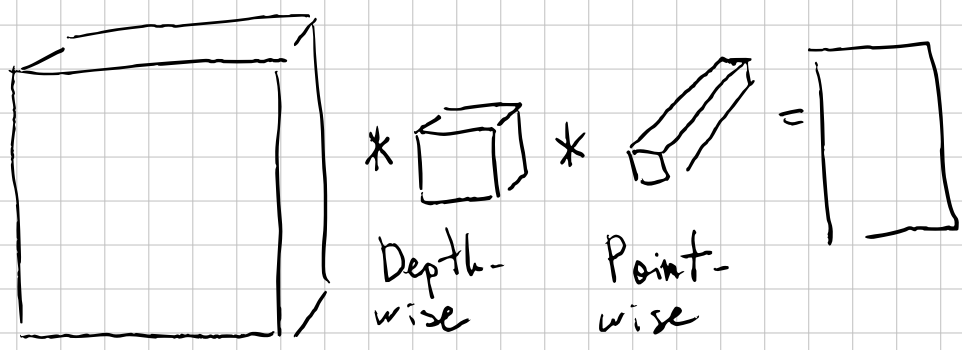
- Low computational cost at deployment
- Useful for mobile and embedded vision applications
- Key idea: Normal vs. depthwise-separable convolutions

Normal convolution

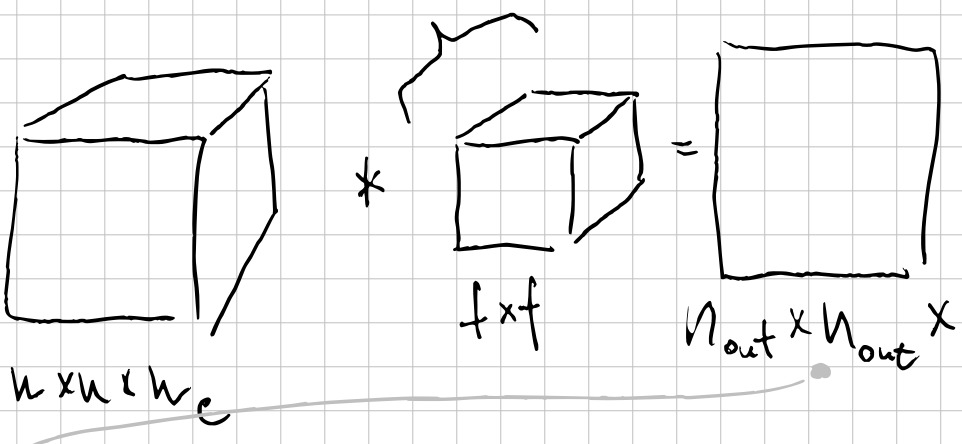


Computational cost: # filter params · # filter positions · # of filters

Depth-wise Separable Convolution



Depthwise conv n_c filters

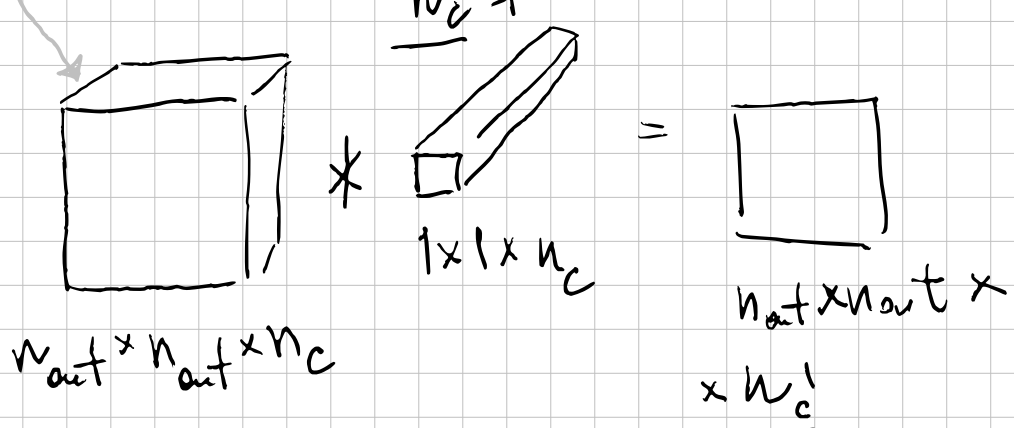


Apply every filter layer-by-layer to the corresponding layer

Computational cost = # filter params · # filter positions · # of filters

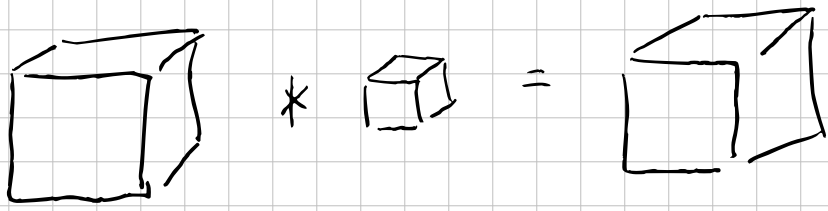
Pointwise conv.

n'_c filters

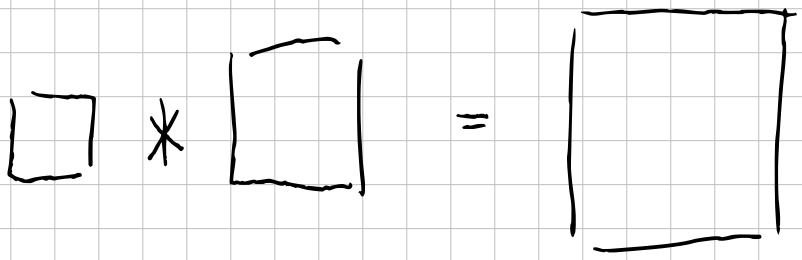


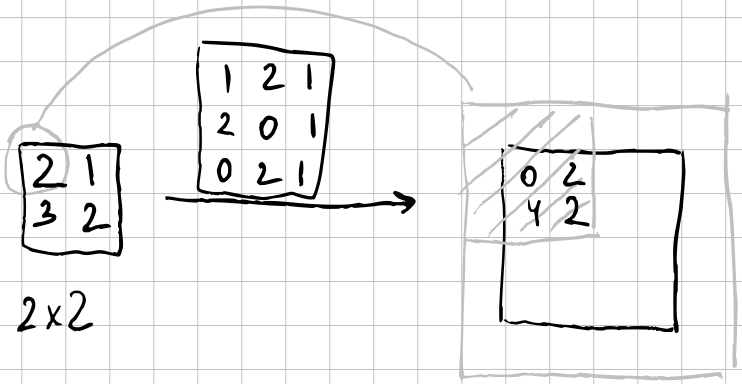
Transpose convolution

- Normal convolution



- Transpose convolution





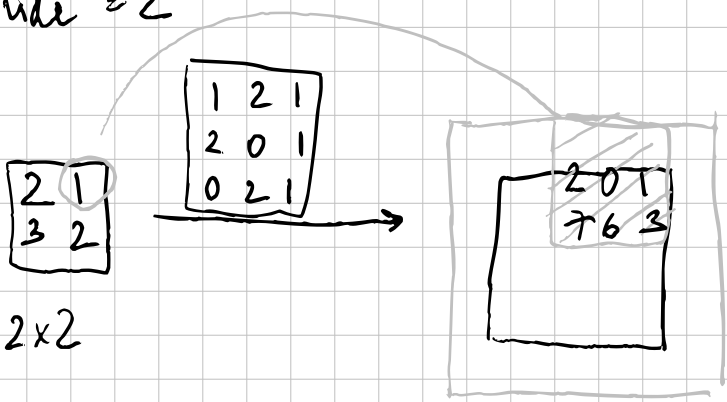
2x2

4x4

filter $f \times f = 3 \times 3$

padding = 1

stride = 2



2x2

4x4

Sequence models

$x^{<t>}$ index of a word in the sequence

T_y = length of the output sequence

T_x = input sequence length

$T_x^{(i)}$ = input sequence length of a i-th example

$$a^{<0>} = \vec{0}$$

$$a^{<1>} = g(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a)$$

$$\hat{y}^{<1>} = g(W_{ya} a^{<1>} + b_y)$$

The activation functions usually are: ReLU, tanh

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

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

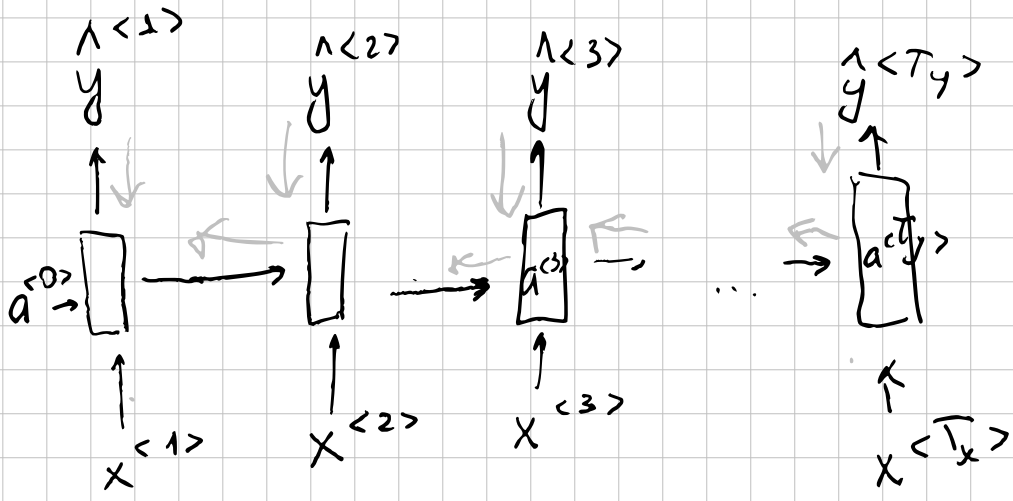
*alternative way to write:

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

$$[W_{aa}; W_{ax}] = W_a$$

$$[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ \hline x^{<t>} \end{bmatrix}$$

Backprop. in RNN



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

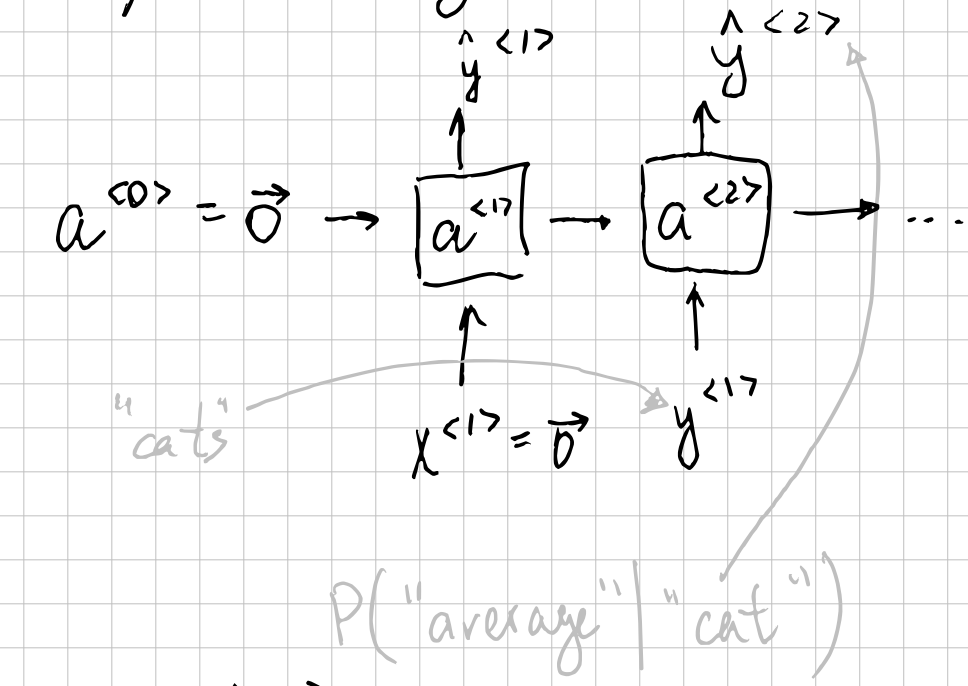
$$\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

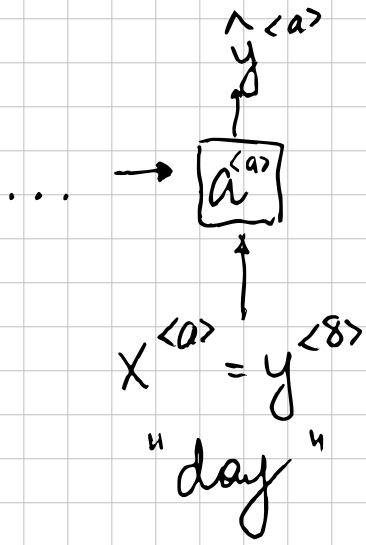
Language model and sequence generation

Training set: large corpus of English text.

$$x^{<t>} = y^{<t-1>}$$

Ex.
Cats average 15 hours of sleep a day. <EOS>

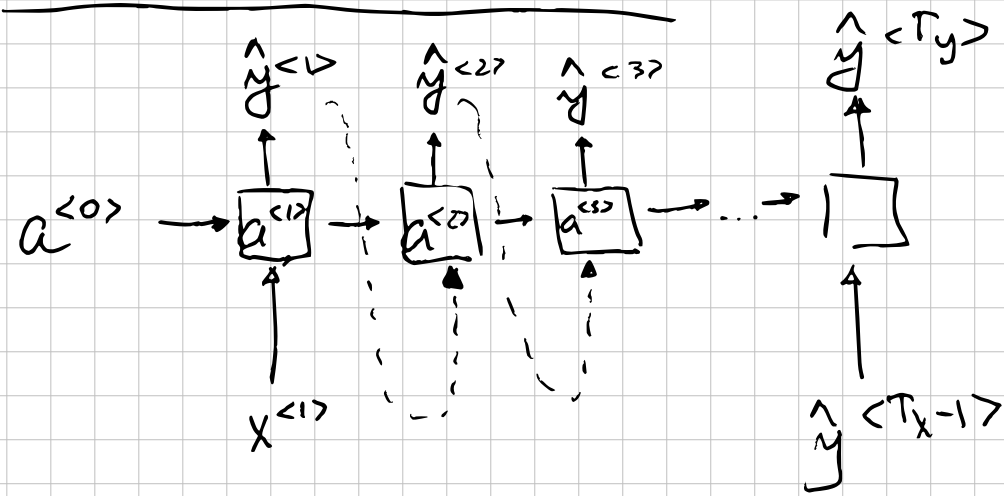




$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

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

Sampling a sequence from a trained RNN



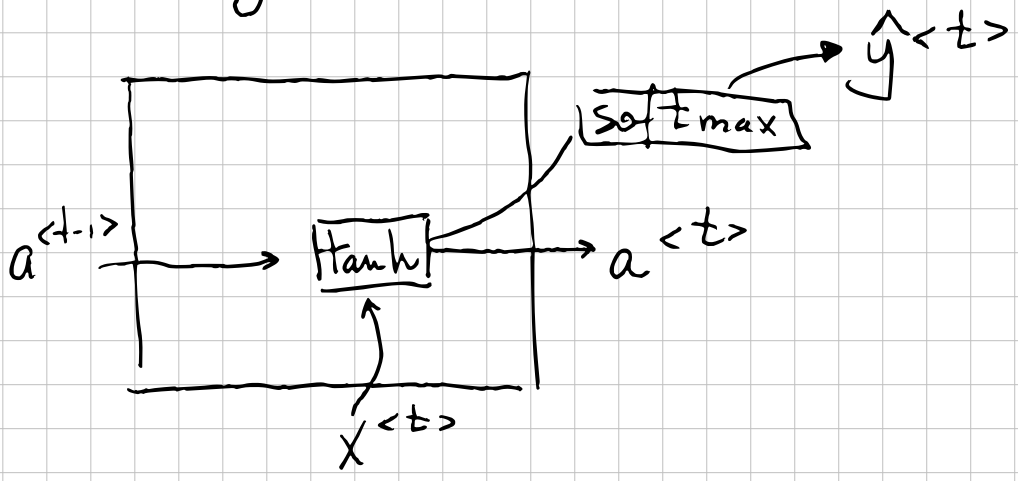
Vanishing gradients with RNNs

Gradient clipping: "fix" the gradient if the threshold been exceeded.

Gated Recurrent Unit

RNN unit

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



GRU

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

C = memory cell

$$\boxed{c^{<t>} = a^{<t>}}$$

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

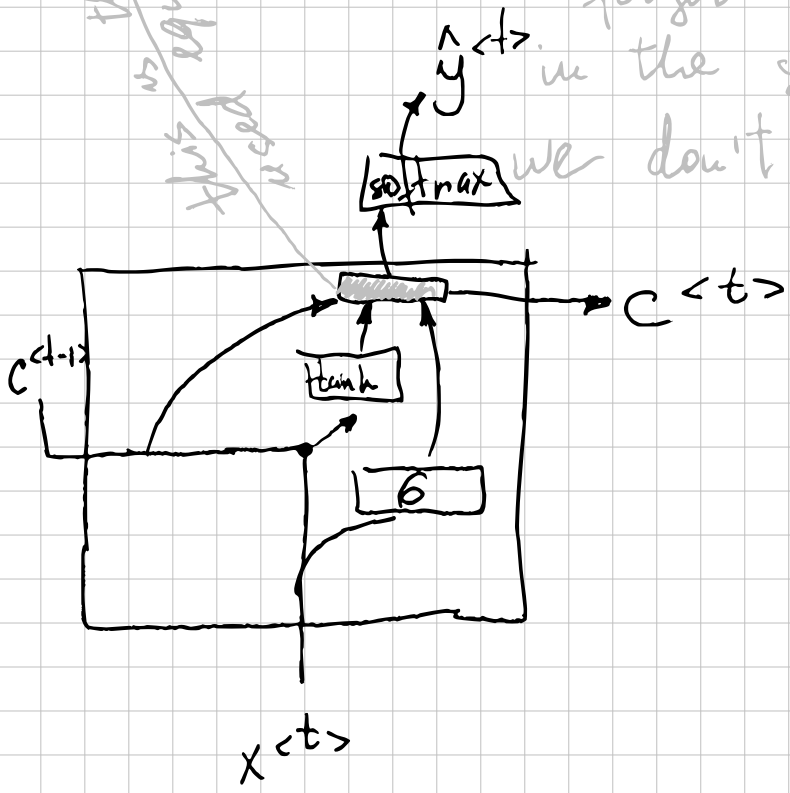
$$\Gamma_u = \sigma(w_u[c^{<t-1>}, x^{<t>}] + b_u)$$

↑ "u" stands for "update"

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$$

this is a vector

allows us to "forget" words in the sequence we don't need.



*Helps a lot with vanishing gradient problem

Full GRU

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

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

$$\Gamma_r = \sigma(w_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$$

NLP and word embeddings

Word representation

$V = [a, aaron, \ldots, zoo, <UNK>]$

1-hot representation

"Man" $(5391) = (0\ 0 \ldots 1 \ldots 0)$

5391

I want a glass of orange _____.

I want a glass of apple _____.

Words "orange" & "apple" can be far away from each other in 1-hot encoding which makes it really hard to generalize.

Featurized representation: word embedding

	Man	Woman	King	Queen	Apple	Orange
	5391	9853	4914	7157	456	6257
Gender	-1	1	-0.95	0.97	0.0	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.0
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
⋮						

t-SNE algorithm

Analogies

Man → Woman

King → ?

$e_{man} - e_{woman} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ← gender

$e_{king} - e_{queen} \approx \begin{bmatrix} -2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$ ← gender

$e_{man} - e_{woman} \approx e_{king} - e_?$

* Mikolov

Find word w:

$$\arg\max_w sim(e_w, e_{king} - e_{man} + e_{woman})$$

Usually cosine similarity is used to calculate similarity.

Embedding matrix

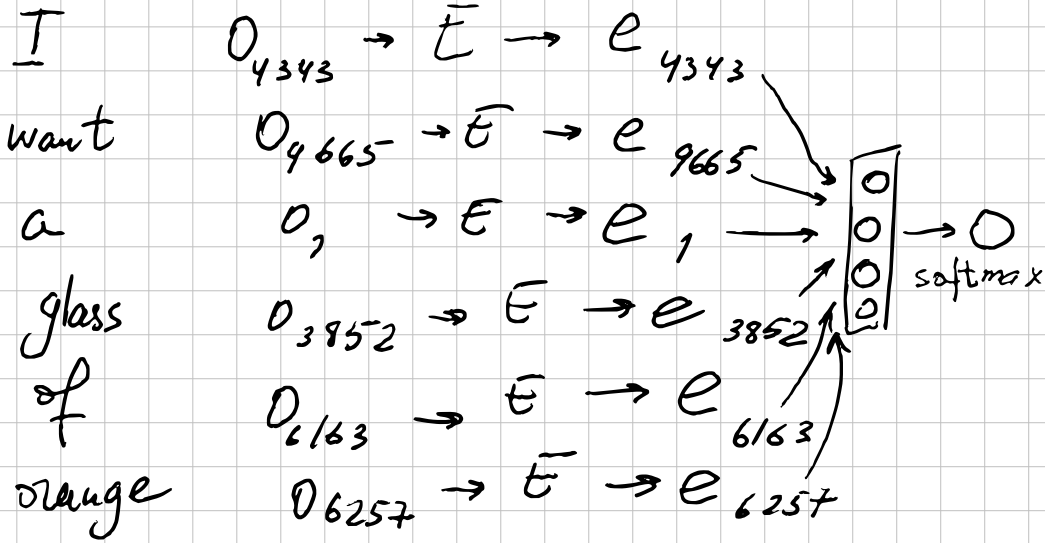
$$\begin{bmatrix} a & aaron & \dots & orange & \dots & zoo & \dots \end{bmatrix}$$

↑ 300 ↓

← 10 000 →

E - embedding matrix

Learning word2d embeddings



Skip gram model

Context: . Last 4 words
. 4 words on left & right
. Last 1 word
. Nearby word

Word 2Vec

I want a glass of orange juice to go along with my cereal.

Context (some random word)		Target
orange	→	juice
orange	→	glass
orange	→	my

Model

Vocal size = 10,000

Context c ("orange") → Target t ("juice")

$x \longrightarrow y$

O_c (one-hot vector for context)

Softmax: $p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$

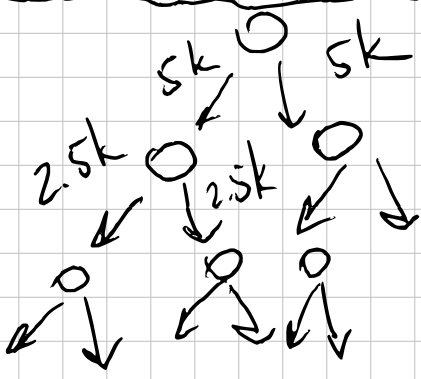
θ_t = parameter associated with output t

$\mathcal{L}(\hat{y}, y) = -\sum_{i=1}^{10000} y_i \log \hat{y}_i$

y - one-hot encoding vector

(* skip-gram model)

Can be really slow. Solution: use Hierarchical softmax



How to sample the context c?

*Use less frequent/common words.

Negative sampling

* Softmax is really "heavy"/ expensive to calculate

Given a pair of words

orange → juice

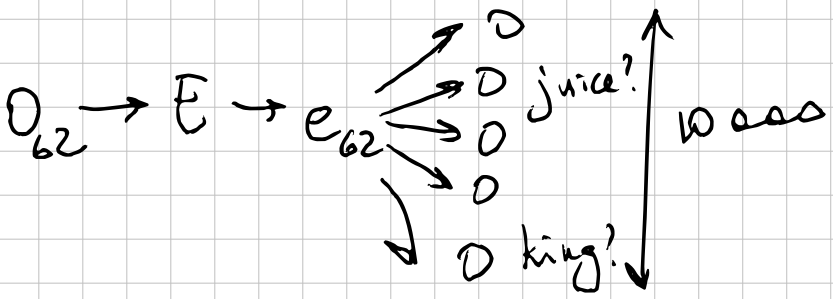
Is it a pair of context - target?

orange - juice	1
orange - king	0
orange - book	0
orange - the	0
orange - of	0

pick second word rand. from the vocabulary

$$p(t|c) = \frac{e^{\theta_t^T e_c}}{\sum_{j=1}^{10000} e^{\theta_j^T e_c}}$$

$$P(y=1 \mid c, t) = \sigma(\theta_t^T e_c)$$



10,000 binary classification problem which is <u>much</u> faster

How to choose negative examples?

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{10000} f(w_j)^{3/4}}$$

GloVe word vector

Global Vectors for word representation

c, t

X_{ij} = # times j appears in context of i

i ↑ c

j ↑ t

minimize the difference

$$\sum_{i=1}^{10000}\sum_{j=1}^{10000} f(X_{ij})\left(\theta_i^T e_j + b_i + b_j' - \log X_{ij}\right)^2$$

$f(X_{ij})$ ↓ weight term

b_i ↑ t

b_j' ↑ c

Sequence to sequence model

$x^{<1>}$ $x^{<2>}$

Jane visite l'Afrique en septembre

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

Jane is visiting Africa in September

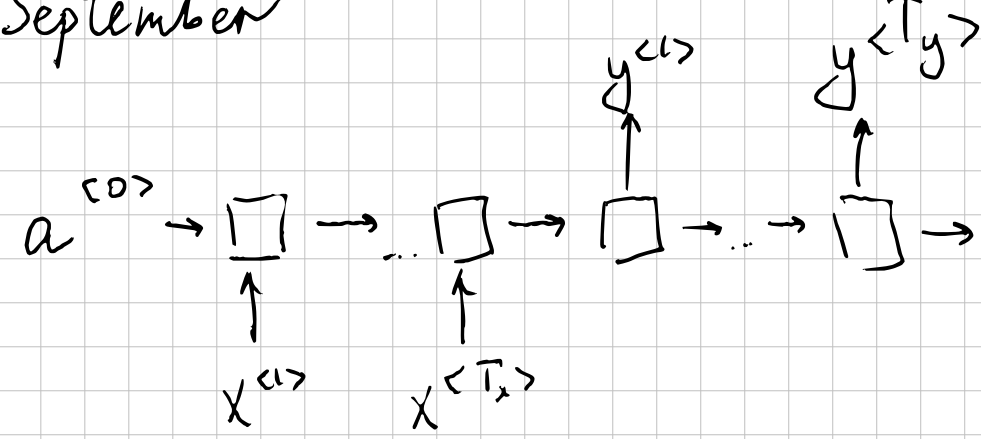
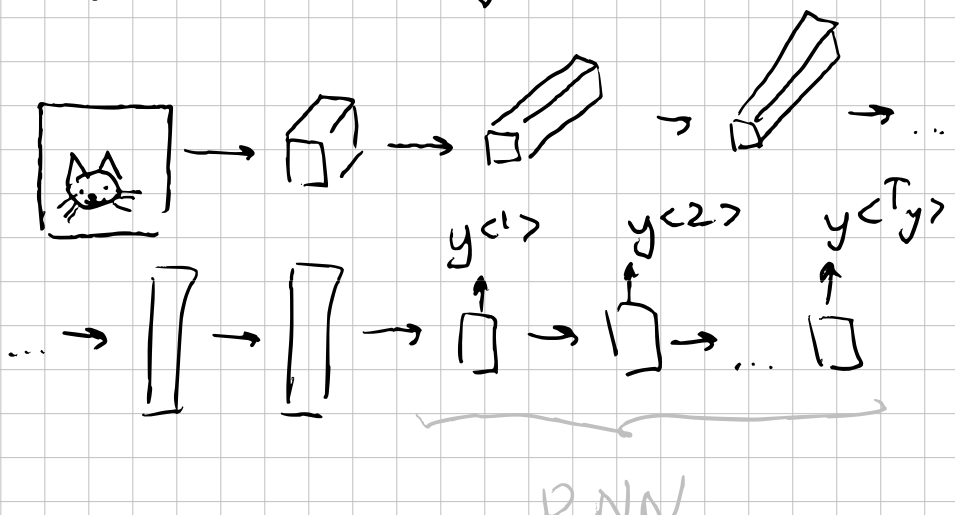
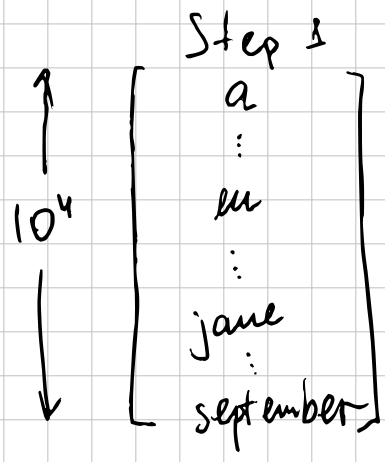
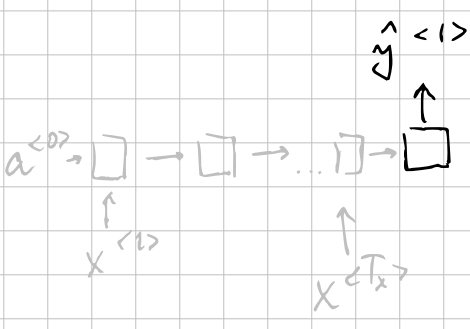


Image captioning

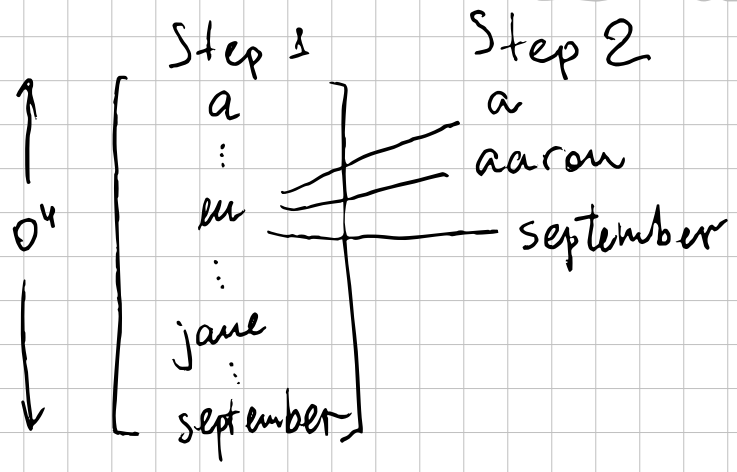


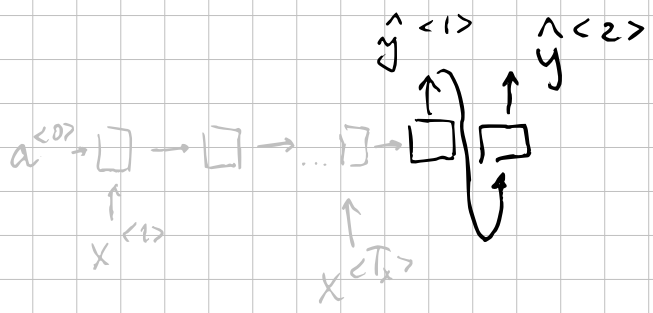
Beam Search





$$P(y^{<1>} | x)$$





$$P(y^{<1>}, y^{<2>} | x) = P(y^{<1>} | x) \cdot P(y^{<2>} | x, y^{<1>})$$

Refinements to beam search

Length normalization

- $\arg\max_y \prod_{t=1}^{T_y} P(y^{<t>} | x, y^{<1>}, \cdots, y^{<t-1>})$

numbers less than 1

multiplication of a plenty of small numbers result in underflow

- $\arg\max_y \sum_{y=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \cdots, y^{<t-1>})$

Normalization by the number of translation

- $\frac{1}{T_y^{\alpha}} \sum_{t=1}^{T_y} \log P(y^{<t>} | x, y^{<1>}, \cdots, y^{<t-1>})$

Usually, $\alpha = 0.7$

$\alpha = 1$

Tips

Beam width B?

large B: consider more possib., better results, slower

smaller B: worse result, faster

$B \in [10, 100, 1000, 3000]$

Unlike BFS/DFS, Beam Search runs faster but not guaranteed to find exact minimum.

Error Analysis in Beam search

Jane visite l'Afrique en septembre.

Human: Jane visits Africa in September (y^*)

Algo: Jane visited Africa last September ($\hat{y}$)

RNN computes $P(y|x)$

Lets compute $P(y^*|x)$ and $P(\hat{y}|x)$

Case 1: $P(y^*|x) > P(\hat{y}|x)$

Beam search chose $\hat{y}$. But y^* attains higher $P(y|x)$.

Conclusion: Beam search is at fault

Case 2: $P(y^*|x) \leq P(\hat{y}|x)$

y^* is a better translation than $\hat{y}$. But RNN predicted $P(y^*|x) \leq P(\hat{y}|x)$

Conclusion: RNN model is at fault.

Bleu Score

How to evaluate machine translation if there are multiple equal translations?

Bleu - bilingual evaluation understudy

Le cat est sur le tapis.

Ref. 1: The cat is on the mat.
Ref. 2: There is a cat on the mat.

MT output: the the the the the the.

Precision: $\frac{7}{7}$

Modified precision: $\frac{2}{7}$ Count clip ("the")

*MT output: The cat the cat on the mat.

	Count	Count clip
the cat	2	1
cat the	1	0
cat on	1	1
on the	1	1
the mat	1	1

Bigrams precision: $\frac{\sum \text{Count clip}}{\sum \text{Count}} = \frac{4}{6}$

Bleu score on unigrams

$$p_1 = \frac{\sum_{\text{unigrams}} \text{Count}_{\text{clip}}(\text{unigram})}{\sum_{\text{unigram}} \text{Count}(\text{unigram})}$$

$$p_n = \frac{\sum_{n\text{-grams}} \text{Count}_{\text{clip}}(n\text{-gram})}{\sum_{n\text{-grams}} \text{Count}(n\text{-grams})}$$

Bleu details

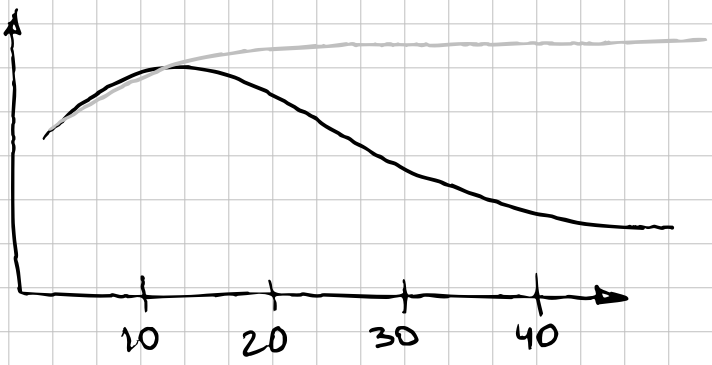
p_n - bleu score on n-grams only

Combined Bleu score:

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

Attention model

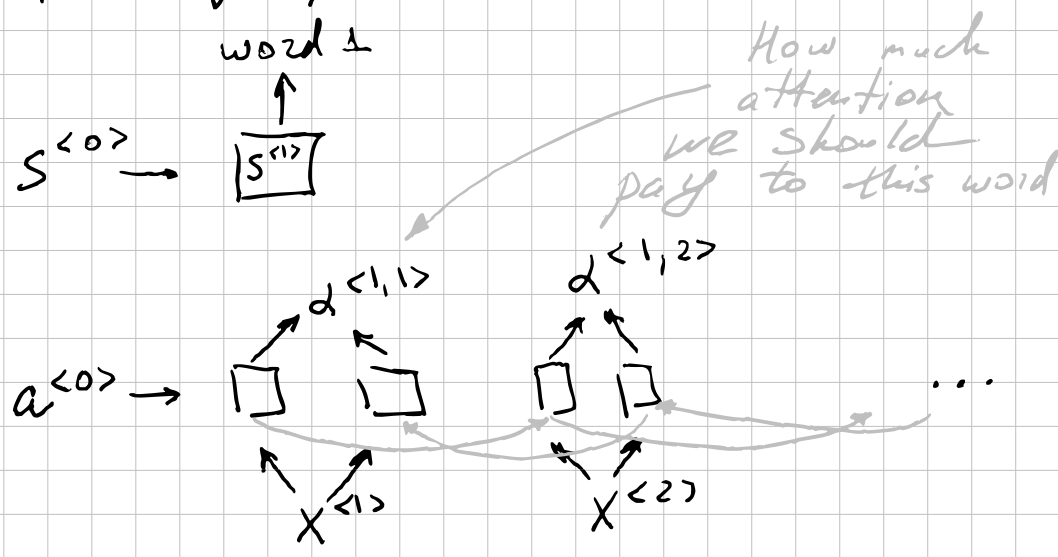
Encoder-decoder architecture



encoder-decoder model

attention model

Attention model processes the input by parts.

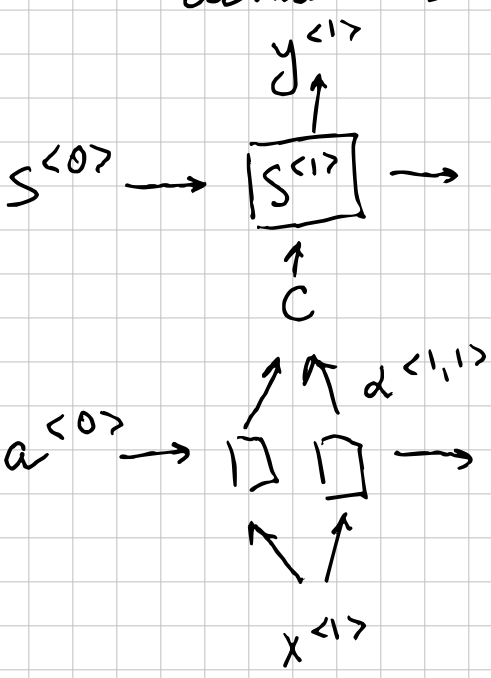


for word 2 we will generate $\alpha^{<2,1>}, \alpha^{<2,2>}$ ....

for word 3 : $\alpha^{<3,1>}, \alpha^{<3,2>}$ ...

$$a^{<t>} = \left(\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>} \right)$$

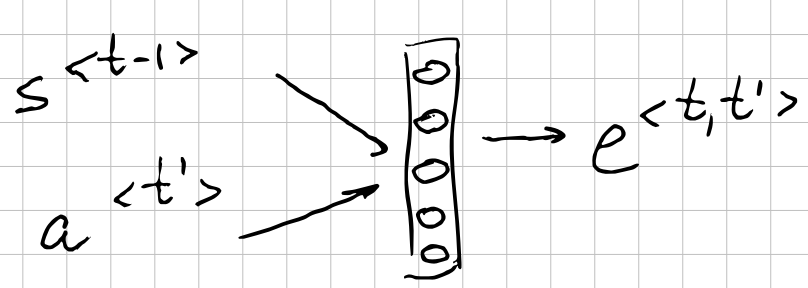
forward activations — backward activations



$\alpha^{<t,t'>}$ = amount of "attention" $y^{<t>}$ should pay to $a^{<t'>}$

$c^{<t>}$ = context vector

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

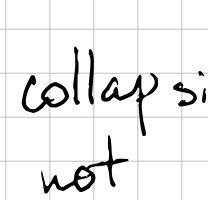


Complexity: $O(T_x T_y)$

Speech recognition

X → Y

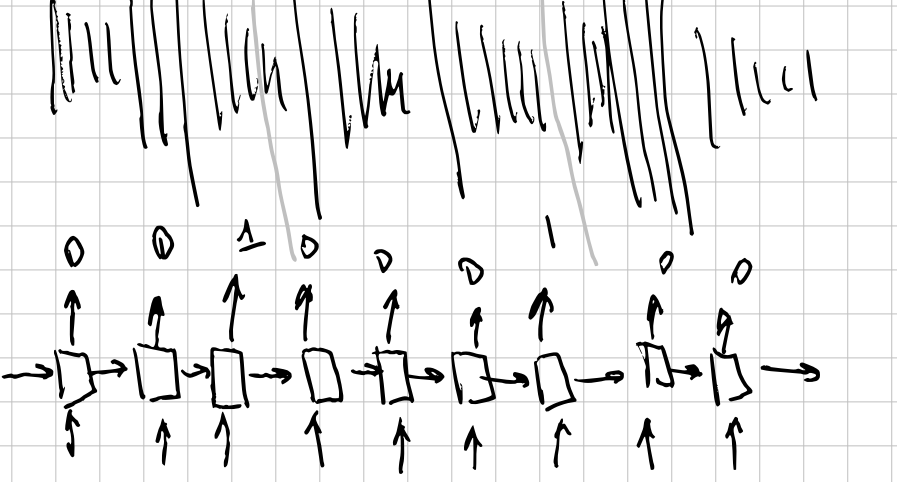
X: audio clip



Y: "the quick brown fox"

CTC cost for speech recognition

(connectionist temporal classification)



CTC allows to generate special output

ttt_h_eee__␣__q__

"blank"

"space"

collapsing repeating characters not separated by blank.

the␣q

Trigger Word Detection

Transformers

- Attention + CNN
  - Self-attention
  - Multi-Head Attention

Self-Attention

$A(q, K, V)$ = attention-based vector representation of a word

↳ Calculate for each word $A^{<1>}, A^{<2>} \ldots A^{<3>}$

$x^{<1>}$	$x^{<2>}$	$x^{<3>}$	$x^{<4>}$	$x^{<5>}$
Jane	visite	l'Afrique	en	septembre

Transformers Attention

$$A(q, K, V) = \sum_i \frac{\exp(q \cdot k^{<i>})}{\sum_j \exp(q \cdot k^{<j>})} v^{<i>}$$

$x^{<3>}$ l'Afrique

1. Associate each of the words with Query, key, Value.

$(q^{<3>}, k^{<3>}, v^{<3>})$

$q^{<3>} = W^Q x^{<3>}$

$k^{<3>} = W^K x^{<3>}$

$v^{<3>} = W^V x^{<3>}$

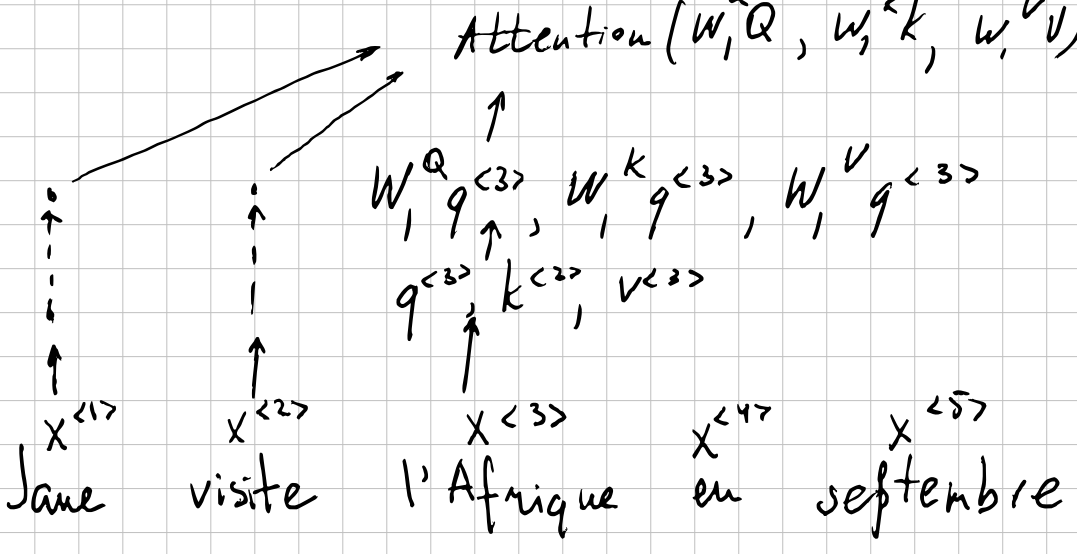
$q^{<3>}$ is a question you ask about the subject

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

factor to scale dot-product so it won't get explode

Multi-Head Attention

MultiHead(Q, K, V)



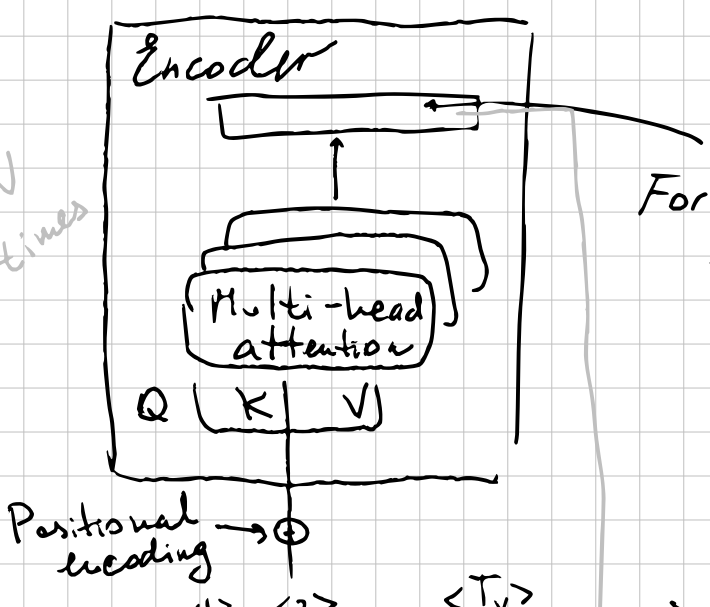
Instead of W_1^Q, W_1^K, W_1^V we will use W_i^Q, W_i^K, W_i^V to be able to address multiple questions.

$\text{MultiHead} = \text{concat}(\text{head}_1, \text{head}_2 \ldots \text{head}_h) W_0$

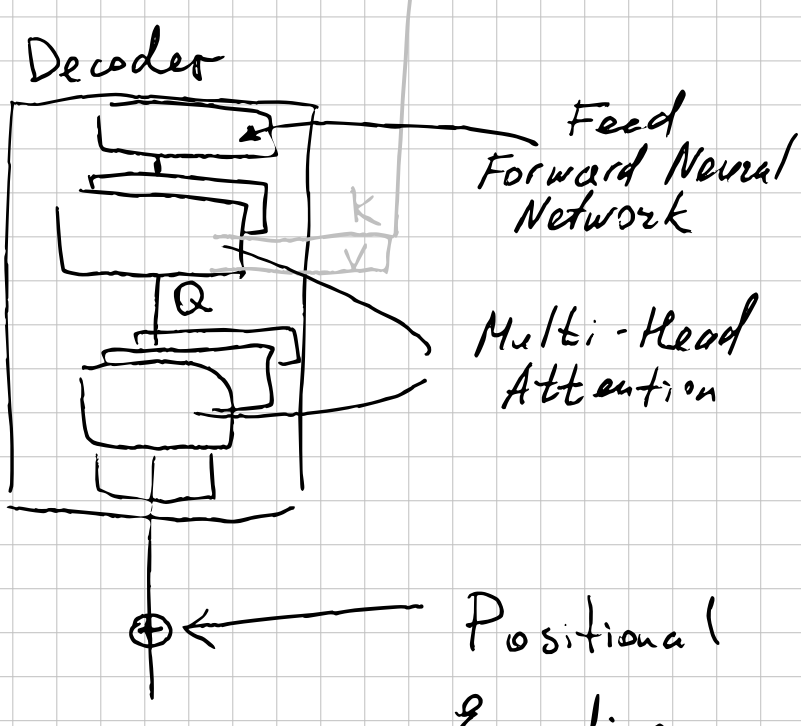
$\text{head}_i = \text{Attention}(W_i^Q Q, W_i^K K, W_i^V V)$

Transformer network

1. Embeddings get fit into encoder



2.



Pos. encoding

$$PE_{(pos, 2t)} = \sin\left(\frac{pos}{10000^{\frac{2t}{d}}}\right)$$

$$PE_{(pos, 2t+1)} = \cos\left(\frac{pos}{10000^{\frac{2t}{d}}}\right)$$

Add & Norm

Similar to Batch Norm

Masked Multi-Head Attention

Helps during training: blocks out parts of the sentence

Long Short Term Memory

(LSTM)

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

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

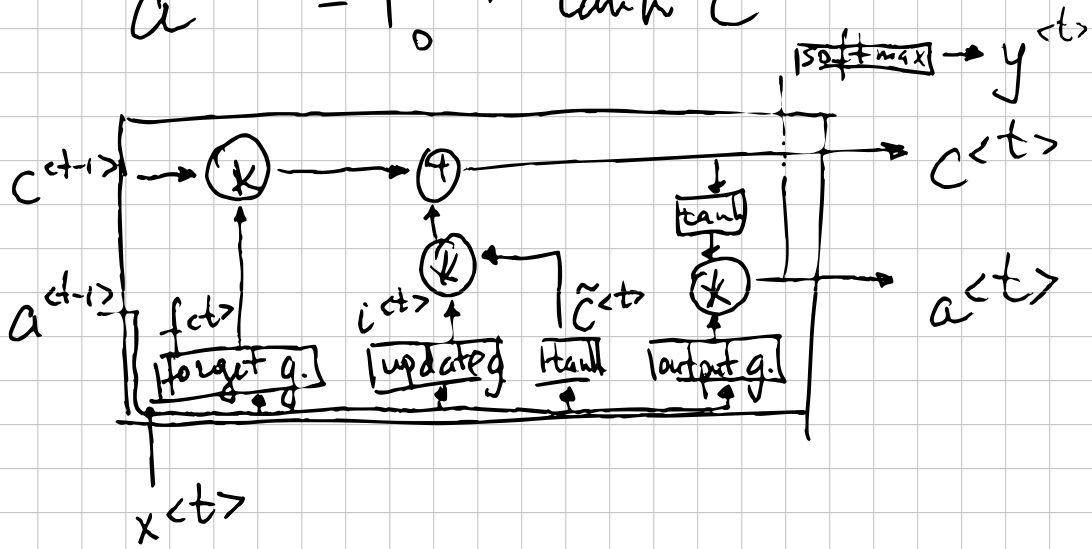
$$F_f = \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right)$$

("forget" gate

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

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

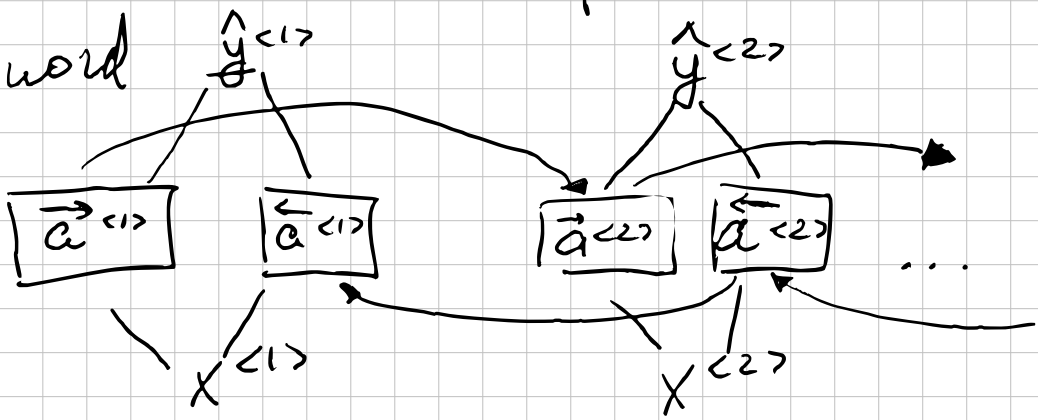
$$a^{<t>} = \Gamma_o * \tanh c^{<t>}$$



Bidirectional RNN

- He said, "Teddy Roosevelt was a great president."
- He said, "Teddy bears are on sale"

It is not enough to take a look just at the word (without looking ahead) to determine the semantic of the word

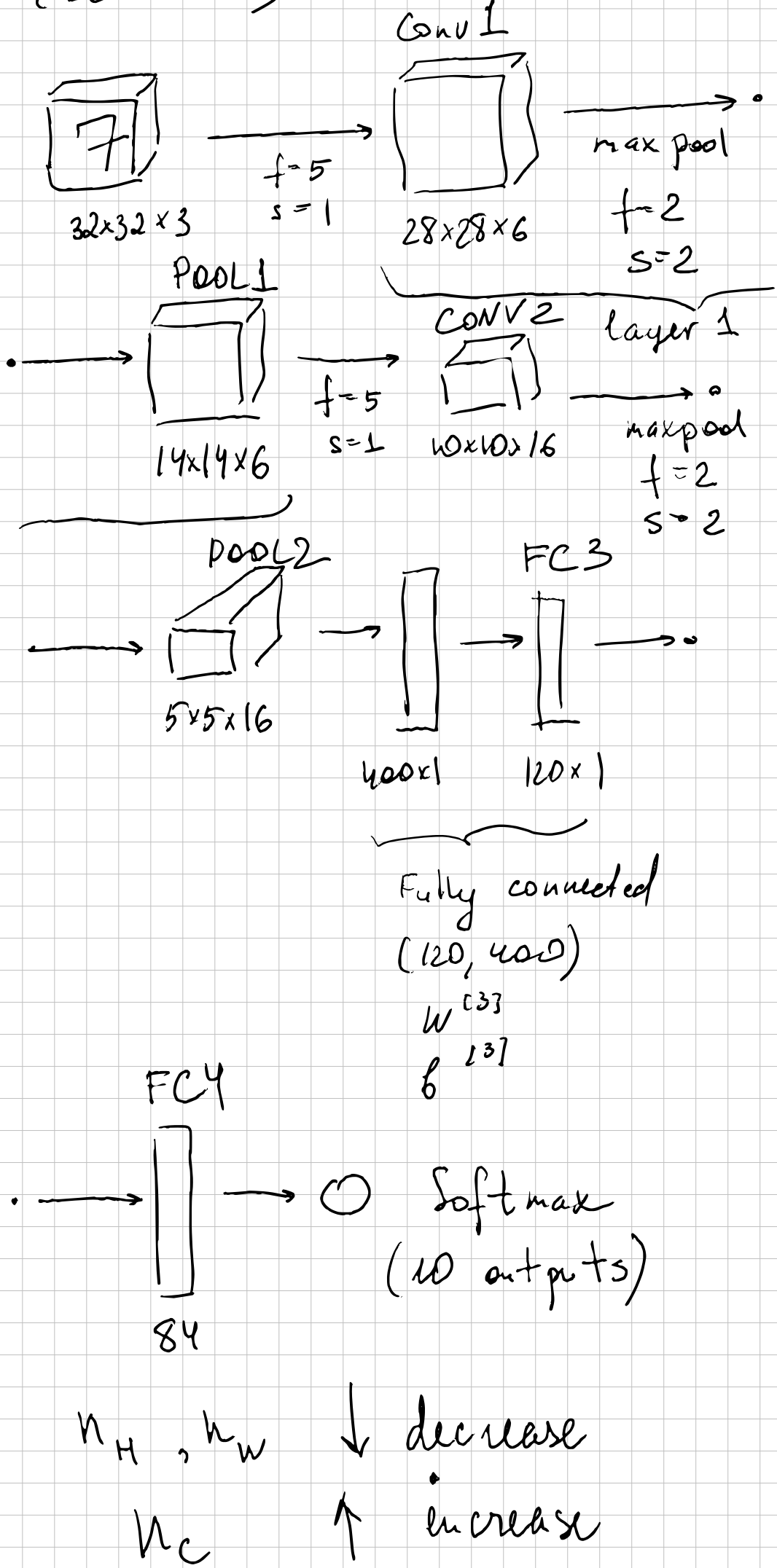


$$\hat{y}^{<t>} = g\left(W_y \left[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}\right] + b_y\right)$$

Acyclic graph

Neural network example

(LeNet-5)



n_H, n_W ↓ decrease

n_C ↑ increase

Why convolutions?

- FC have a lot of parameters, even be unfeasibly large

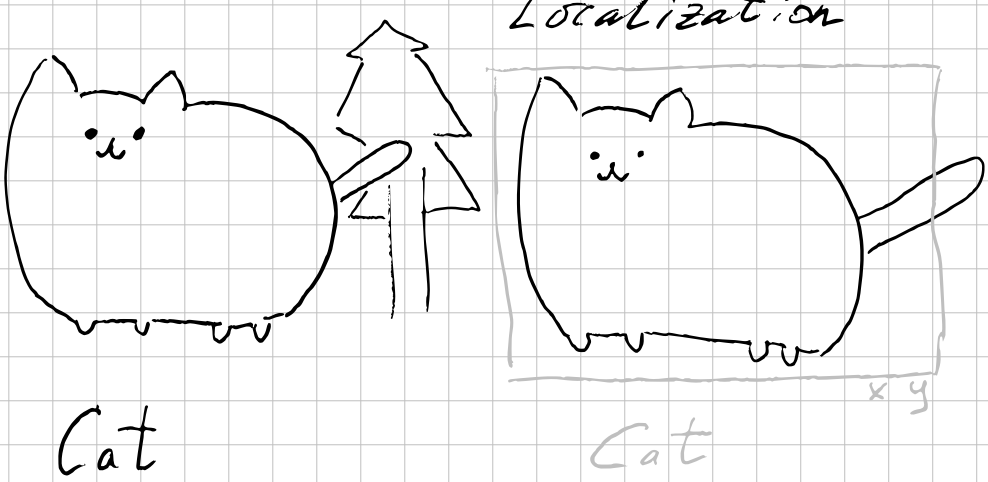
- Parameter sharing: a feature detector (such as vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.

- Sparsity of connections: In each layer, each output value depends only on a small number of inputs.

Object detection task

Classification

Classification + Localization



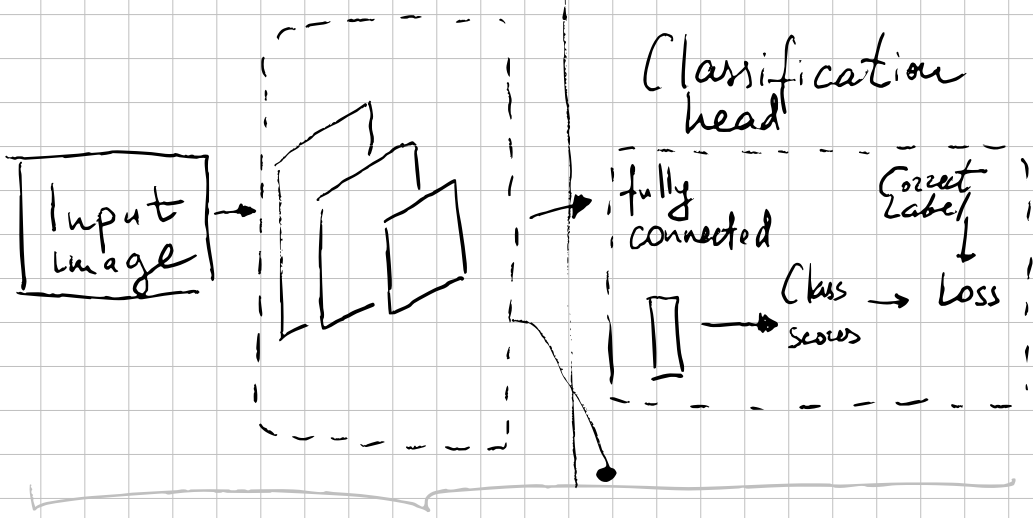
Cat

Cat

What and where the single object exist in an image

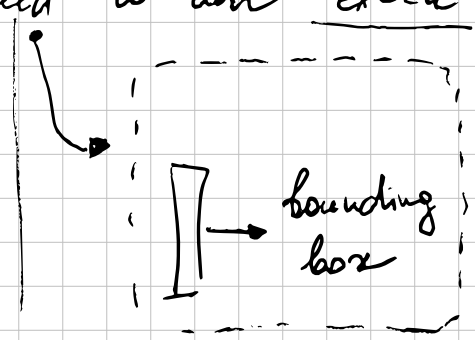
Basic classification

CNN



Classification

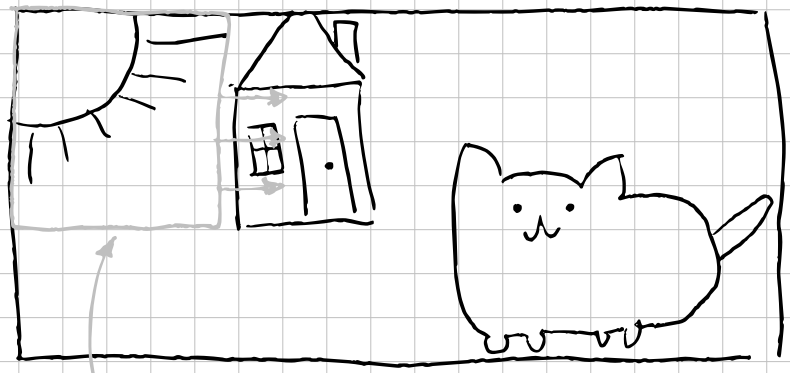
For localization we need to add extra step



⊕ localization

Sliding window approach

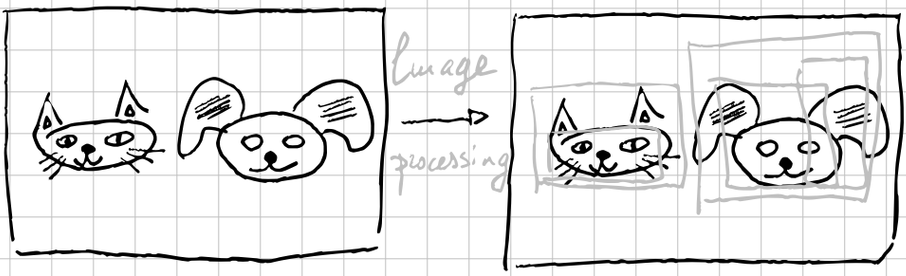
We take an area of the image of a fixed size and "slide" it over the image testing every position independently



Computationally expensive to cover all possible cases (locations, scales, aspect ratios)

Region-based approaches

- Find some image regions where object can be potentially found

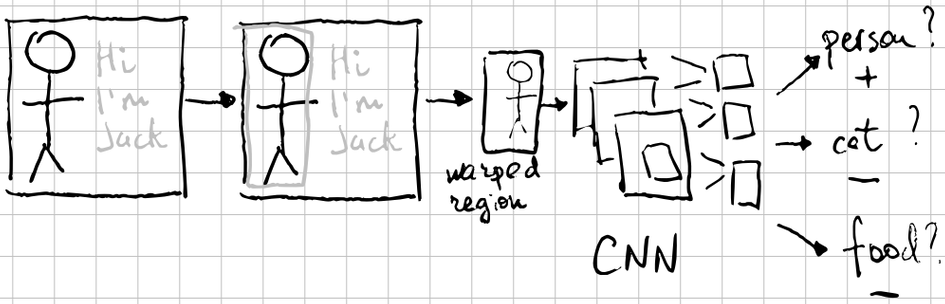


- Relatively fast

R-CNN

Pipeline

① Generate region proposals with selective search
② Resize each region proposal
③ Predict class probabilities with linear SVM (including background class)
④ Predict bounding box refinements

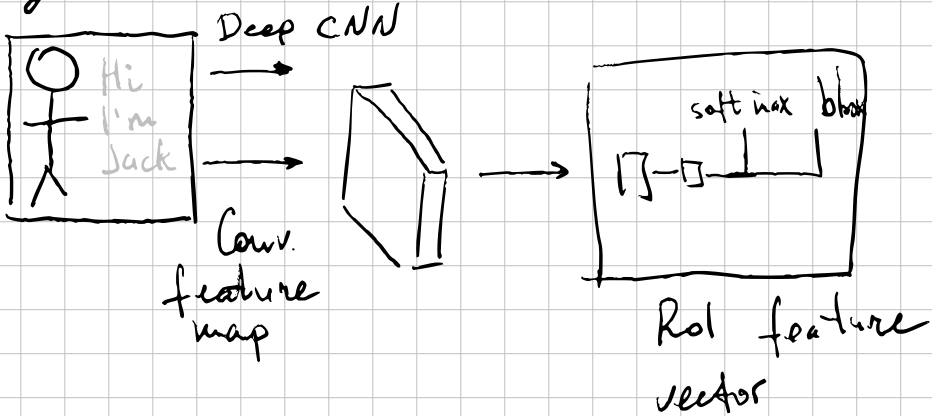


Downsides ↓:

- Inference is quite slow ($\sim 47 \frac{\text{sec}}{\text{image}}$)
- Region proposal algorithm is fixed and not trainable
- Complicated and slow training mechanism requiring a lot of space

Fast R-CNN

Instead of feeding region proposals we feed the whole image to the CNN



Pipeline 

1. Extract feature map through CNN
2. Generate region proposals on feature maps using <u>selective search</u>
3. RoI pooling
4. Predict class probabilities with Softmax
5. Predict bounding box refinements

Fast R-CNN

VS.

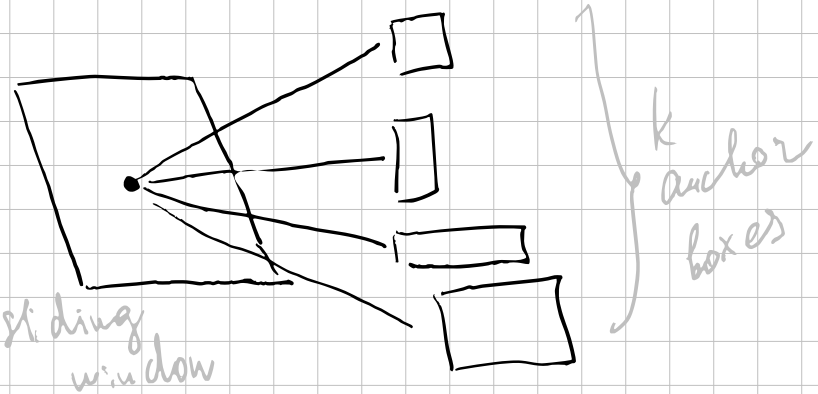
R-CNN

With selective search	2.3 $\frac{sec}{image}$	49 $\frac{sec}{image}$
w/o selective search	0.32 $\frac{sec}{image}$	47 $\frac{sec}{image}$

Faster R-CNN

Pipeline

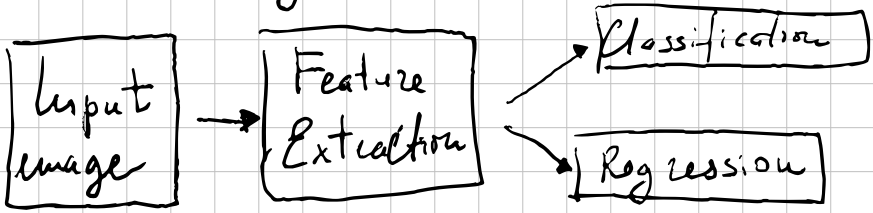
1. Extract feature map using CNN
2. Generate region proposals on feature map using RPN
3. RoI pooling
4. Predict class probabilities with Softmax
5. Predict bounding box refinements



Single - Stage detectors

- 2 - Stage Detector

- 1 - Stage Detector



We go directly from image to detection

YOLO

Each grid cell predicts: B bounding boxes and C classification scores

Each bounding box consists of:

- Center coordinates (x, y)
- Width and height (w, h)
- Confidence score

Final prediction: $S \times S \times (5B + C)$, where

S - grid size

B - number of base boxes

C - number of categories

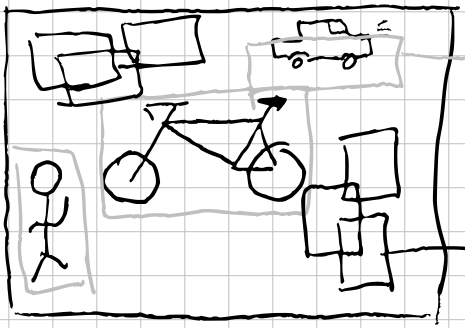
SSD (single short multibox detector)

We apply CNN in this model and reduce feature space.

- Each grid cell predicts: K bounding boxes
- Each bounding box consists of C+4 numbers:
  - 4 offsets relative to the original default box shape
  - C classification scores
- Final prediction for feature map $M \times N \times K \times (C+4)$
  - $M \times N$ – grid
  - K – Number of base boxes
  - C – number of categories (+ background)

Retina Net

Problem that led to model invention: **Foreground - background imbalance**



The architecture of Retina Net consists of four main parts:

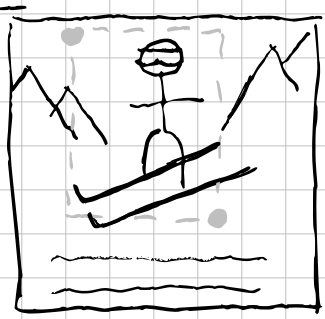
- Backbone
- FPN
- Classification Head
- Regression Head

Single Stage Detectors

anchor-free

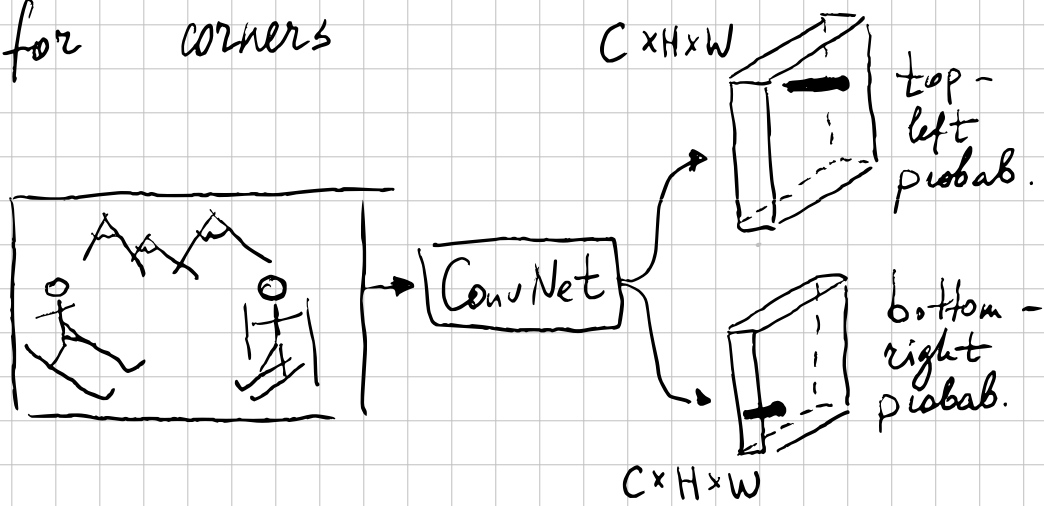
CornerNet

Represents an object as a pair of keypoints - the top-left and the bottom-right corners



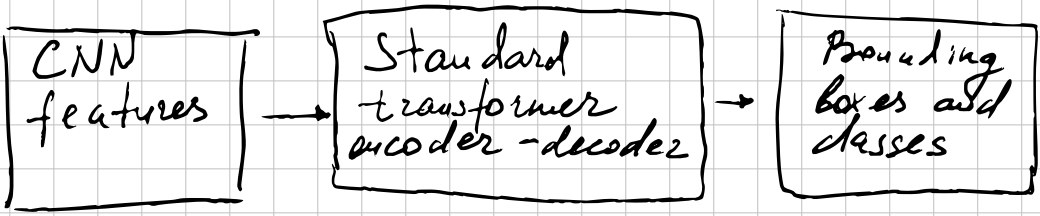
Approach

1. Extract feature map through CNN
2. Predict two sets of heatmaps for corners



3. Predict an embedding vector for each detected corner
4. Predict offsets to slightly adjust the locations of the corners

DETR

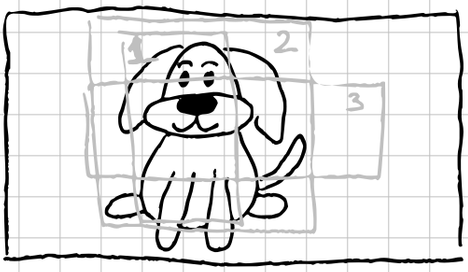


Approach:

- Extract feature map through CNN
- Transform feature map into sequence
- Pass sequence to encoder
- Pass to decoder
- Predict coordinates and class through FFN

Non-Maximum Suppression

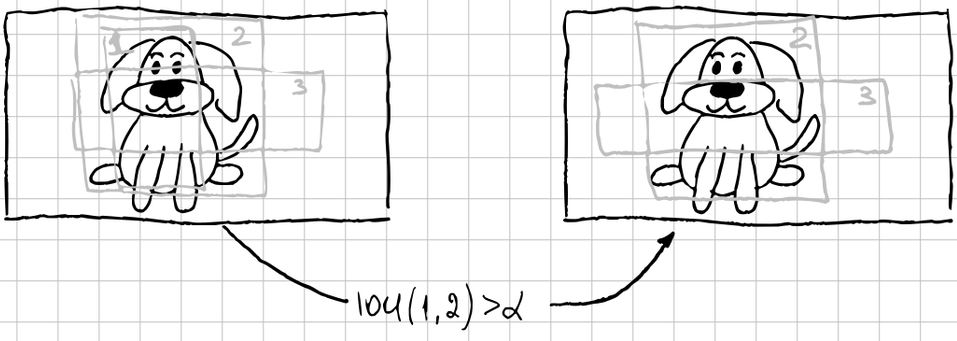
NMS is an algorithm for prediction filtering. It allows selecting the best bounding box from several predictions.

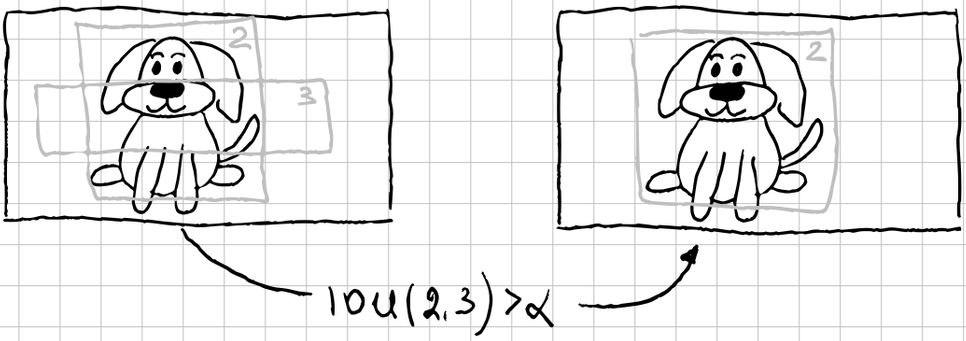


Boxes	Scores
1	0.6
2	0.9
3	0.55

To find the best bounding box we consider using intersection over union (iou).

$$IOU = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$



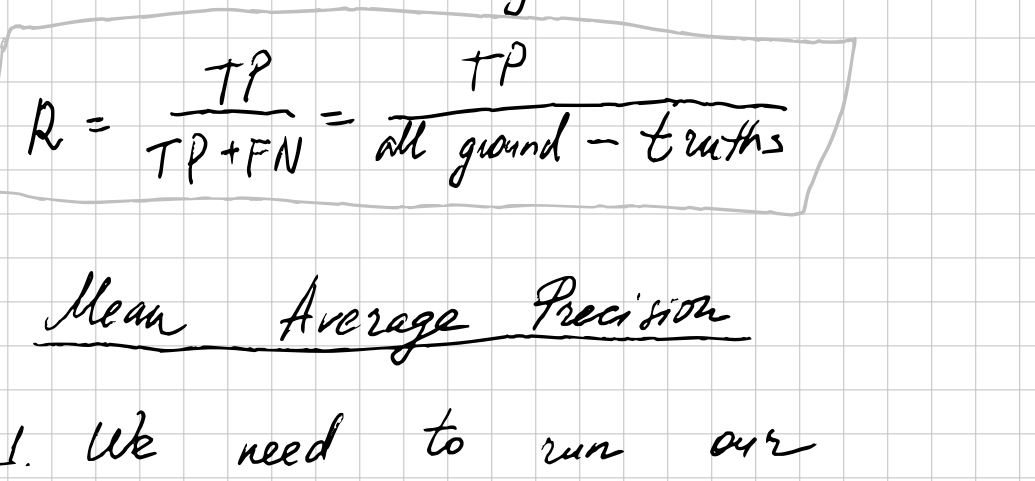


If IOU is bigger than pre-defined parameter α we simply remove the box

Evaluating

Let's define:

- True Positive (TP) - a valid detection ($IOU \geq \alpha$)
- False Positive (FP) - an invalid detection ($IOU < \alpha$)
- False Negative (FN) - Ground-truth missed by the model α-IOU threshold



Precision

Precision is the ratio of correctly detected boxes to all predicted boxes and this measures how accurate predictions are

$$P = \frac{TP}{TP+FP} = \frac{TP}{\text{all detections}}$$

Recall

Recall is the ratio of correctly detected boxes to all target boxes and this measures the ability of the model to find all relevant objects

$$R = \frac{TP}{TP+FN} = \frac{TP}{\text{all ground-truths}}$$

Mean Average Precision

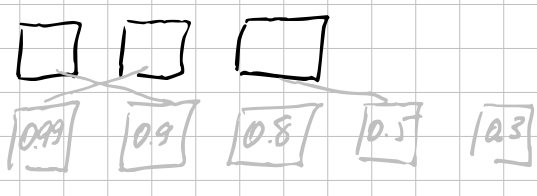
1. We need to run our detector on all images from the test set

2. For every image: sort all found objects based on classification score (the most confident box first)

3. For every predicted box we are going to assign a target box, based on IOU

Example

Ground-truth boxes 

Predicted boxes 

Score	Matched with GT	TP/FP	Precision	Recall
0.99	+	TP	1	1/3
0.9	−	FP	1/2	1/3
0.8	+	TP	2/3	2/3
0.5	+	TP	3/4	1
0.3	−	FP	3/5	1

Precision - recall curve

Area under the curve is average precision (AP)

$$AP = \int_0^1 p(r)\,dr$$

The AP is calculated individually for each class. There are as many AP values as there are classes. These AP values are averaged to produce Mean Average Precision (mAP) metric.

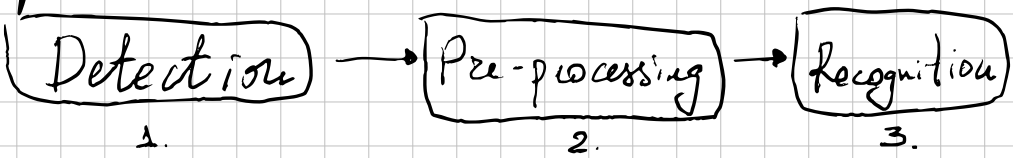
$$mAP = \frac{1}{n}\sum_{i=1}^{n} AP_i, \text{ for } n \text{ classes}$$

AP@α means Average Precision at IOU threshold of α.

AP@50 means AP at IOU threshold of 50%

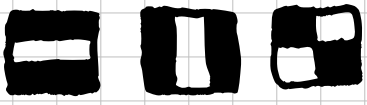
Person Re identification

Pipeline



The most popular face detectors by now:

- Haar Cascades



Haar-like features

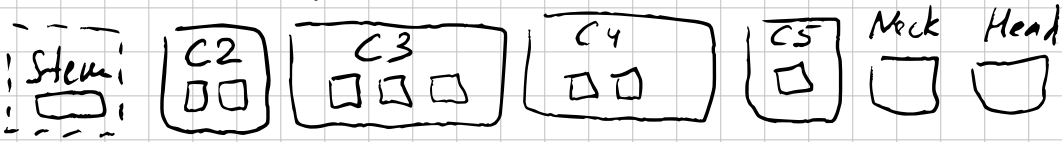
- Histogram of Oriented Gradients

- MTCNN (the first success full neural net detector)

Stages

1. Resize (using Image pyramid)
2. P-Net
3. R-Net
4. O-Net

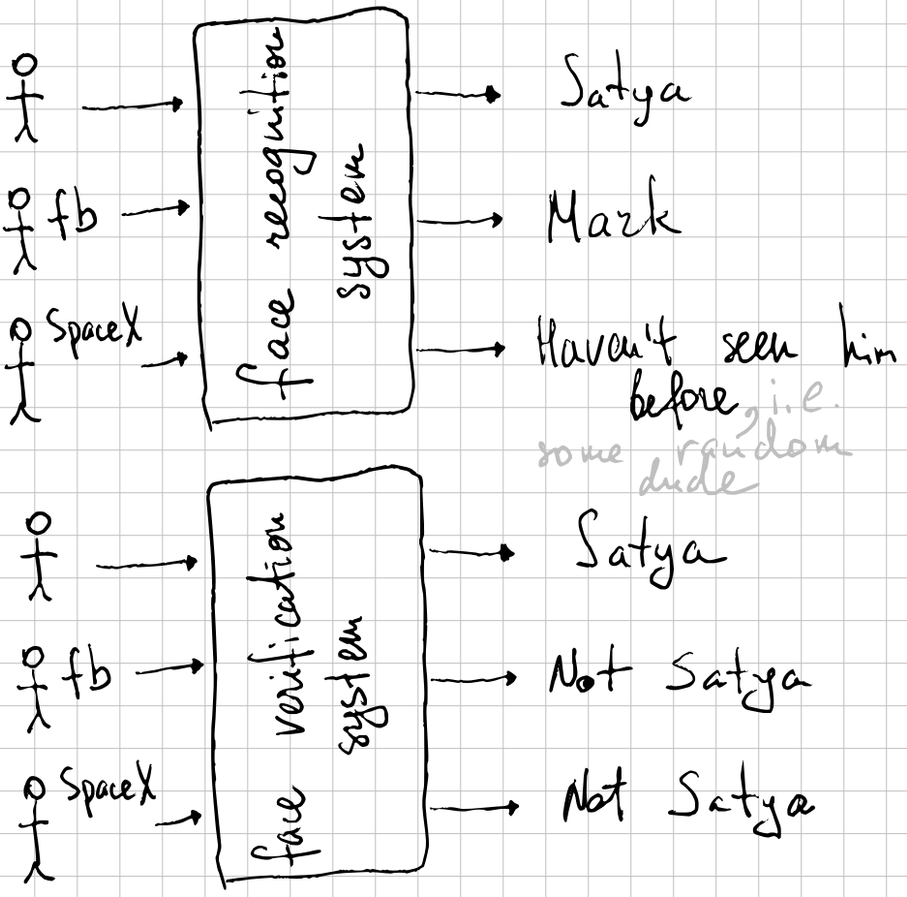
- RetinaFace
- SCRFD



Recognition vs. Verification

In general face recognition task consists of 2 subtasks:

- Identity recognition
- Identity verification (authentification)



Metrics

Verification

- Recall
- Precision

Recognition

- False Rejection Rate

$$\frac{\#\text{ false rejected}}{\#\text{ total attempts}}$$

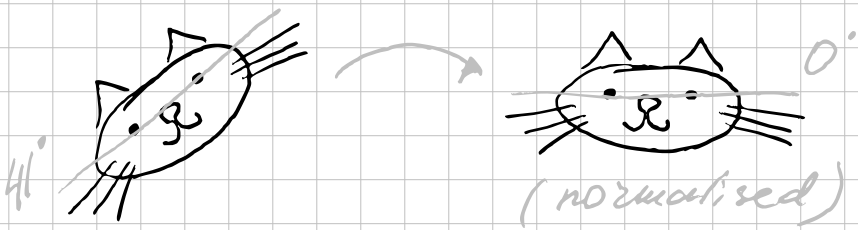
- False Acceptance Rate

$$\frac{\#\text{ false accepted}}{\#\text{ total attempts}}$$

Face Normalization

It was extremely important for faces to be uniformed.

For example, eyes should be on the same vertical level



Popular algorithms

- Eigen Faces
- Fisher Faces
- LBP

Classical Face Recognition

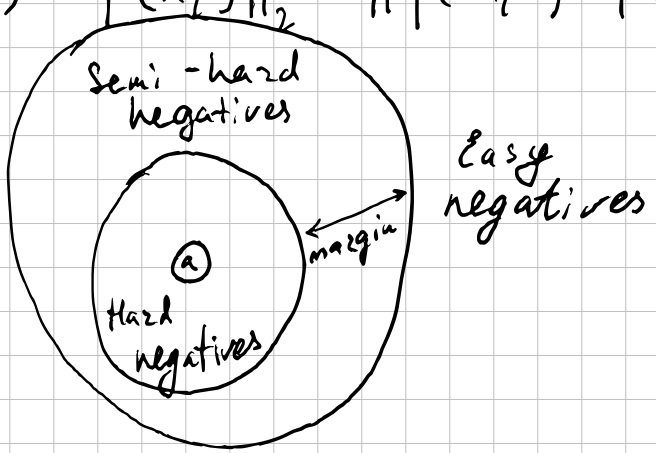
Some classical operator applied to the image (usually in a sliding window manner)

Triplet Loss

For every minibatch (for every identity):

- take equal number of identity photos
- take equal number of random photos from another identities
- for every pair of anchor-positive lets find the hardest negative

$$\|f(x_i^a) - f(x_i^p)\|_2^2 < \|f(x_i^a) - f(x_i^n)\|_2^2$$



Object tracking

Problems

1. Ambiguous metrics
2. Object movement poorly predictable
3. Fine-tuning is costly

Tracker Types

- Online / Offline
- Single-Object (SOT) / Multi-Object (MOT)
- Neural / Classical
- Static camera / Moving camera
- Predictive / Stateless
- Single-modal / Multi-modal

Tracking features

- Coordinates of object
- Kinematics
- Physiological characteristic
- Appearance features
- Extra features

MOT

1. Starts with object detection

2. Feature Generation
3. State prediction (predicts the state of a new frame)
4. Assignment (Matching)
5. Management

Algorithms

① Sort

- Find objects
- State prediction (using Kalman filter)
- Matching (using Hungarian algorithm)
- Management

② Deep SORT

* add cosine proximity between object last appearance and the new detection

Deep Learning for Image Segmentation

Segmentation types

1. Semantic segmentation
   - All pixels have labels
   - Single class not divide on objects
2. Instance segmentation
   - Some pixels does not have labels
   - Single class divide on objects
3. Panoptical segmentation
   - Combines semantic and instance

Segmentation task

Send image in model and get mask image with same resolution

1. We need to encode labels

Encoding - transforms rgb label mask H*W*C using one-hot encoding to H*W*1

```
0 0 0   2 2 2
0 1 0   2 2 2
1 1 1   0 2 0
0 1 0   0 2 0
1 0 1   0 2 0
```

0 - ground
1 - person
2 - plant

Loss and Metrics

① Pixel accuracy

$$acc = \frac{TP + TN}{TP + TN + FP + FN}$$

This metric can proride misleading results when the class representation is small within the image, as the measure will be biased in mainly reporting how well you identify negative case (class unbalance problem)

② Dice Coefficient

$$Dice = \frac{2TP}{2TP + FP + FN} \quad \left(Dice = \frac{2\,|A \cap B|}{|A| + |B|} \right)$$

Dice Coefficient is 2* the area of overlap divided by the sum of ground true and predicted mask

③ Intersection-Over-Union (IoU, Jaccard Index)

IoU is the area of overlap between the predicted segmentation and the ground truth divided by the area of union between the predicted segmentation

$$Jac = \frac{TP}{TP + FP + FN} \qquad Jac = \frac{|A \cap B|}{|A \cup B|}$$

Loss

1. Distribution-based loss
   - Focal loss
   - Top-K
   - Cross-Entropy
2. Compound loss (recombination of other losses)
   - ELL
   - Dice CE
3. Region-based loss
   - Dice
   - Jaccard
   - Tversky
4. Boundary-based loss
   - HD loss

(aims to max. overlap between ground true and prediction)

FCN

Fully convolution network

Transforms classification network to segmentation:

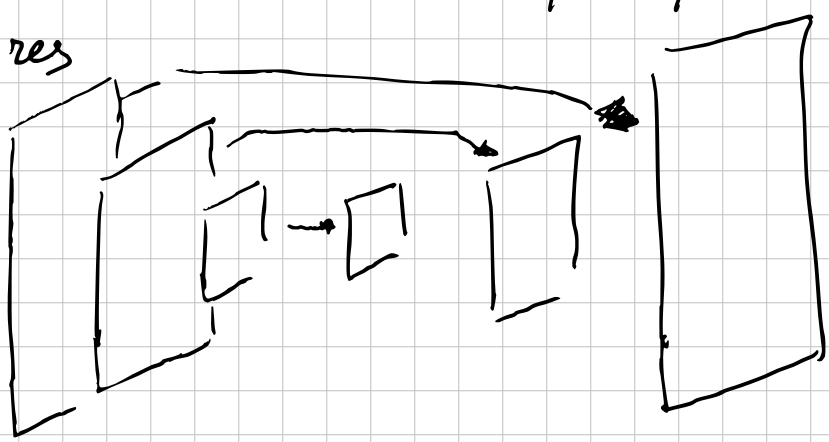
- Remove dense layer
- Add decoder part
- Use 1*1 convolution to predict labels
- Upsample image to default resolution

Architecture

- Fast and big upsampling gives a rough result, so we make it smoother and we mix the results with the previous encoder outputs

Segnet

- Full encoder-decoder architectures
- Save pooling indices in encoder part and use them in decoder when upsample features

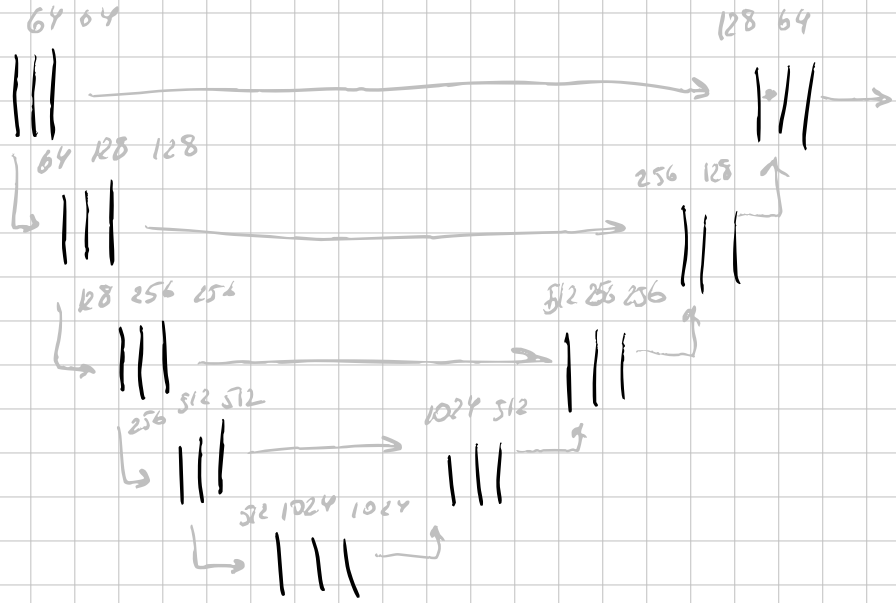


Max pooling

Saves maximum element in encoder and reuse positions from pooling layer (max unpooling)

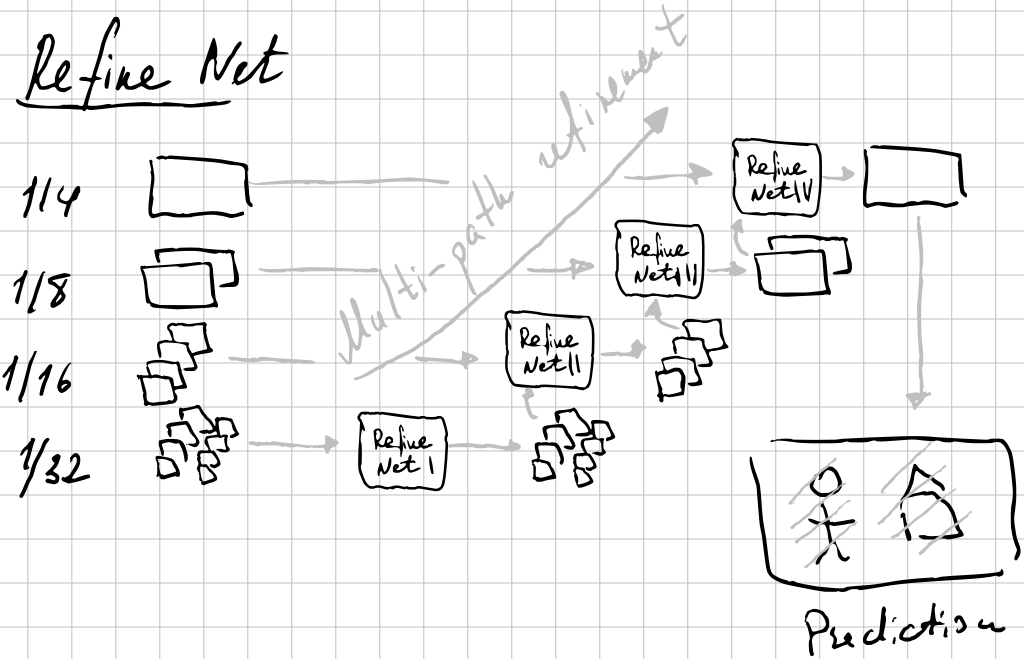
* When we use max pooling / unpooling we lost many information when unpooling and sometimes this is not enough

U-Net



- Share features from encoding to corresponding decoder layers. This sharing combine deep features from encoder which know more about all image and more locally accuray features
- Better to use pre-trained encoder from another dataset or task
- We can train just decoder part if we have tiny dataset
- Possible to use with multi-resolution input

Refine Net



Combined the long-range skip-connection connections from the original U-net architecture, with a lot of added skip-connections

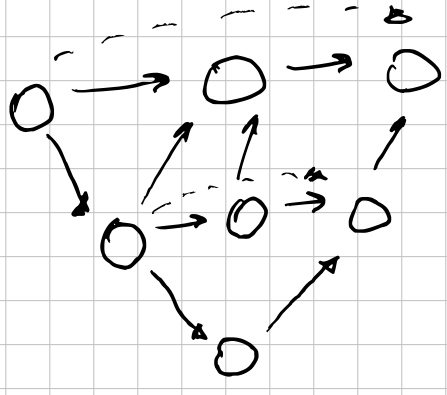
Consists of:

- Residual Conv Unit
- Multi-resolution fusion
- Chained Residual Pooling

U-Net ++

This work continues the ideas of U-net, but at the same time solves some of its problems:

1. Selected models because best results on different data sets have different depth
2. Rigid scheme of skip connections which is possible only on features of the same resolution



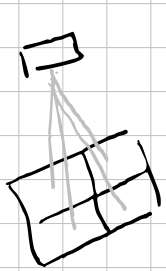
After training the whole model, we can use smaller network to improve performance

Train small U-net in U-net ++ architecture gives more IoU then isolated training

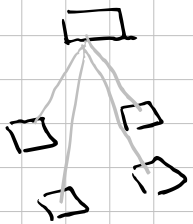
Deep Lab

Dilated convolution

Use of dilated convolution allows one to have larger receptive field (global view) with the same computation and memory costs while also preserving resolution

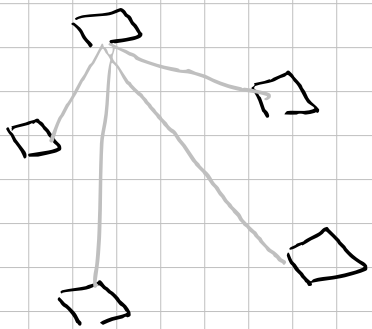


dilation 1



2

(skipped 1 cell)



3

(skipped 2 cells)

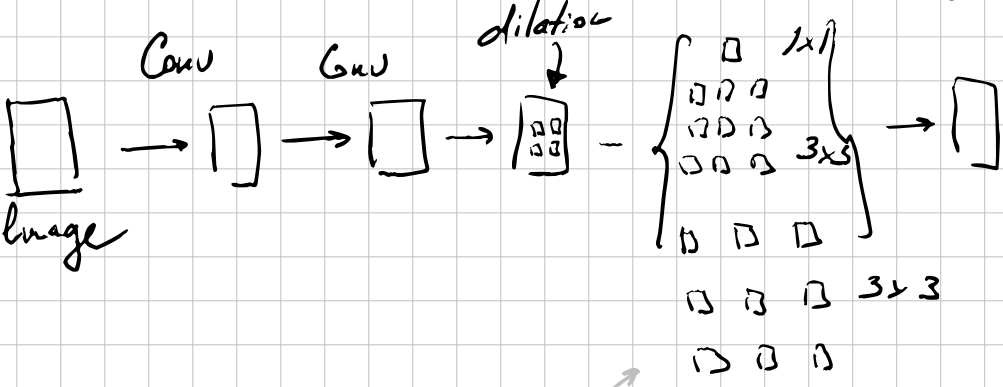
CRF

Conditional Random Field allowes to correct errors

(Post-processing step)

Deep Lab v2

Atrous Spatial Pyramid Pooling



(v1) does not have this block

Deep Lab v3

- Add decoder part to previous version
- Combine low-level and final encoder features
- More smoother upsample

PSPNet

Pyramid Scene Parsing Network

Use pyramid pooling module for global scene prior construction upon the final-layer-feature-map of the deep neural network

Hierarchical Multi-scale Attention

The same image with different resolutions can give different results:

- bigger scale gives better result on thin object;
- lower scale gives better result on large objects;

So we want to combine this result using hierarchical multi-scale attention

Architecture

Hierarchical attention mechanism learns to predict a relative weighting between adjacent scales.

* With this network we can combine different image scales

HR Net

The main body consists of several components:

- parallel multi-resolution convolutions;
- multi-resolution fusions;
- representation head.

Connects high-to-low resolution convolution streams in parallel rather than in series

Repeats multiresolution fusions to boost the high-resolution representations with the help of low-resolution repr.

Mask-RCNN

Extends Faster R-CNN

Architecture

- Generate binary mask for each class
- Change RoI Pool to RoI Align for better score

Mask Head

Collecting mask annotations, for instance, segmentation is very expensive so we want to train just using bounding boxes for all categories but use masks only for a subset of categories.

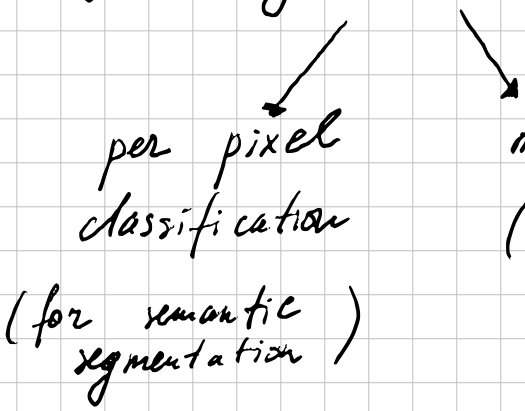
Adapt Mask-RCNN for partially-supervised training just by using better mask-head architectures without extra losses or modules.

The following factors have a big influence on generalisation of unseen classes

1. train using tight ground-truth boxes instead of combination of ground-truth and noisy proposals
2. mask-head architecture

Mask Former

There are 2 approaches in image segmentation



Mask classification is sufficiently general to solve both semantic- and instance-level segmentation tasks, so the authors propose a structure that transforms any per-pixel classification model to mask classification.

Architecture

- pixel-level module
- transformer module
- segmentation module