- ✓ Goffrey Hinton (back Prop, NN)
- ✓ Pieter Abbeel (reinforcement learning)
- ✓ Ian Goodfellow (GANs)

- Yoshua Bengio
- Yuanqing Lin

- Andrej Karpathy
- Ruslan Salakhutdinov

- Yan LeCun

Deep Learning : Week 1 — Intro to Deep Learning

Intro :

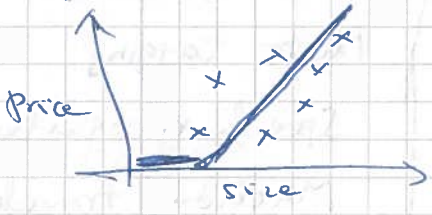
- AI is new Electricity
- end-to-end learning
- train / dev / test
- CNN
- RNN / LSTM

1. Neuronal Networks and Deep Learning
2. Improving Deep Neuronal Networks: Hyperparameter tuning, Regularization Optimization
3. Structuring your ML project
4. CNN
5. Natural Language Processing Building sequence models

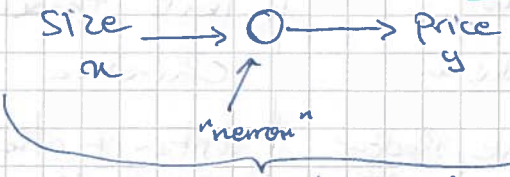
Learning Objectives

- Understand the major trends driving the rise of Deep Learning.
- Be able to explain how deep learning is applied to supervised learning
- Understand what are the major categories of Models (such as CNN and RNN) and when they should be applied
- Be able to recognize when Deep learning will and will not work well.

What is a neuronal Network ?

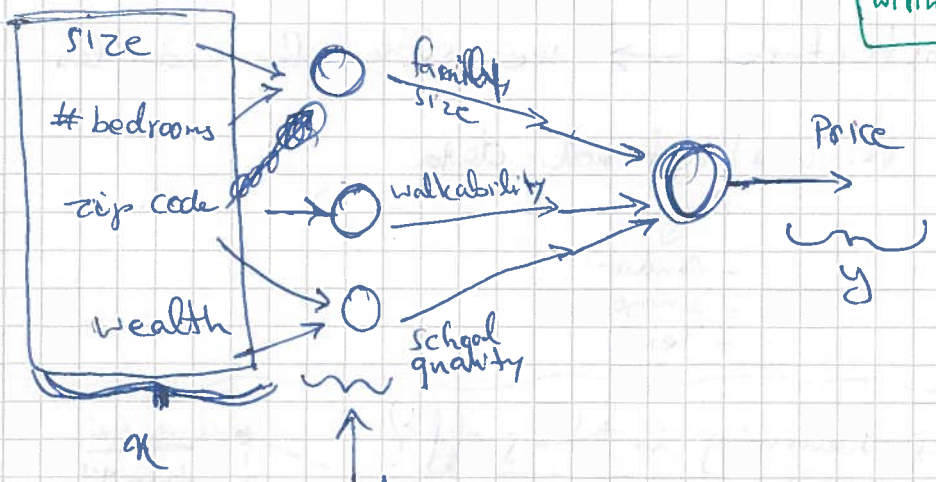


Price = f(size)



simplest single neuron NN

ReLU : Rectified Linear Unit



Take all four input features (Densely connected)

- NN are best for the supervised learning settings
  : finding a function that maps an input x to an output y. (x, y)
  $y = f(x)$

Supervised Learning with NNs

	Input (x)	output (y)	Application
Standard NN	Home features	Price	Real state
Standard NN	Ad, user info	click on ad? (0/1)	Online advertisement
CNN	Image	Object (1, ..., 1000)	Photo tagging
RNN	Audio	Text transcript	Speech recognition
RNN	English	Chinese	Machine translation
Custom / Hybrid	Image, Radar Info	Position of other Cars on the Road	Autonomous Driving

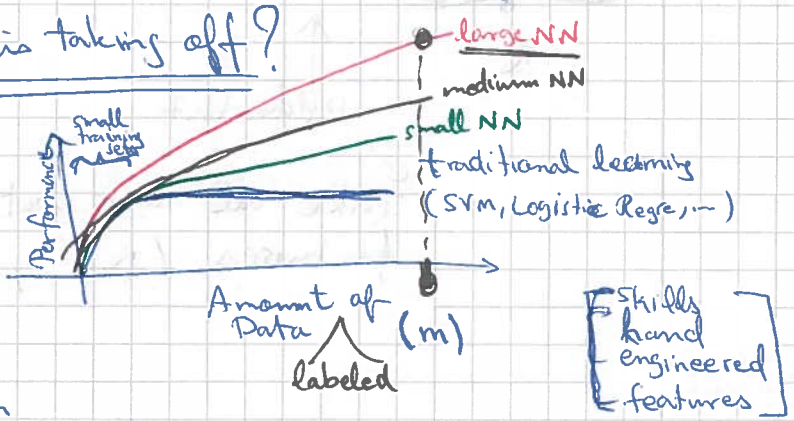
- Cleverly selecting what should be x and what should be y
- NN architecture → see slides for examples
- Structured vs. unstructured data

  - Home info
  - Ad info

  - Audio
  - Image
  - Text

Why deep learning is taking off?

- Drivers of DL:
  - scale
    - data
    - computation
    - Algorithms (recently) (innovations)

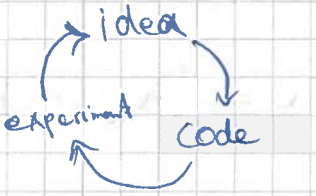


- Algorithms improvements

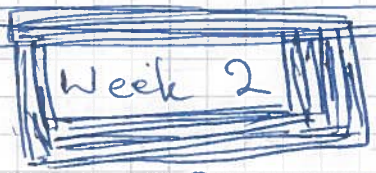
e.g.

made computing very much faster

- faster computation



10 min
1 day
1 month
} try a lot more ideas



Neural Networks Basics

Learning Objectives:

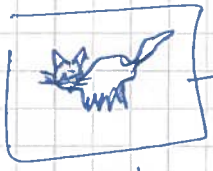
- Build a logistic Regression model as a shallow NN
- Implement the main steps of an NN algorithm, including making predictions, derivitive computation, and Gradient descent.
- Implement computationally efficient, highly vectorized version of a model
- Understand How to compute derivitives for logistic regression using Backprop.
- Become familiar with Python & numpy
- work with iPython notebooks
- Implement vectorization across multiple training examples

Binary Classification:

- forward pass / backward pass
- logistic regression

↓

An algorithm for binary classification



input

→ 1 (cat) vs. 0 (not cat)

output

Image is 64 x 64 matrices of RGB

↳ "unroll" pixel values into a vector $x = \begin{bmatrix} \vdots \end{bmatrix}$ of size 64x64x3 = 12288

$n = n_x = 12288$

Notation:

(x, y) single training example

$x \in \mathbb{R}^{n_x}$, $y \in \{0, 1\}$

m training examples: $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \dots (x^{(m)}, y^{(m)})\}$

→

m_{train} m_{test} = # test examples

$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix} \quad n_x \text{ rows}, \; m \text{ columns}$$

$X \in \mathbb{R}^{n_x \times m}$

$X.shape = (n_x, m)$

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

$Y \in \mathbb{R}^{1 \times m}$

$Y.shape = (1, m)$

Logistic Regression

in a supervised learning problem

A learning algorith when output labels Y are either 0 or 1

Given x, want $\hat{y} = P(y=1|x)$, $0 \leq \hat{y} \leq 1$

x: Image

$\hat{y}$: what is the chance that x is a cat picture?

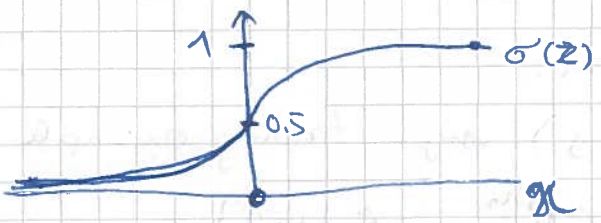
$x \in \mathbb{R}^{n_x}$

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

Output: $\hat{y} = \sigma(w^T x + b)$ → linear Regression would not work

$[\hat{y} = w^T x + b]$ → linear regression would not work because $\hat{y}$ should be $\in [0,1]$

Sigmoid function: σ



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

(4) if z large $\Rightarrow \sigma(z) \approx \frac{1}{1+0} = 1$ / if z large negative $\Rightarrow \sigma(z) \approx 0$

← Logistic Regression:

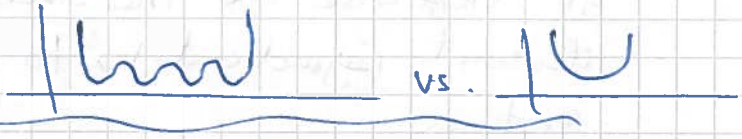
your job is to learn the parameters w, b so that $\hat{y}$ becomes a good estimate of $P(y=1|x)$

Logistic Regression Cost function:

- ground truth

We could define the "Loss function" / "error function" as

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

but in logistic regression people don't do it because the optimization problem becomes non-convex when learning the parameters. It will have multiple local optima and gradient descent won't find a global optima  vs.

Loss Function: defines how good the estimate $\hat{y}$ is of y is.

works on one single training example

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

Intuition:

If $y=1$: $\mathcal{L}(\hat{y}, y) = -\log \hat{y}$ we want it to be as small as possible

$\Rightarrow$ let's make $\log \hat{y}$ as big as possible $\Rightarrow$ make $\hat{y} = 1$

If $y=0$: $\mathcal{L}(\hat{y}, y) = -\log(1-\hat{y})$ we want it to be as small as possible $\Rightarrow$ make $\log(1-\hat{y})$ as large as possible

$\Rightarrow$ make $\hat{y} = 0$

Cost Function: for the entire training set

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

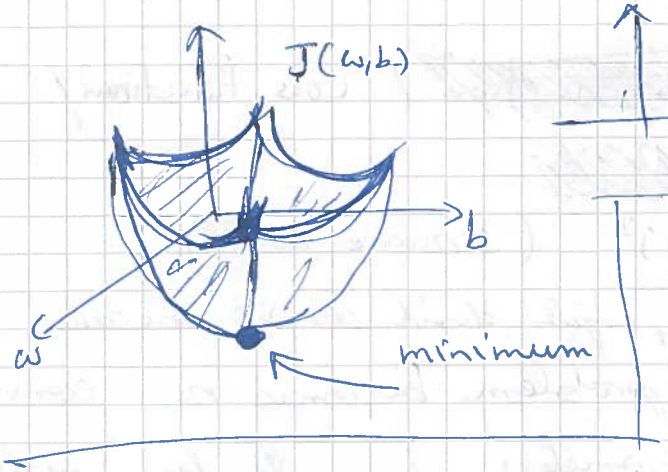
average of the loss function

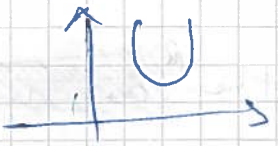
Gradient Descent

Recap: $\hat{y} = \sigma(w^T + b)$, $\sigma(z) = \frac{1}{1+e^{-z}}$

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

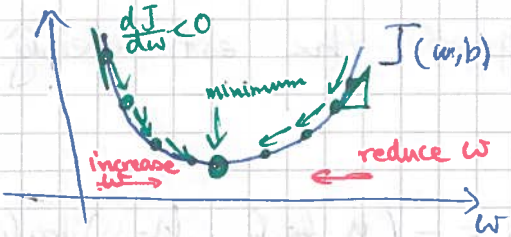
- We want to find w, b that makes our cost function $J(w,b)$ minimum.





convex function

So: initialize w and b, to zero or random, go step wise at the direction of the steepest descent (-gradient), to converge to the global optima.



Repeatedly: {

$$w := w - \alpha \underbrace{\frac{dJ(w)}{dw}}_{\text{"dw"}}$$

learning rate

}

$$\boxed{w := w - \alpha\, dw}$$

$\frac{dJ}{dw}$ slope of the function

So that we know in which direction we need to step in order to go downhill.

$J(w,b)$

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

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

dw

$$\frac{\partial J(w,b)}{\partial w}$$

"Partial derivitive"

slope of the Function in the w direction

db

$$\frac{\partial J(w,b)}{\partial b}$$

Computation Graph

$J(a,b,c) = 3(a+b*c)$

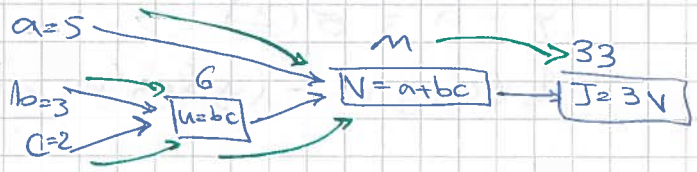
$u = b * c$

$v = a + u$

$J = 3v$

Compute the Cost Function

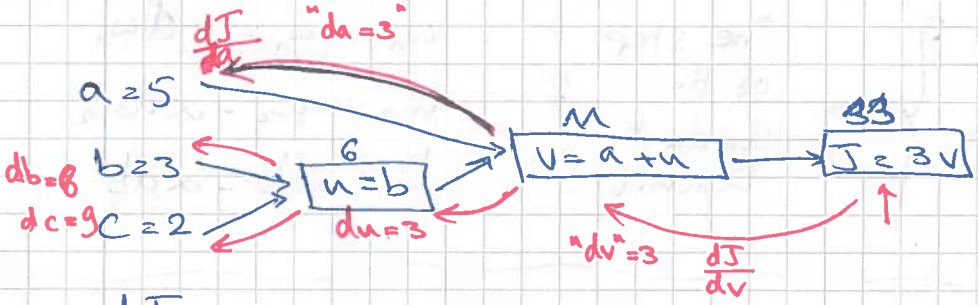
Forward Pass



(such as J)

- Comes in handy when there is an output variable that you want to optimize.
- In case of logistic regression J is the cost function that we want to minimize
  - left to right pass computes the value of J
  - A right to left pass would be the most natural to computation of the derivatives

Derivatives with the Computation Graph



Backward Pass

Compute the derivitive of the Cost Function

$\frac{dJ}{dv} = ? = 3$

$\frac{dJ}{da} = 3 = \underbrace{\frac{dJ}{dv}\frac{dv}{da}}_{\text{chain rule } a \to v \to J} = 3 \times 1$

$\frac{dv}{da} = 1$

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

$\frac{dJ}{db} = \underbrace{\frac{dJ}{du}}_{=3} \cdot \underbrace{\frac{du}{db}}_{=2} = 6$

$\frac{dJ}{dc} = \underbrace{\frac{dJ}{du}}_{=3} \cdot \underbrace{\frac{du}{dc}}_{=3} = 9$

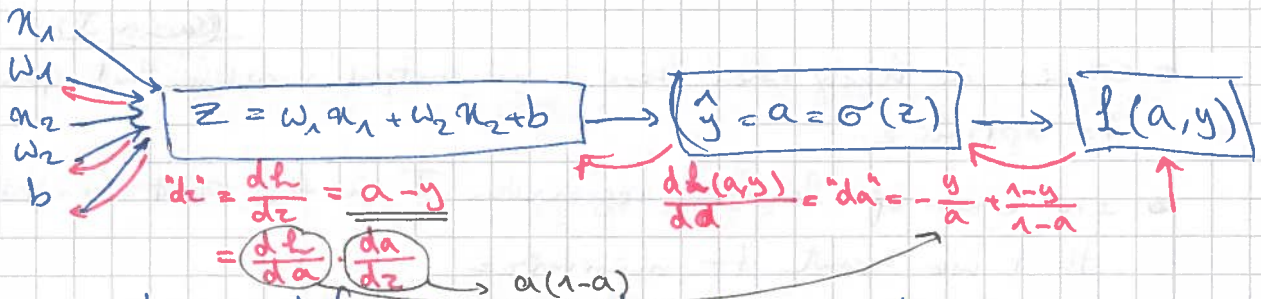
In code following convention:

use variable name "dvar" for

$\frac{d\,\text{FinalOutputVar}}{d\,\text{var}}$

Logistic Regression Gradient Descent

$$z = w^T x + b$$
$$\hat{y} = a = \sigma(z)$$
$$\mathcal{L}(a, y) = -(y \log(a) + (1-y)\log(1-a))$$



- Task: modify the parameter w_1, w_2, and b in order to reduce the loss $\mathcal{L}(a,y)$
- Forward propagation: comput the loss
- Backward propagation: compute the derivitives

$$\frac{\partial \mathcal{L}}{\partial w_1} = "dw_1" = x_1 dz$$
$$dw_2 = x_2 dz$$
$$db = dz$$

$\Rightarrow$ One step of the Gradient Descent

$$w_1 := w_1 - \alpha dw_1$$
$$w_2 := w_2 - \alpha dw_2$$
$$b := b - \alpha db$$

Gradient Descent on m Examples

- Cost Function:

$$J(w,b) = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$$
$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T x^{(i)} + b)$$

previous lecture for 1 training example: $(x^{(i)}, y^{(i)})$ $dw_1^{(i)}, dw_2^{(i)}, db^{(i)}$

$$\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m} \sum_{i=1}^{m} \underbrace{\frac{\partial}{\partial w_1} \mathcal{L}(a^{(i)}, y^{(i)})}_{dw_1^{(i)}}$$

Algorithm : Logistic Regression on m examples

→ ~~x_1, x_2~~, 2 features

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

For $i = 1$ to m:

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

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

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

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

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

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

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

Calculate the value of cost function and its derivate with respect for w_1, w_2 and b

$\frac{\partial J}{\partial w_1}$

$$J \mathrel{/}= m$$

$$dw_1 \mathrel{/}= m$$

$$dw_2 \mathrel{/}= m$$

$$db \mathrel{/}= m$$

$\frac{\partial J}{\partial w_2}$ $\frac{\partial J}{\partial b}$

- dw_1, dw_2, db don't have a superscript. They are accumulators.

→ One step of Grad. Desc. will then be:

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

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

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

- Repeat three above code till termination criteria is reached
- Two weaknesses
  - two For loops: 1. over m training examples
    2. over n features
- Explicit fors makes Algorithms inefficient
  → Vectorization [get rid of for loops]
- In deep learning era vectorazition is no more a "nice-to-have" but a necessity

Vectorization

$z = w^T x + b$

$w = \begin{bmatrix} | \\ | \end{bmatrix}$, $x = \begin{bmatrix} | \\ | \end{bmatrix}$ $\quad w \in \mathbb{R}^{n_x}$, $x \in \mathbb{R}^{n_x}$

Non-Vectorized

```
z = 0
for i in range(n_x):
    z += w[i] * x[i]
z += b
```

Vectorized

```
z = np.dot(w, x) + b
```

($w^T x$)

↑ 300 times faster!

because it uses parallel processing of multicore CPU / GPU

- Vectorization uses parallelization instructions — SIMD, simd
- **Rule of thumb:** Whenever possible avoid using explicit For Loops!

Vectorizing Logistic Regression

$J = 0$, $dw_1 = 0$, $dw_2 = 0$, $db = 0$

for $i = 1$ to m:

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

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

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

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

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

$dw_2 \mathrel{+}= x_2^{(i)} dz^{(i)}$ } $n_x = 2$

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

$J = J/m$, $dw_1 = dw_1/m$, $dw_2 = dw_2/m$, $db = db/m$

Vectorized:

$dw = np.zeros((n_x, 1))$

$Z = np.dot(w.T, X) + b$

$A = \sigma(Z)$

broadcasting →

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

$dw /= m$

$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$ $\quad (n_x, m)$, $X \in \mathbb{R}^{n_x \times m}$

$[\ \xrightarrow{\quad w^T \quad}\ ] \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$

$Z = [z^{(1)}\ z^{(2)} \cdots z^{(m)}] = w^T X + \underbrace{[b\ b \cdots b]}_{1 \times m} = [w^T x^{(1)} + b \quad w^T x^{(2)} + b \cdots w^T x^{(m)} + b]$

← • Broadcasting: expanding values/vectors to the required shape.

$dz^{(1)} = a^{(1)} - y^{(1)}$, $dz^{(2)} = a^{(2)} - y^{(2)}$, ...

$dZ = \underbrace{[dz^{(1)}\ dz^{(2)} \dots dz^{(m)}]}_{1 \times m}$

$A = [a^{(1)}\ a^{(2)} \dots a^{(m)}]$ $\quad Y = [y^{(1)}\ y^{(2)} \dots y^{(m)}]$

$dZ = A - Y = [a^{(1)} - y^{(1)}\ \ a^{(2)} - y^{(2)} \dots a^{(m)} - y^{(m)}]$

m times:

$dw = 0$	$db = 0$
$dw \mathrel{+}= x^{(1)} dz^{(1)}$	$db \mathrel{+}= dz^{(1)}$
$dw \mathrel{+}= x^{(2)} dz^{(2)}$	$db \mathrel{+}= dz^{(2)}$
⋮	⋮
$dw \mathrel{/}= m$	$db \mathrel{/}= m$

vectorized

$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$

$= \frac{1}{m}$ np.sum(dZ)

$dw = \frac{1}{m} X\, dZ^T$

$= \frac{1}{m} \begin{bmatrix} | & & | \\ x^{(1)} & \dots & x^{(m)} \\ | & & | \end{bmatrix} \begin{bmatrix} dz^{(1)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$

$= \frac{1}{m} \underbrace{\begin{bmatrix} x_1^{(1)} dz^{(1)} + \dots + x_1^{(m)} dz^{(m)} \\ x_2^{(1)} dz^{(1)} + \dots + x_2^{(m)} dz^{(m)} \\ \vdots \\ x_n^{(1)} dz^{(1)} + \dots + x_n^{(m)} dz^{(m)} \end{bmatrix}}_{n \times 1 \text{ vector}}$

All together:

$Z = w^T X + b$ (forward pass)

$= np.dot(w.T, X) + b$

$A = \sigma(Z)$

$dZ = A - Y$

$dw = \frac{1}{m} X\, dZ^T$

$db = \frac{1}{m}$ np.sum(dZ)

$w := w - \alpha\, dw$ | GD

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

(backward propagation)

A single iteration of the Gradient Descent

Broadcasting:

(m,n) matrix $+ - * /$ (1,n) ⟶ (m,n)

(m,n) matrix $+ - * /$ (m,1) ⟶ (m,n)

(m,1) $+$ $\mathbb{R}$ ⟶ (m,1)

$\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} + 100 \longrightarrow \begin{bmatrix} 101 \\ 102 \\ 103 \end{bmatrix}$

$[1\ 2\ 3] + 100 \longrightarrow [101\ \ 102\ \ 103]$

A Note on Python/numpy Vectors

* Broadcasting advantage/powerful, but also if not used with care can lead to very strange very hard to find bugs.

```
a = np.random.randn(5)   ⟶  [. . . . .]
a.shape()                ⟶  (5,)        rank 1 array!

a = np.random.randn(5,1) ⟶  [[.]
                             [.]
                             [.]
                             [.]
                             [.]]
a.shape()                ⟶  (5,1)
```

column vector

```
assert(a.shape == (5,1))   ⟸ !
a = a.reshape((5,1))
```

- Don't use "rank 1 arrays", always use column/row vectors with defined shape
  $n \times 1$, $1 \times n$ matrices
- Do assertions regularily
- Use reshape to make sure your vectors have correct shape

Explanation of Logistic Regression cost Function

$$\hat{y} = \sigma(w^T x + b) \qquad \sigma(z) = \frac{1}{1+e^{-z}}$$

Interpret $\hat{y} = P(y=1|x)$

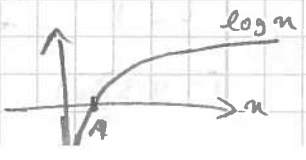
if $y=1$: $P(y|x) = \hat{y}$
if $y=0$: $P(y|x) = 1-\hat{y}$

} $P(y|x)$

y has to be 1 or 0 because it is a binary classification

$P(y|x) = \hat{y}^y (1-\hat{y})^{(1-y)}$ } ⟶ summarizes $P(y|x)$

(12) • log function is strictly monotonic func



$$\log P(y|x) = \log \hat{y}^{y} (1-\hat{y})^{(1-y)} = y \log \hat{y} + (1-y)\log(1-\hat{y})$$

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

- negative sigma because:
  - we want to maximize the probability
    ⇒ minimize the loss function

- For cost function on the entire train set:

Assuming all samples are drawn i.i.d ! identically & independently distributed.

$$P(\text{labels in the training set}) = \prod_{i=1}^{m} P(y^{(i)}|x^{(i)})$$

$$\log P(\text{labels in the training set}) = \log \left(\prod_{i=1}^{m} P(y^{(i)}|x^{(i)}) \right)$$

$$\log P(\text{labels in the training set}) = \sum_{i=1}^{m} \log P(y^{(i)}|x^{(i)}) = -\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})$$

$\log P(y^{(i)}|x^{(i)}) = -\mathcal{L}(\hat{y}^{(i)}, y^{(i)})$

- Maximum Likelihood Estimations:
  - simply means: Choose parameters that maximizes this thing

- Cost Function:

maximize likelihood
⇒ minimize the cost
⇒ get rid of ⊖ sign

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

Add for convenience to make sure the quantities better scale

Programming Exercise:

- reshape data — size dataset, image

$(209, 64, 64, 3)$

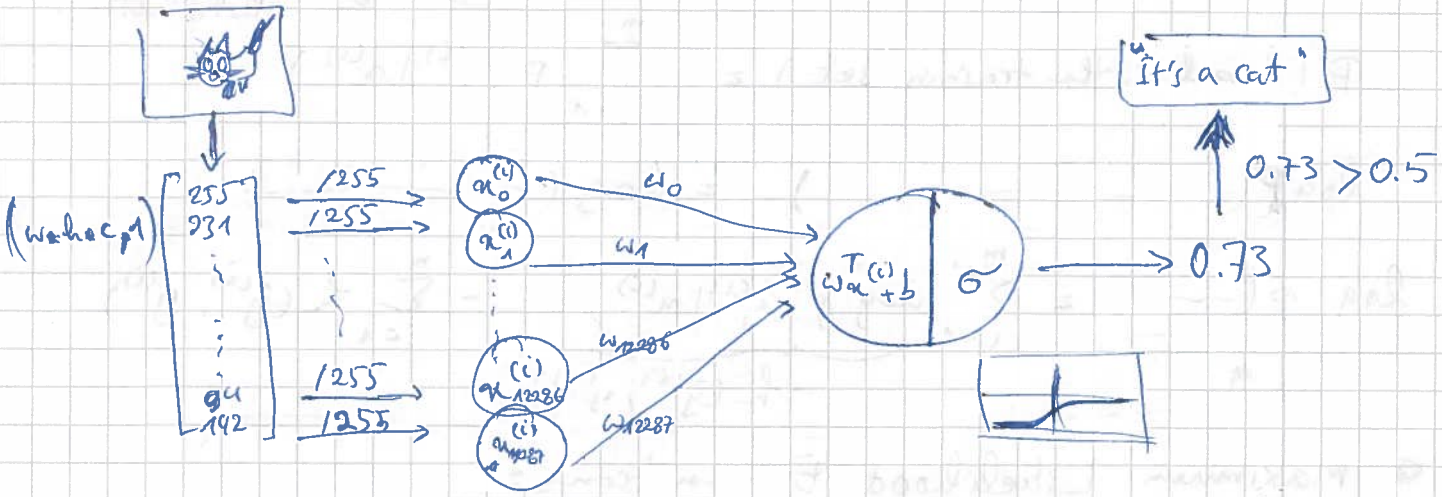
X.shape = (a, b, c, d)

X.reshape(X.shape[0], -1).T ⟶ (b*c*d, 1)

$$\begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(209)} \\ | & | & & | \end{bmatrix}$$

- H5 data format
- Standardize :
  - subtract mean
  - devide by standard dev
  - for Images :
    - just devide by 255

General Architecture of the Learning Alg.



$$z^{(i)} = w^T x^{(i)} + b \qquad \hat{y}^{(i)} = a^{(i)} = \sigma(z^{(i)})$$

loss function $\leftarrow \mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log a^{(i)} - (1-y) \log(1-a^{(i)})$

Cost Function : $J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)})$

Steps :

- init params
- learn the params for the mode by minimizing the cost
- Use the learned params to make predictions (on the test set)
- Analyse & conclude

- initialize (w, b)
- Optimize the loss iteratively to learn parameters (w, b)
  - compute the cost and its gradient
  - update the params using the gradient descent
- use the learned (w, b) to predict the labels for a given set of examples.

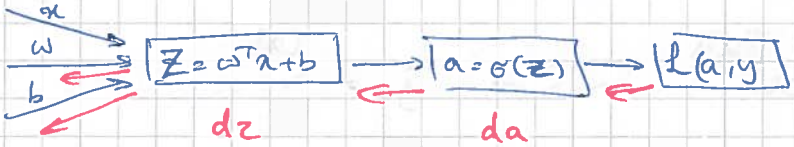
1. preprocessing the dataset is important
2. You implemented each function separately ~~initialize(), propagate(), optimize()~~
   - initialize(), propagate(), optimize(),
   then you built your model() by combining them.
3. Tuning the learning rate (which is an example of a "hyperparameter" can make a big difference to the Algorithm.

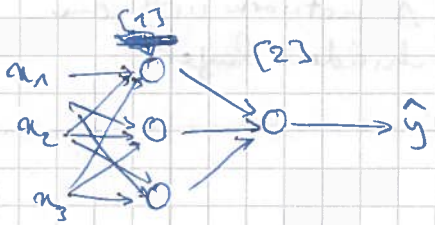
Week 3

Shallow neural Networks.

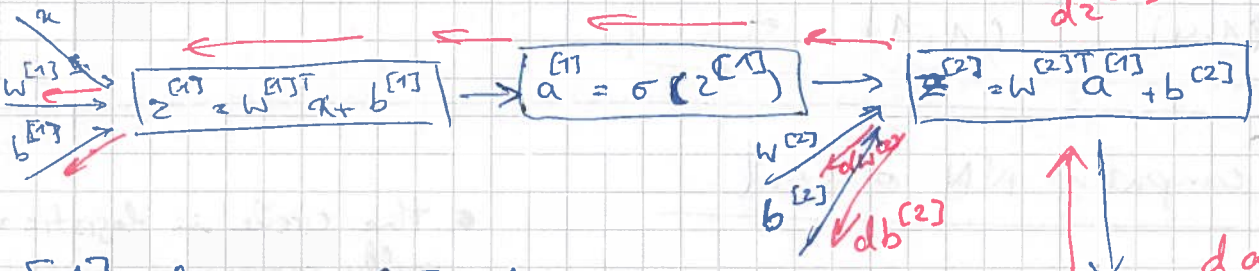
Learning Objectives

- Understand hidden units and hidden layers
- Be able to apply a variety of activation functions in neural network.
- Build your first forward & Backward Propagation with a hidden layer.
- Apply random initialization to your neural network.
- Become fluent in deep learning notation an Neural Network Representation
- Build and train neural networks with one hidden Layer.



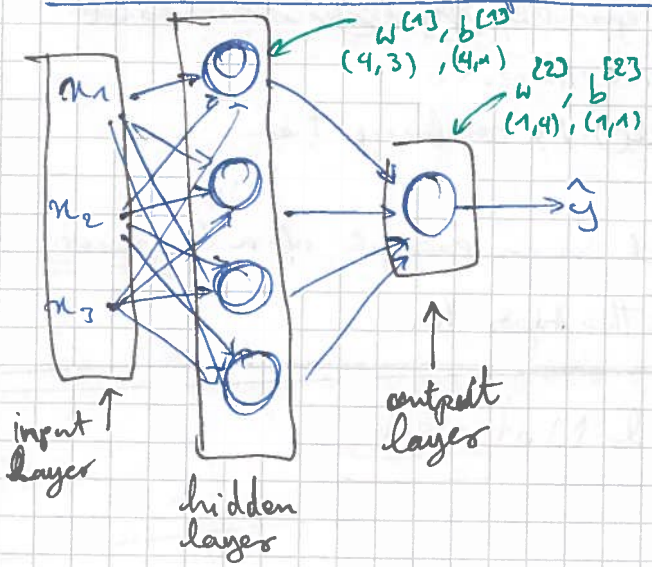


How does a NN look like?



- [1] → layer 1, [2] → layer 2
- NN is like taking the logistic Regression and repeating it twice

Neural Network Representation



Input features:

$a^{[0]} = X$

a stands for activation.

activation: Different values that different layers of NN are passing on.

$a^{[1]}_1$ → value of first node in hidden layer 1

$$a^{[1]} = \begin{bmatrix} a^{[1]}_1 \\ a^{[1]}_2 \\ \vdots \\ a^{[1]}_4 \end{bmatrix}$$ → 4 hidden units/nodes

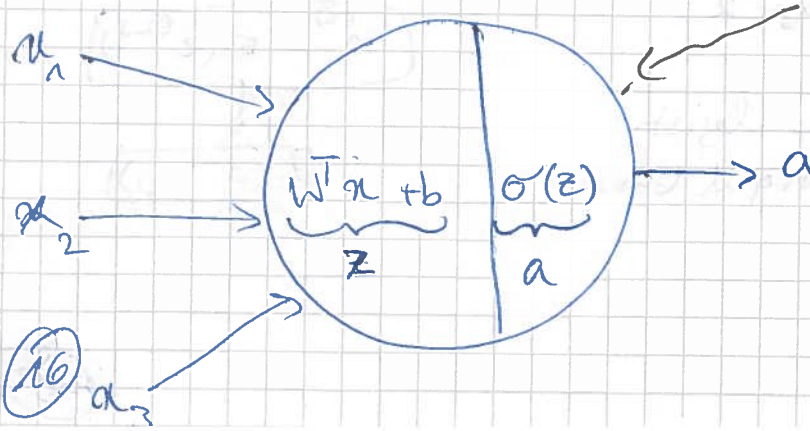
$a^{[2]}$ → $\hat{y} = a^{[2]}$

$W^{[1]}$ (4,3), $b^{[1]}$ (4,1) dimension

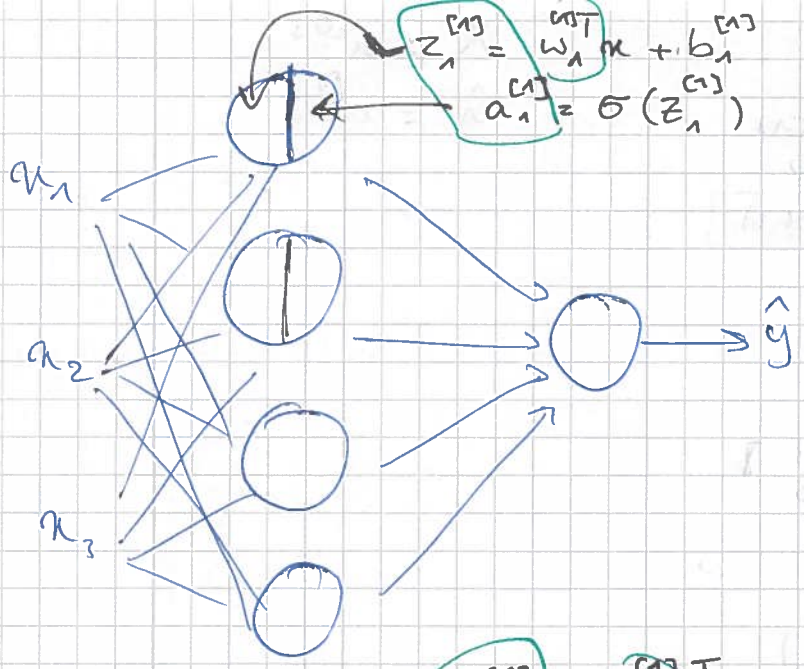
$W^{[2]}$ (1,4), $b^{[2]}$ (1,1)

A 2 layer NN,
- the input layer is not counted.
- A network with one hidden layer.

Computing NN's output



- the circle in logistic regression really represents two steps of computations
- A Network with hidden layer just does this a lot more times



$[l]$ → layer

$(i)^{[l]}$ → node i in layer l

- accordingly:

$$z_2^{[1]} = w_2^{[1]T} x + b_2 \qquad a_2^{[1]} = \sigma(z_2^{[1]})$$

$$z_3^{[1]} = w_3^{[1]T} x + b_3 \qquad a_3^{[1]} = \sigma(z_3^{[1]})$$

$$z_4^{[1]} = w_4^{[1]T} x + b_4 \qquad a_4^{[1]} = \sigma(z_4^{[1]})$$

Vectorization

$$\begin{bmatrix} w_1^{[1]T} \\ w_2^{[1]T} \\ w_3^{[1]T} \\ w_4^{[1]T} \end{bmatrix}_{4\times 3} = W^{[1]}$$

- we can think of it as we have 4 logistic regression units there and each has a parameter vector w of size 3 and by stacking them together we end up with a 4×3 matrix.

$$W^{[1]} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1^{[1]} \\ b_2^{[1]} \\ b_3^{[1]} \\ b_4^{[1]} \end{bmatrix} = \begin{bmatrix} w_1^{[1]T} x + b_1^{[1]} \\ w_2^{[1]T} x + b_2^{[1]} \\ w_3^{[1]T} x + b_3^{[1]} \\ w_4^{[1]T} x + b_4^{[1]} \end{bmatrix}$$

$b^{[1]}$

$$= \begin{bmatrix} z_1^{[1]} \\ z_2^{[1]} \\ z_3^{[1]} \\ z_4^{[1]} \end{bmatrix} = Z^{[1]}$$

$$a^{[1]} = \begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \\ a_3^{[1]} \\ a_4^{[1]} \end{bmatrix} = \sigma(Z^{[1]})$$

recap:

$X = a^{[0]}$

$\hat{y} = a^{[2]}$

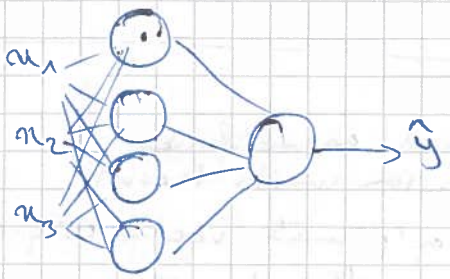
$z^{[1]} = W^{[1]} x + b^{[1]}$ — (4,1) (4,3) (3,1) (4,1)

$a^{[1]} = \sigma(z^{[1]})$ — (4,1) (4,1)

$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$ — (1,1) (1,4) (4,1) (1,1)

$a^{[2]} = \sigma(z^{[2]})$ — (1,1) 1,1

Vectorizing across Multiple Examples



$$\begin{cases} z^{[1]} = W^{[1]} a + b^{[1]} \\ a^{[1]} = \sigma(z^{[1]}) \\ z^{[2]} = W^{[2]} a^{[1]} + b^{[2]} \\ a^{[2]} = \sigma(a^{[2]}) \end{cases}$$

for $i = 1$ to m:

$x \rightarrow a^{[2]} = \hat{y}$

$x^{(1)} \rightarrow a^{[2](1)} = \hat{y}^{(1)}$

$x^{(2)} \rightarrow a^{[2](2)} = \hat{y}^{(2)}$

$\vdots$

$x^{(m)} \rightarrow a^{[2](m)} = \hat{y}^{(m)}$

$a^{[2](m)}$: layer 2, training example m

$z^{[1](i)} = W^{[1]} x^{(i)} + b^{[1]}$

$a^{[1](i)} = \sigma(z^{[1](i)})$

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

$a^{[2](i)} = \sigma(z^{[2](i)})$

Let's get rid of this for loop by vectorization.

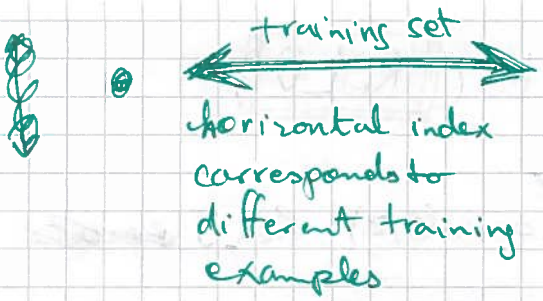
$$X = \begin{bmatrix} | & | & & | \\ x^{(1)} & x^{(2)} & \cdots & x^{(m)} \\ | & | & & | \end{bmatrix}$$

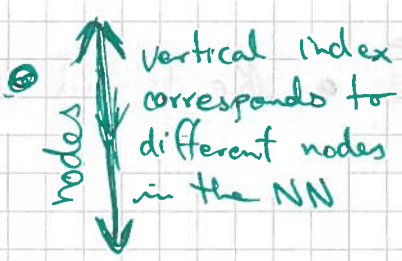
(n_x, m)

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = \sigma(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = \sigma(Z^{[2]})$$

Vectorized Forward pass :)

$$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix} \qquad A^{[1]} = \begin{bmatrix} | & | & & | \\ a^{[1](1)} & a^{[1](2)} & \cdots & a^{[1](m)} \\ | & | & & | \end{bmatrix}$$

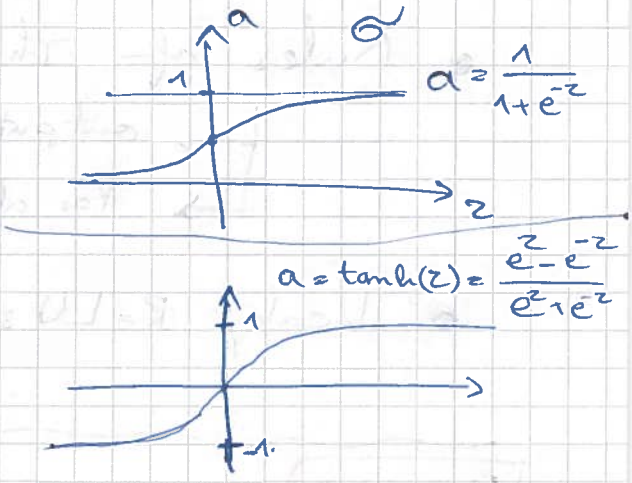


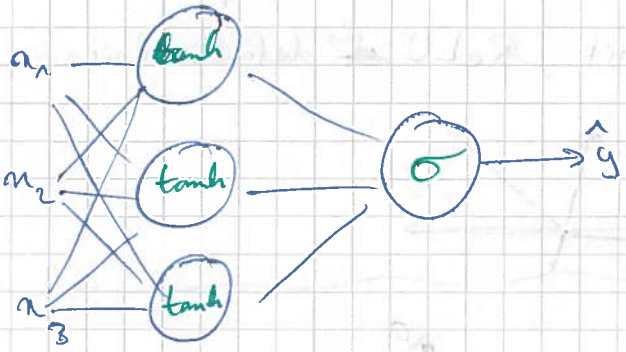


Explanation for the Vectorized implementation

↳ Obvious! See the video!

Activation Function:





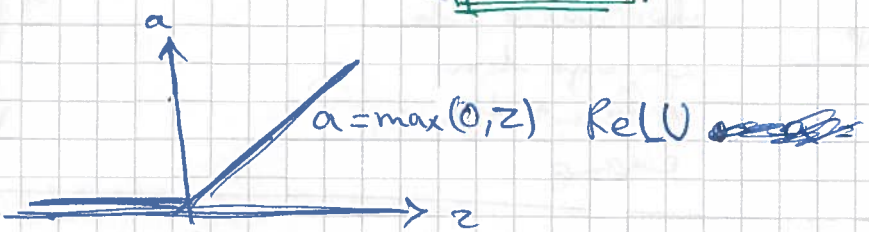
activation function

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

- tanh is like shifted scaled version of σ
- tanh is almost always superior to σ because it has a mean of 0 and centers the data in hidden layers
- use σ at the output layer when $y \in \{0,1\}$ (you want) $\Rightarrow 0 \leq \hat{y} \leq 1$
- a downside of both tanh and σ is if z is very large or z is very small, the slope of the function nears the zero so this slow downs the GD

→

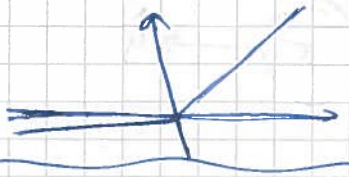
- Rectified Linear Unit (ReLU)



- the derivitative is 1 when $z>0$, and 0 when $z<0$
- in theory the function is not deffrontiable (at $z=0$), but in practice this is not important.

- Rules of thumb:
  - output is $\{0,1\}$ $\Rightarrow$ σ for output
  - for all other units, ReLU is default choice

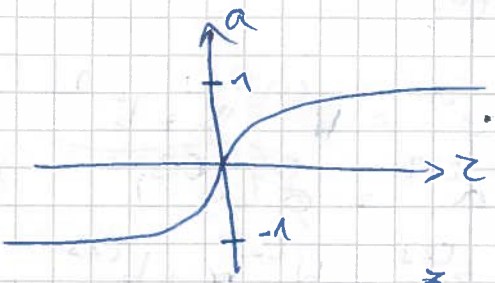
- Leaky ReLU:



Recap:

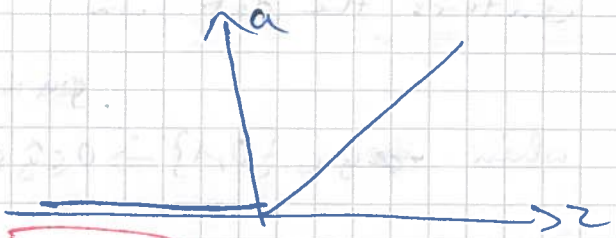
σ Sigmoid $a = \frac{1}{1+e^{-z}}$

- almost never use except for output layer of a $\{0,1\}$ classification



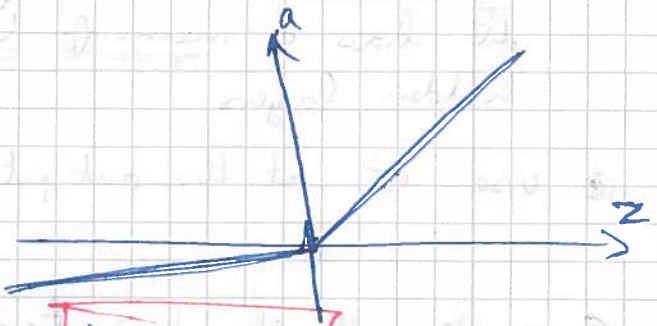
tanh: $a = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$

- always superior to σ



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

- default
- the most commonly used
- if not sure, use ReLU



Leaky ReLU: $a = \max(0.01, z)$

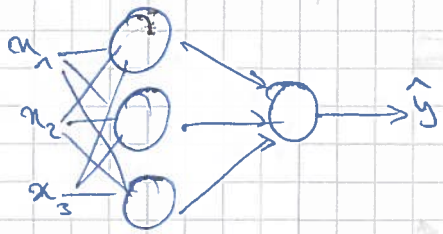
- sometimes better
- not very common
- try it, if better use it

Advise:

- When starting a new NN project, there are a lot of choices on how to build the network:
  - number of hidden units
  - choice of the activation function
  - how to initialize the choices.
- There are common practices in the industry. But it always depend on the specific problem.
- So good is to first try different choices on a smaller data set, and see how it works

Development Set

Why Do we need a sort of Non-linear Activation Function



Given x

$$z^{[1]} = W^{[1]} x + b^{[1]}$$
$$a^{[1]} = g(z^{[1]})$$

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

why not get rid of g function and just say $a^{[1]} = z^{[1]}$ or let's say alternatively $g(z) = z$ "called "linear activation function"

better name would be an "identity activation function"

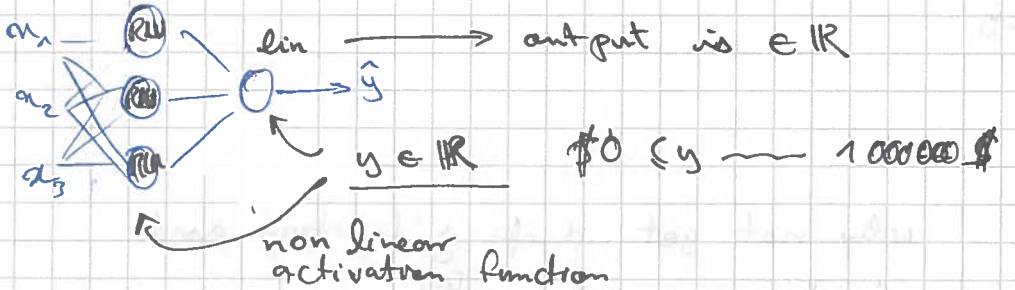
If we do this, then our model just computes $\hat{y}$ as a linear function of our input features:

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

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

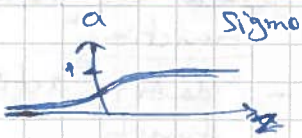
$$\Rightarrow a^{[2]} = \hat{y} = W' x + b'$$

- the network only outputs a linear function of the input
- no matter how many layer you have, the output is always a linear function of the input, so you could simply achieve it without any hidden layers
- if you have linear activation in hidden layers and a sigmoid at last node, the the model is not more expressive than a normal logist regression model
- One case you want to have a linear activation function is doing ML on a "Regression" problem like estimating the price of a house



Derivitives of Activation Functions

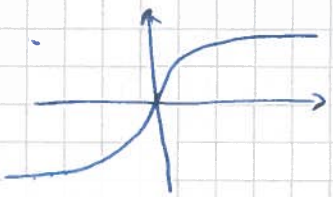
sigmoid



$$g(z) = \frac{1}{1+e^{-z}} \rightsquigarrow \frac{dg}{dz} = g(z)\,(1-g(z))$$

$$g' = g(1-g) = a(1-a)$$

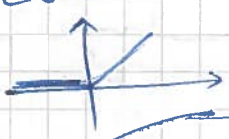
tanh:



$$g(z) = \frac{e^{z} + e^{-z}}{e^{z} + e^{-z}} \rightsquigarrow \frac{dg}{dz} = 1 - (\tanh(z))^2$$

$$g' = 1 - g^2 = 1 - a^2$$

ReLU



$$g(z) = \max(0,z) \rightsquigarrow g'(z) = \frac{dg}{dz} = \begin{cases} 0 & z<0 \\ 1 & z \geq 0 \\ \text{undef} & 0 \end{cases}$$

$$g' = \begin{cases} 0 \\ 1 \end{cases}$$

for Leaky ReLU: $g(z) = \max(0.01, z) \rightarrow g' = \begin{cases} 0.01 \\ \end{cases}$

Gradient Descent for NN

Neural Network with a single hidden layer:

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

$(n^{[1]}, n^{[0]})$ $(n^{[1]}, 1)$ $(n^{[2]}, n^{[1]})$ $(n^{[2]}, 1)$

$n_x = n^{[0]}, n^{[1]}, n^{[2]}=1$

Assuming we are doing a binary classification:

Cost Function: $J(W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}, y)$ ($\hat{y} = a^{[2]}$)

Gradient Descent:

- init Params randomly
- Repeat {
  - compute the predictions ($y^{(i)}$, $i=1 \dots m$)
  - compute the derivatives ($dW^{[1]} = \frac{dJ}{dW^{[1]}}$, $db^{[1]} = \frac{dJ}{db^{[1]}}$, ...)
  - update gradient descent

$$W^{[1]} = W^{[1]} - \alpha\, dW^{[1]}$$
$$b^{[1]} = b^{[1]} - \alpha\, b^{[1]}$$
$$W^{[2]} = W^{[2]} - \alpha\, dW^{[2]}$$
$$b^{[2]} = b^{[2]} - \alpha\, b^{[2]}$$

}

Forward Prop. equations:

$$Z^{[1]} = W^{[1]} X + b^{[1]}$$
$$A^{[1]} = g^{[1]}(Z^{[1]})$$
$$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$$
$$A^{[2]} = g^{[2]}(Z^{[2]}) \longrightarrow \text{bin. class.} \Rightarrow g^{[2]} = \sigma$$

Bwd Prop.:

$$dZ^{[2]} = A^{[2]} - Y \qquad Y = [y^{(1)}\ y^{(2)} \dots y^{(m)}]$$
$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$$
$$db^{[2]} = \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis}=1, \text{keepdims}=\text{True})$$

horizontal

output not rank 1 $(n,)$ this. output $(n^{[2]}, 1)$

$$dZ^{[1]} = \underbrace{W^{[2]T} dZ^{[2]}}_{(n^{[1]}, m)} * \underbrace{g^{[1]\prime}(Z^{[1]})}_{(n^{[1]}, m)}$$

elementwise prod

deriv. of $g^{[1]}$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$$
$$db^{[1]} = \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis}=1, \text{keepdims}=\text{True})$$

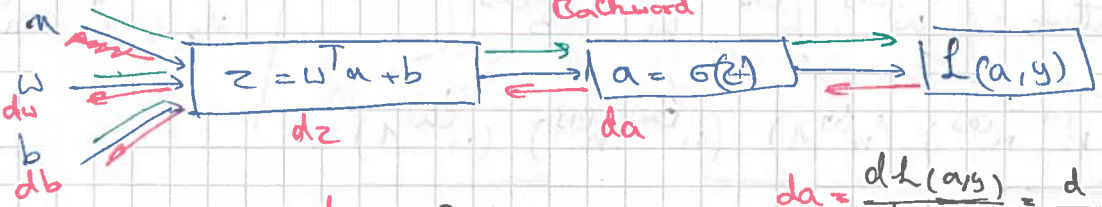
$(n^{[1]}, 1)$

no rank 1 array $(n^{[1]}, 1)$

Backpropagation Intuition

Logistic Regression

For one Training Example



$da = \frac{dL(a,y)}{da} = \frac{d}{da}\left(-y\log a - (1-y)\log(1-a)\right)$

$= -\frac{y}{a} + \frac{1-y}{1-a}$

$dz = a - y$

$dw = dz \cdot x$

$db = dz$

$dz = da \cdot g'(z)$

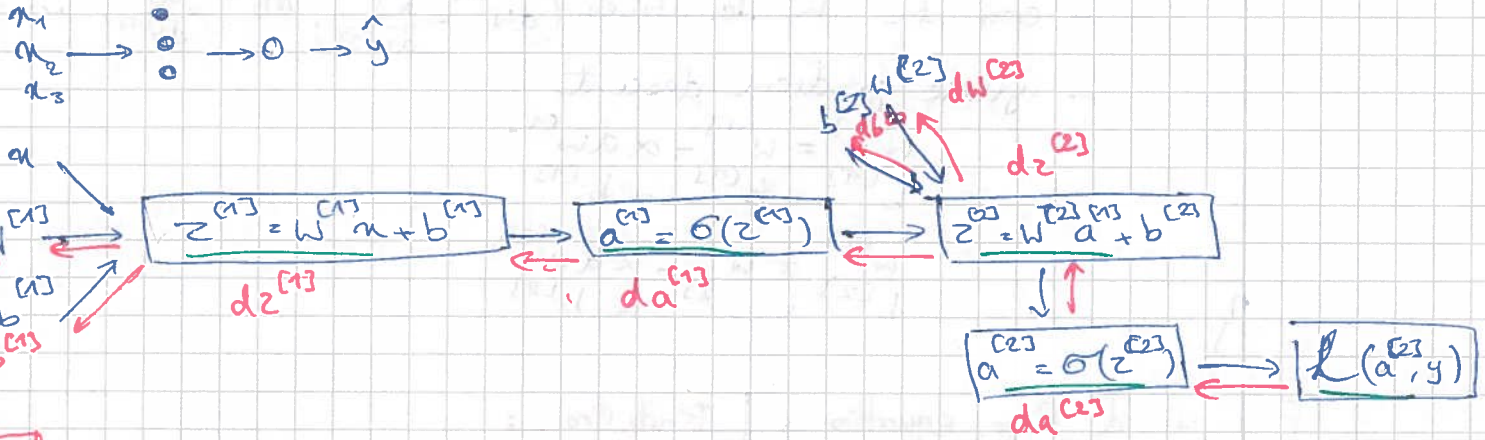
$a = g(z) = \sigma(z)$

$g'(z) = a(1-a)$

$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial z}$

"dz" $= da \cdot g'(z)$

- calculation for a NN will be very similar to Log. Regression, unless we do it twice



$dz^{[2]} \quad dz^{[2]} = a^{[2]} - y$

$dW^{[2]} \quad dw^{[2]} = dz^{[2]} a^{[1]T}$ → quite similar to log. reg.: "$dw = dz \cdot x$" now $a^{[1]}$ plays the role of x here

$db^{[2]} \quad db^{[2]} = dz^{[2]}$

$dw^{[2]}$ is here a row vector $dw^{[2]} = [\quad\quad]$ in the case of log. reg. with a single output. Whereas dw was a column vector.

so $a^{[1]}$ has a Transpose

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

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

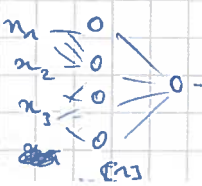
same dimension as $z^{[1]}$

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

$a^{[2]}, z^{[2]}, dz^{[2]} : (n^{[2]},1)$

$a^{[1]}, z^{[1]}, dz^{[1]} : (n^{[1]},1)$

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



$n = n^{[0]}$ $n^{[2]} = 1$

* Tip: when implementing back prop. just making sure that the dimensions match up, eliminates sometimes a lot of bugs already.

$dW^{[1]} = dz^{[1]} \cdot x^T$

$db^{[1]} = dz^{[1]}$

} Very similar to $dw^{[2]}$ and $db^{[2]}$ because X plays the same role here as $a^{[1]}$ ($a^{[0]}$)

Recap: for a single training example:

$dz^{[2]} = a^{[2]} - y$

$dw^{[2]} = dz^{[2]} a^{[1]T}$

$db^{[2]} = dz^{[2]}$

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

$dw^{[1]} = dz^{[1]} x^T$

$db^{[1]} = dz^{[1]}$

Vectorized Back prop.

$dZ^{[2]} = A^{[2]} - Y$

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$

$db^{[2]} = 1/m \; np.sum(dZ^{[2]}, axis=1, keepdims=True)$

$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]})$

$(n^{[1]}, m) \quad (n^{[2]}, m) \quad (n^{[1]}, m)$

$dW^{[1]} = 1/m \; dZ^{[1]} X^T$

$db^{[1]} = 1/m \; np.sum(dZ^{[1]}, axis=1, keepdims=True)$

$Z^{[1]} = \begin{bmatrix} | & | & & | \\ z^{[1](1)} & z^{[1](2)} & \cdots & z^{[1](m)} \\ | & | & & | \end{bmatrix}$, $A^{[1]} = g^{[1]}(Z^{[1]})$

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

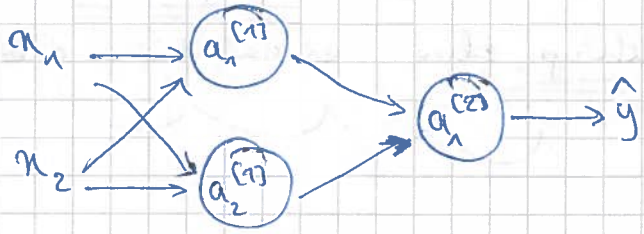
Same for Back Prop.

Initializing the Weights

- for NN the weight must be initialized "randomly"

In logistic Regression we could do it by init to zero, but it does not work in NN. Why? ↓

→



$n^{[0]} = 2$ $\quad n^{[1]} = 2$

$$W^{[1]} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \qquad b^{[1]} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$(2,2)$

$$a_1^{[1]} = a_2^{[1]} \qquad dz_1^{[1]} = dz_2^{[1]} \;, \Rightarrow W^{[2]} = [0\ 0]$$

- if the initial weights for both hidden units are the same, then both units compute the same function. **Symetric**
- by induction after several iteration they still computing the same function } $\Rightarrow$ it is like having only **1 single hidden unit**.

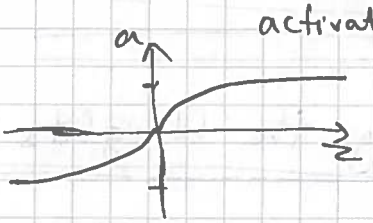
- solution:
  - $\hookrightarrow$ initialize the weight **randomly**. You want your hidden units to compute different functions.

$$W^{[1]} = np.random.randn((2,2)) * 0.01$$

$\hookrightarrow$ so that weights are small, because if activation function tanh or sigmoid

$$b^{[1]} = np.zero((2,1))$$

b is not important because we initialized W randomly



$\Rightarrow$ W large
$\Rightarrow$ z large
$\Rightarrow$ slope ≈ 0
$\Rightarrow$ GD very slow
$\Rightarrow$ learning slow

$$W^{[2]} = np.random.randn((1,2)) * 0.01$$

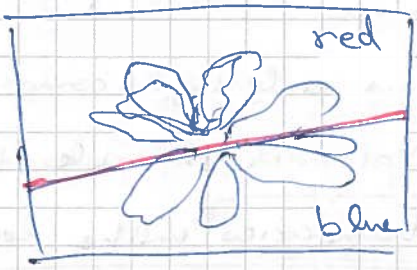
$$b^{[2]} = 0$$

Programing Assignment 3: Planar Data Classification with one hidden layer

- Cross entropy loss
- Scikit Learn:

```
clf = sklearn.linear_model.LogisticRegressionCV()
clf.fit(X.T, Y.T)
```

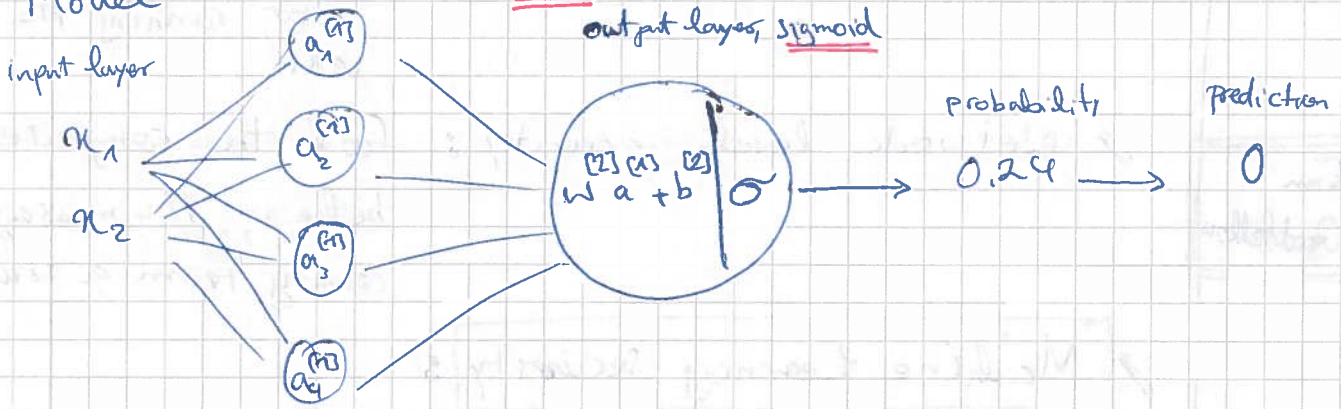
- plot decision boundary



Accuracy = 47%

- linear Regression does not perform well because the data is not linearly separable

NN Model



for one example $x^{(i)}$

$$z^{[1](i)} = W^{[1]} a^{(i)} + b^{[1]}$$

$$a^{[1](i)} = \tanh(z^{[1](i)})$$

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

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \; ; \quad y^{(i)}_{\text{prediction}} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

Cost function:

$$J = -\frac{1}{m} \sum_{0}^{m} \left(y^{(i)} \log(a^{[2](i)}) + (1-y^{(i)}) \log(1-a^{[2](i)}) \right)$$

→

General Methodology to Build a NN:

1. Define the NN structure (# input units, # hidden units, etc.)
2. Initialize the model parameters
3. Loop
   - Impl forward prop.
   - Compute loss
   - Impl Backward prop. to get the gradients
   - update params (gradient descent)

- Accuracy is really high (90%) compared to Logistic Regression (47%) &
- Neural Networks are able to learn even highly non-linear decision boundaries unlike Logistic Regression

- post on Github / ArXiv

// Application level security: fool the computer into running the wrong code

// Network level security: fool the computer into believing the messages are coming from a trusted party.

Ian Goodfellow

// Machine Learning security:
↳ Adversarial Networks

Week 4

Deep Neural Networks

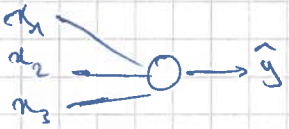
Learning Objectives:

- See deep neural Networks as successive blocks put one after each other
- Build and train deep L-layer NNs
- Analyze Matrix & vector dimensions to check NN impls
- Understand how to use a cache to pass information from forward prop. to backprop.
- Understand the role of hyperparameters

Deep L-layer neural Networks

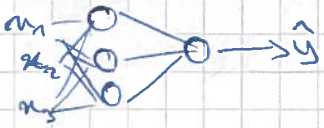
- forward prop., back prop., log. regression, NN with single hidden layer, Vectorization, init weights randomly

- What is a deep NN?

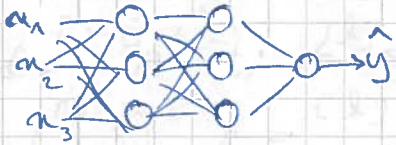


Logistic Regression

"shallow model"



1 hidden layer

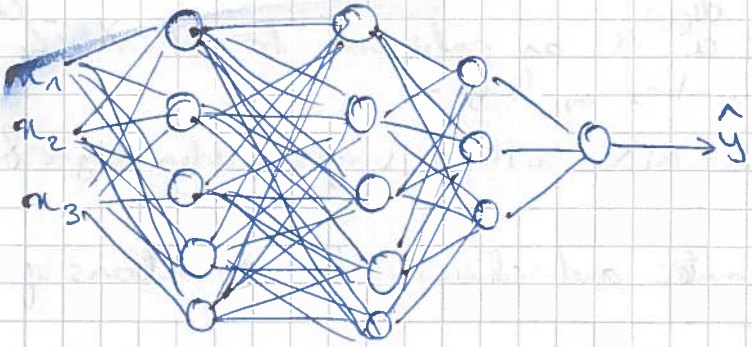


2 hidden layers

5 hidden layers "deep"

- # hidden layers: another example of hyperparameter

Deep NN Notation



4 layer NN
with 3 hidden Layers.

$L = 4$ (# layers)

$n^{[l]}$ (# units in layer l): $n^{[1]} = 5$, $n^{[2]} = 5$, $n^{[3]} = 3$, $n^{[4]} = n^{[L]} = 1$

$n^{[0]} = n_x = 3$

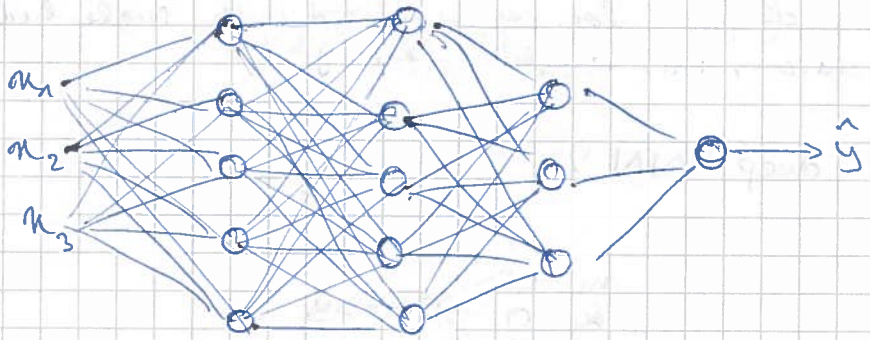
$a^{[l]}$ (activations in layer l)

$a^{[l]} = g^{[l]}(z^{[l]})$, $w^{[l]}$ (weights for $z^{[l]}$), $b^{[l]}$

$x = a^{[0]}$

$a^{[L]} = \hat{y}$

Forward Propagation in a Deep Network:



x; Single training example:

$z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$

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

⋮

$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$
$a^{[4]} = \hat{y} = g^{[4]}(a^{[4]})$

! ⇝ $\hat{Y} = g^{[4]}(Z^{[4]}) = A^{[4]}$

General forward Prop. equation:

$Z^{[l]} = W^{[l]} a^{[l-1]} + b^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$

Vectorized Version:

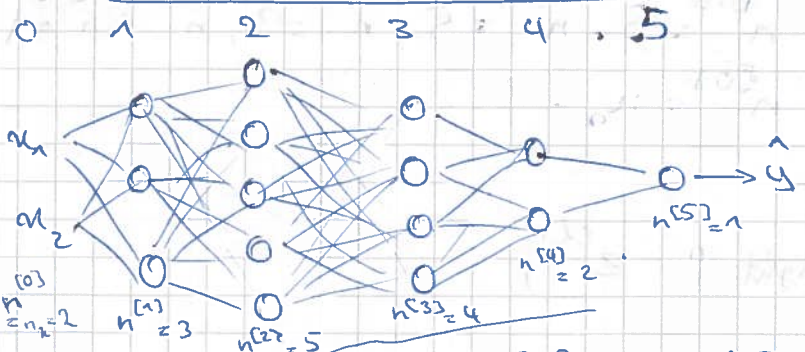
$X = A^{[0]}$

$Z^{[1]} = W^{[1]} X + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$
⋮

for $l = 1 - L$

- stacking $x^{(i)}$'s, $z^{(i)}$'s, $a^{(i)}$'s as columns to get X, $Z^{[l]}$, $A^{[l]}$
- we need a for of number of layers
- very similar to a NN with a single hidden layer & repeating that.
- We need to systematic and checking the dimentions of matrices

Getting your Matrix Dimensions right:



$z^{[1]} = W^{[1]} x + b^{[1]}$ ↳ let's ignore b for now.

$(3,1) = (3,2)(2,1)$
$(n^{[1]},1) = (n^{[1]}, n^{[0]})(n^{[0]},1)$

$\begin{bmatrix} \cdot \\ \cdot \\ \cdot \end{bmatrix} = \begin{bmatrix} \cdot & \cdot \\ \cdot & \cdot \\ \cdot & \cdot \end{bmatrix} \begin{bmatrix} \cdot \\ \cdot \end{bmatrix}$

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

$W^{[2]} : (n^{[2]}, n^{[1]}) \longrightarrow (5,3)$

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

$(5,1) = (5,3)(3,1)$

$W^{[3]} : (4,5) \longrightarrow (n^{[4]}, n^{[3]})$

$W^{[4]} : (2,4)$, $W^{[5]} : (n,2)$

General Rule

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

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

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

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

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

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

Vectorized:

- dimensions of W, b, dW, db stay the same
- Z, A, X :

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

$(n^{[1]}, m) = (n^{[1]}, n^{[0]})(n^{[0]}, m) \quad (n^{[1]}, 1)$ broadcasting $\rightsquigarrow (n^{[1]}, m)$

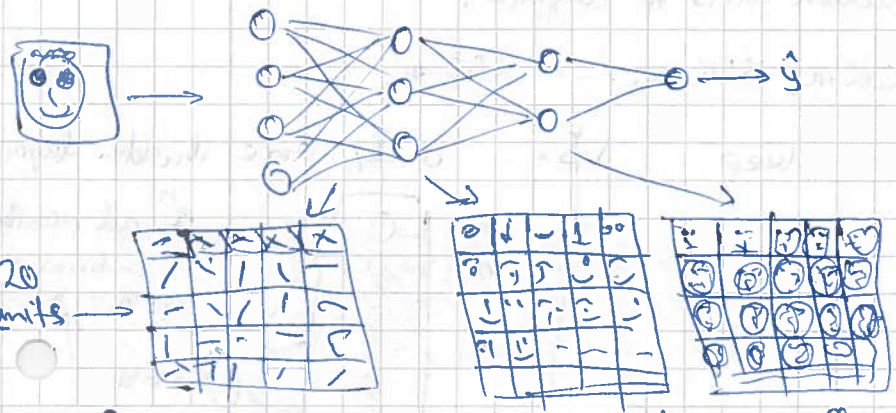
$Z^{[l]}, A^{[l]} : (n^{[l]}, m)$

$l = 0, A^{[0]} = X : (n^{[0]}, m)$

$dZ^{[l]}, dA^{[l]} \; (n^{[l]}, m)$

Why deep representations work?

- The NN does not to be big but needs to be deep
- what does a NN compute?



first layer detects edges grouping together pixels

detects parts of face grouping together edges

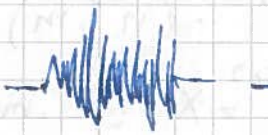
different types of face by grouping parts of face

- think:
  - Earlier layers of NN compute simpler functions
  - later layers compute more complex function by composing those simpler functions.

edge detectors — the simple function look at smaller areas of image, complex functions look at much larger areas

> Simple to complex, hierachical, compositional representation

- This applies to other kinds of data like Audio as well



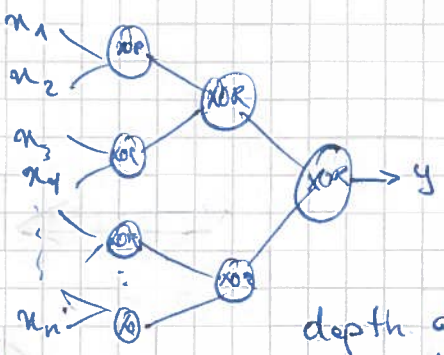
→ low level audio wave form features —compose→ basic units of sounds "Phonems": "c", "a" —compose→ words —compose→ sentence/phrases

- The deeper you go int the network, the more complex things can be detected.
- Some compare it to human brain. Although that is not sure

Circuit Theory: Informally: there are functions you can compute with a "small" L-layer deep neural network that shallower networks require exponentially more hidden units to compute.

Example: $y = x_1 \text{ XOR } x_2 \text{ XOR } x_3 \cdots \text{ XOR } x_n$

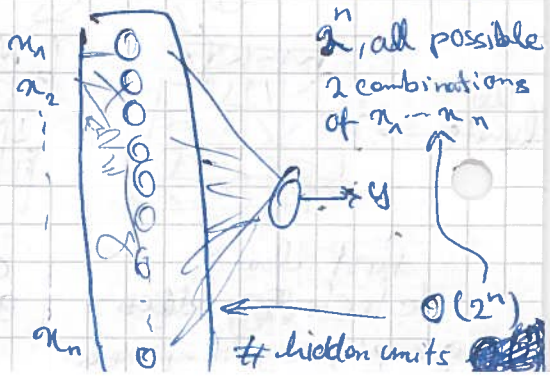
deep



depth of XOR tree: $O(\log n)$

units: $2^{\log n}$

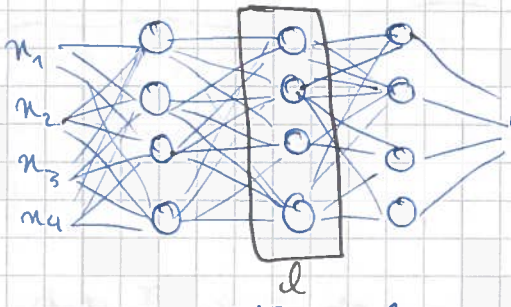
vs. only one hidden layer

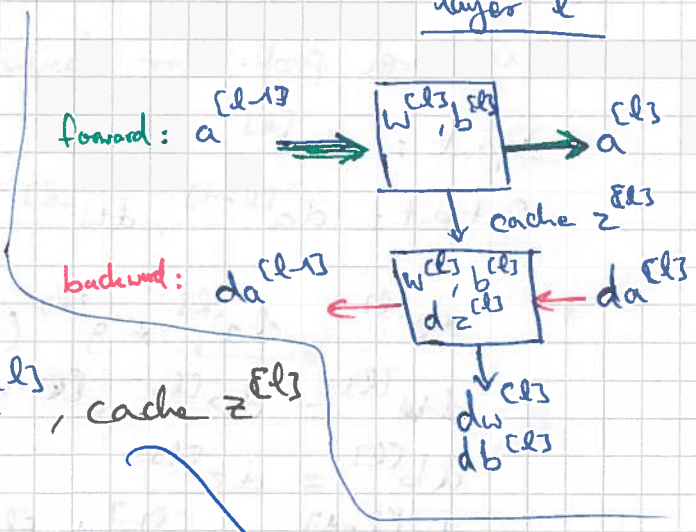


- Other reason for deep learning taking off is "Branding" :)
  It is basically NN or NN with multiple layers rebranded.

- **Tip:** don't get excited. Don't start with many many layers. Start even with log. regression. Then add other layers. and see how your model behaves.

Building Blocks of Deep Neural Networks





Layer l: $w^{[l]}, b^{[l]}$

Forward: input $a^{[l-1]}$, output $a^{[l]}$, cache $z^{[l]}$

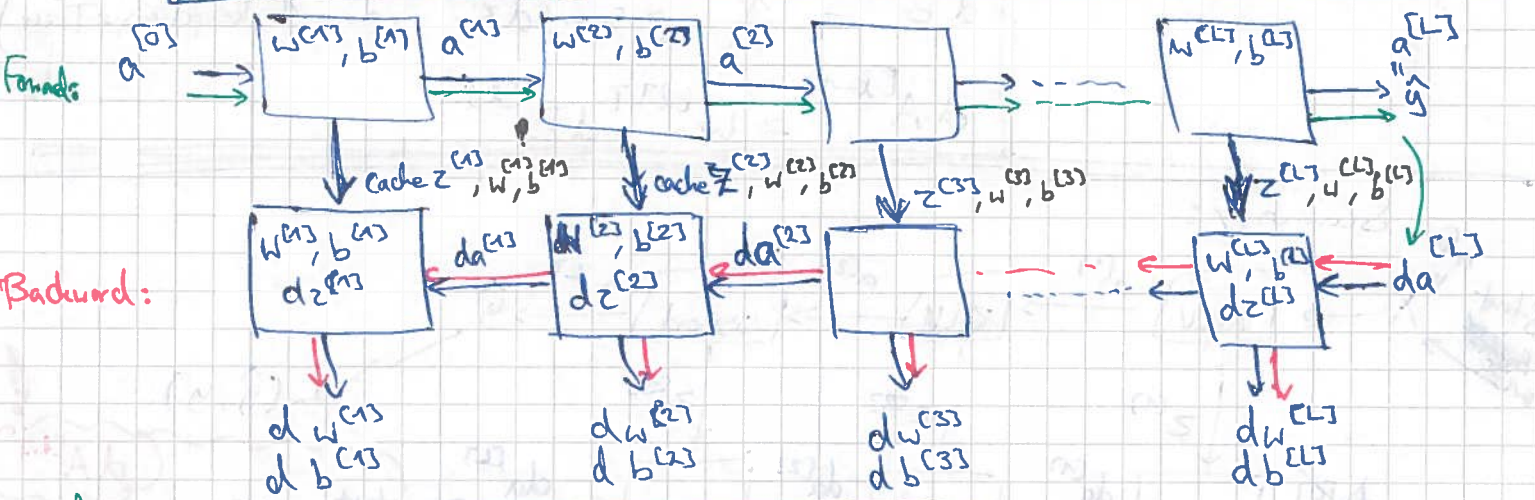
$$z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]}$$

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

Backward: input: $da^{[l]}$ (cached $z^{[l]}$), output $da^{[l-1]}$, $dw^{[l]}$, $db^{[l]}$

For passing Information from forward step to backward step.

One Iteration: forward & Backward for whol Network



! for impl

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

Forward and Backward Propagation:

Forward prop. for layer l

Input $a^{[l-1]}$ ($W^{[l]}, b^{[l]}$)

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

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

Vectorized:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = g^{[l]}(Z^{[l]})$$

$X = A^{[0]} \rightarrow [\ ] \rightarrow [\ ] \rightarrow \cdots [\ ] \rightarrow$

Backward prop. for layer l:

Input: $da^{[l]}$

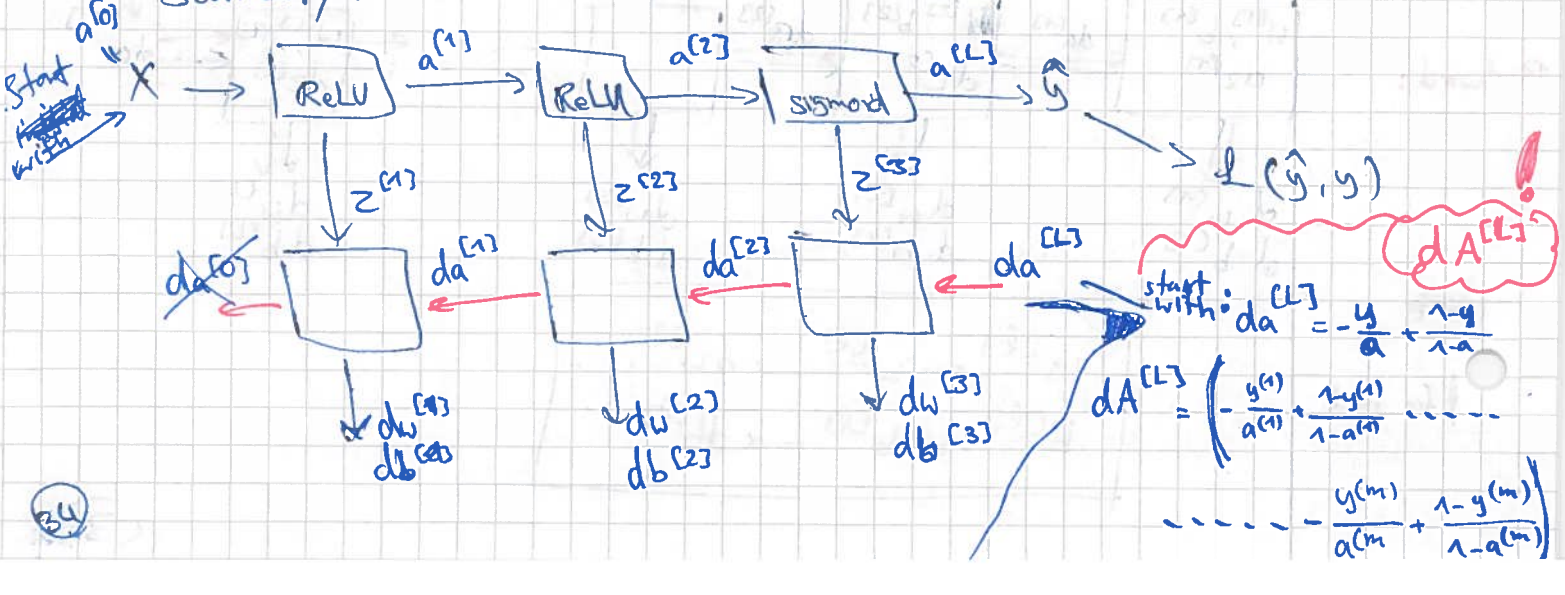
Output: $da^{[l-1]}, dw^{[l]}, db^{[l]}$

$$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]}) \rightsquigarrow dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$$
$$dw^{[l]} = dz^{[l]} \cdot a^{[l-1]}$$
$$db^{[l]} = dz^{[l]}$$
$$da^{[l-1]} = W^{[l]T} dz^{[l]}$$

Vectorize:

$$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$$
$$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$$
$$db^{[l]} = \frac{1}{m} np.sum(dZ^{[l]}, axis=1, keepdims=True)$$
$$dA^{[l-1]} = W^{[l]T} dZ^{[l]}$$

Summary:



- Even Andrew Ng is sometimes surprized when the code works :)

Because a lot of complexity in ML comes from data not from thousands of lines of code

---

Parameters vs. Hyperparameters:

Parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \dots$

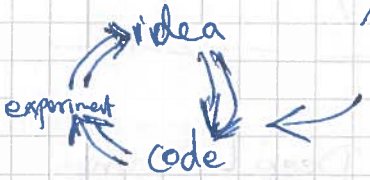
Hyperparameter: learning rate α
iterations
hidden layers L
hidden units $n^{[1]}, n^{[2]}, \dots$
choice of activation function

Hyperparameters control the ultimate parameters b's and W's.

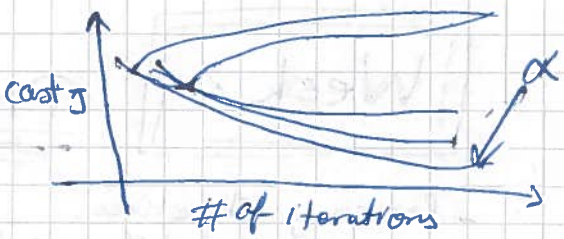
Deep learning has a lot of Hyperparameters:

Other examples:
Momentum, mini-batch size, regularization params, ...

- Values for Hyperparameters can have many different setting that we need to try-out !



Applied deep learning is very impersical



- it is very difficult to know in advance what to choose for hyperparameters, so try a lot of things and see how it works

- Some times the intuition for Hyperparam from one application domain does not carry over. So you have to try a range of values
- Also as the HW progress, the HyParam that was very good, might no more be the best

Deep Learning and Human Brain :

- Deep learning is a very good way to learn flexible complex functions $y = f(x)$, but it does not have much to do with how brain works.
- Actually we still don't even know how a single neuron works/learn
- there are over simplified analogies of NNs and brains or a single logistic regression unit and a single neuron but they are for general public

neuron does a threshold computing on input and then fires

- We still don't know weather learning in brain is like gradient descent or there is a foundamentally different process ?

Course 2 (C2 - Improving Deep Neural Networks, Hyperparameter tuning, Regularization, and Optimization)

Week 1 Practical Aspects of Deep Learning

- learning objectives:
  - Recall that different kinds of initialization lead to different results.
  - Recognize the importance of initialization in complex neural networks
  - Recognize the difference between train/dev/test sets
  - Diagnose the Bais and variance issues in your model
  - Learn how to use ~~Regularization~~ methods such as dropout & L2 Regularization Regularization
  - Understand experimental issues in deep learning such as vanishing and Exploding gradients and learn how to deal with them.
  - use gradient checking to verify the correctness of your backprop implementation

Setting up your Machine Learning Application

Train / Dev / Test sets:

- Hyperparameter tuning
- setting up data
- make sure the optimization Alg. runs quickly.

Applied ML is a highly iterative process:

- # layers
- # hidden units
- learning rate
- activation function

Start with an idea, and try it, and improve it based on outcom

NLP, vision, Speech, Structured data → Ads, search, security, logistics ...

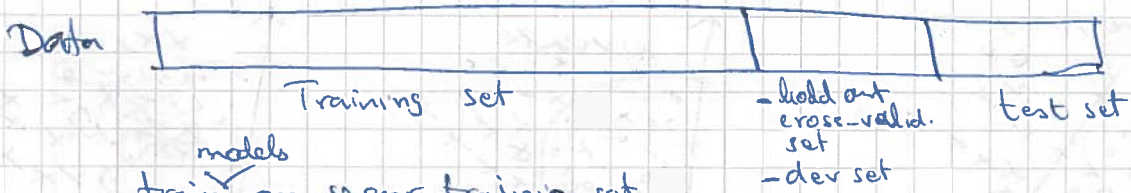
Intuitions from one Application area often donot transfer to other application areas.

A correct guess of correct/good hyperparameter values at the first time is almost impossible

⟹ do iterations ⟹ make iterations faster and more efficient.

- Setting up the data set correctly can make the iteration cycle much more efficient

Train / dev / Test sets:



- train models on your training set
- use dev set to see which of models performs best
- use test set to evaluate the final best model.

Previous era: 70/30 %. 60/20/20 %.

Big data era: 1,000,000 examples ⟹ 10,000 → 98/1/1 %, or even 99/0.5/0.5 %.

Mismatched Train/test distribution

Cats

Training set: Cat pictures from Internet

X

Dev/test sets: Cat pictures from users

Different distributions: Resolution/size/color/frame/...

- Make sure that **Dev/Test** sets come from **the same distribution**!
- Not having a test set might be ok. (only dev set)
- make sure to distinguish between dev/test set.
  Dev set is the "hold out" cross-validation set.

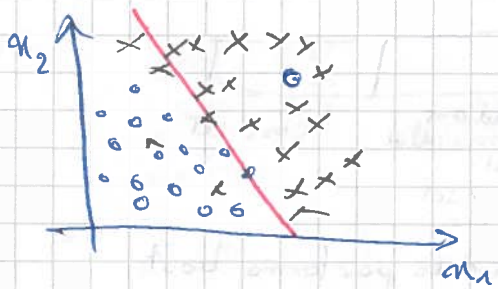
Because Sometimes you overfit dev set.

- Train/Dev/Test sets allow you to iterate more quickly and more efficiently measure the bias and variance of your algorithm

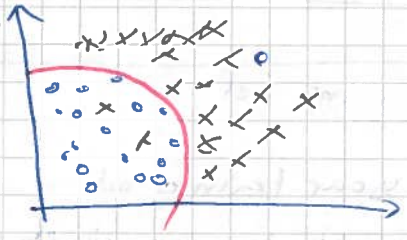
---

Bias and Variance:

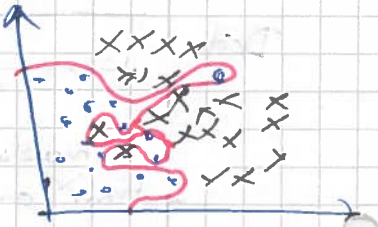
- good ML/DL practioners have sophisticate understanding of Bias & Variance
- Bias & Variance are easily learned but difficult to master concepts
- In DL era there is less discussion of Bias-Variance trade-off
  - less discussion of trade-off



underfitting
high bias



just right



overfitting
high variance

Cat Classification:

Two key values to look at

• Train set error:	1 %	15 %	15 %	0.5 %
• Dev set error:	11 %	16 %	30 %	1 %
	- very well on training set - poorly on Dev set - overfitting - not generalizing well on the dev set	- not even doing well on train set - underfitting - high bias	- worst of both worlds	
	"high Variance"	"high Bias"	high bais high variance	low bais low variance

Assuming humans ≈ 0% error

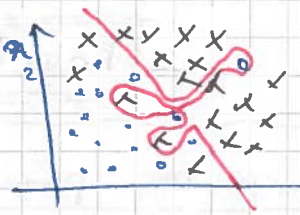
optimal error ≈ 0%
(Bayes Error)

If Bayes Error was 15%. then the classifier with 15% error was doing very well

! The relationship between train set & dev set error gives you insight into bais & variance of your mode

High Bias and high variance:

this can happen easily when you have high dimentional data!



- high bias
- high variance
- =(

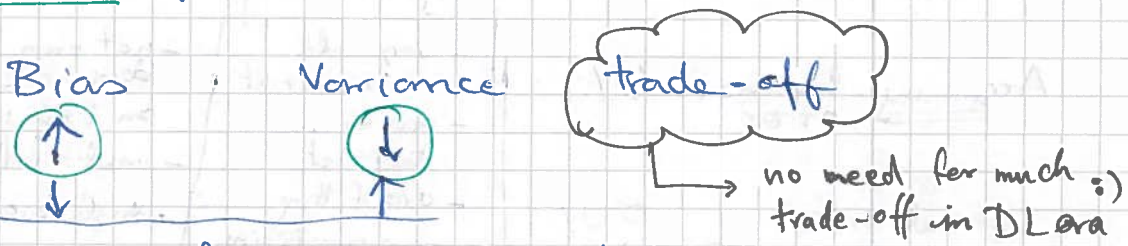
Basic Recipe for Machine learning:

- After training an initial model:
  - high bias? (Permance on training dat)
    - continue iterating till you at least fit the training set
    - yes → Bigger Network
      - more layers
      - more hidden units
      - train longer
      - try another NN architeture
    - no ↓
  - High variance? (dev set performance)
    - yes → more data
      - Regularization
      - try another NN archit.
    - no ↓
  - Done

← • first look at the performance on train & dev sets and then decide what to do

! For example: If you have high Bias adding more data won't help

In Earlier times of ML there was the discussion about



But in DL Era we have now tools that let us reducing both Bias and Variance without charting the other

Regularization:

• If you suspect overfitting (high variance):
  - add more training data → not always possible
    - might be expensiv
  - Regularization often helps

• Regularization penalizes large coefficients of W and shrinks them to zero. This way the coefficients relating to irrelevant features dissapper, and model does not react to noise.

!

Example Logistic Regression:

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

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

Regularization param

$$+ \frac{\lambda}{2m} b^2$$

omit. Not important

L_2 Regularization: $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ ← L_2 norm, Euclidian norm

L_1 Regularization: $\frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m}\|w\|_1$ ⇝ - will make w sparse
- not often used

For a Neural Network:

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} \| w^{[l]} \|_F^2$$

$$\| w^{[l]} \|_F^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{ij}^{[l]})^2 \qquad w: (n^{[l]}, n^{[l-1]})$$

Forbenius Norm $\| \cdot \|_F^2$

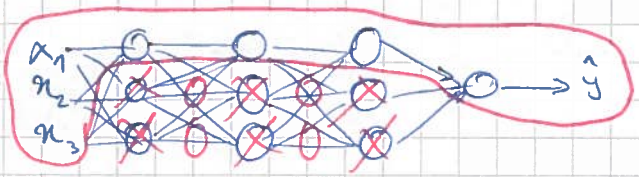
$$\frac{\partial J}{\partial w^{[l]}} = dw^{[l]} = (\text{term from backprop}) + \frac{\lambda}{m} w^{[l]}$$

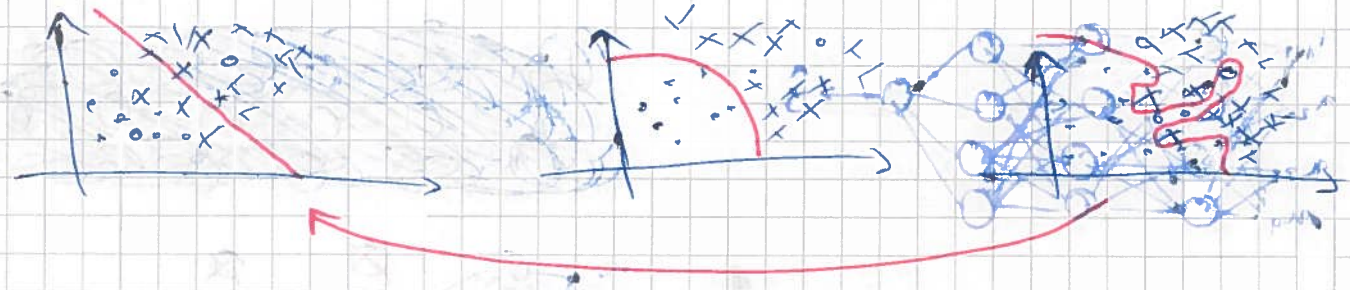
GD: $w^{[l]} := w^{[l]} - \alpha dw^{[l]} = w^{[l]} - \alpha\left[(\text{term from backprop}) + \frac{\lambda}{m} w^{[l]}\right]$

$$= w^{[l]}\left(1 - \frac{\alpha\lambda}{m}\right) - \alpha\,(\text{term from backprop})$$

"Weight decay" ⇝ Alternative name for L_2 Regularization

Why Regularization Reduces Overfitting?





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

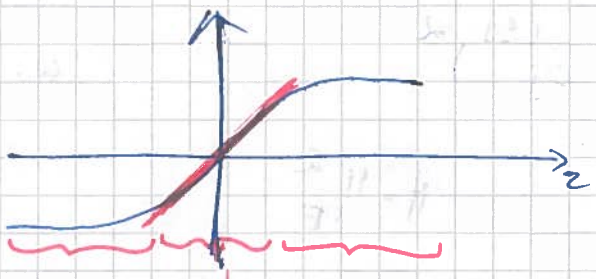
$\lambda \to \infty \Rightarrow w^{[l]} \to 0$

λ large ⇝ $w^{[l]} \approx 0$ ⇝ zeroing out many of the hidden units ⇝ model becomes roughly linear ⇝ ⇝ going from high variance to high bias ⇝ not capable of estimating complex non-linear decision boundaries.

→

Another Intuition

- assume the activation function is tanh



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

$\lambda \uparrow \quad w^{[l]} \downarrow \qquad z^{[l]} = w^{[l]} a^{[l-1]} + b^{[l]} \qquad \downarrow$

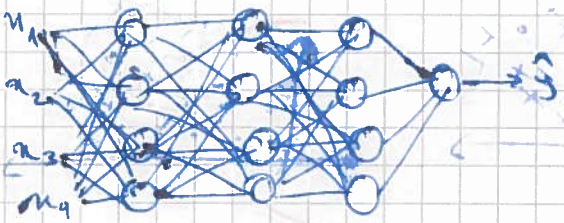
λ high $\rightsquigarrow$ $w^{[l]}$ low $\rightsquigarrow$ $z^{[l]}$ small $\rightsquigarrow$ "linear Regime" of tanh

Every layer $\simeq$ linear $\rightsquigarrow$ model roughly linear

$\rightsquigarrow$ not capable of computing complex non-linear functions

- L2 regularization is used most often

Dropout Regularization

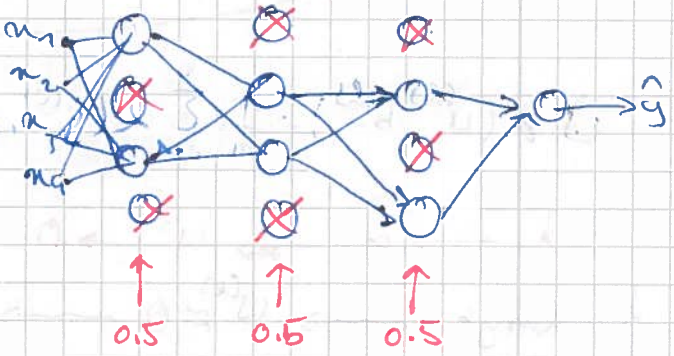


- Dropout:

  just eliminate some of the nodes and their corresponding weights.

  Randomly

  By a given probability



$\Longrightarrow$ Results in a much smaller much deminished Network. Then you do backprop. Then deminish

Implementing "Inverted Dropout"

Take layer 3 for example: $l=3$, keep-prop $= 0.8$

a boolean array

$d3 = np.random.rand(a3.shape[0], a3.shape[1]) < 0.2$ ⇒ eliminating chance $= 0.2$

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

! $a3 \not= keep\text{-}prop$

why?

Imagine you have 50 hidden units at layer 3

⇒ on average at each iteration, you shut-off 10 units

⇒ $z^{[4]} = w^{[4]} a^{[3]} + b^{[4]}$ will be reduced by 20% (in value)

⇒ in order not to change the expected value of $a^{[3]}$ we need to bump it up by 20%.

⇒ hence deviding by keep-prop.

⇒ inverted dropout

! At each iteration of GD, you zero out a different set of units.

! At Test time, prediction:

No Dropout.

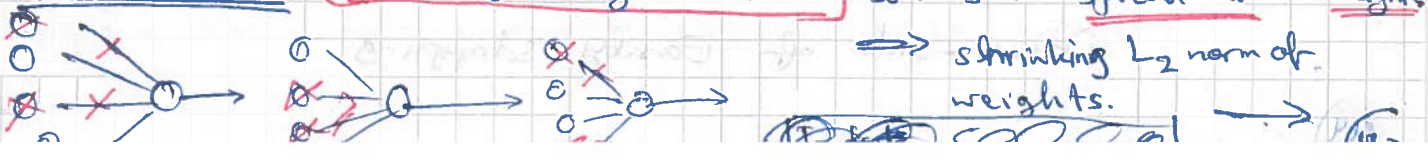
Because you don't want your output to be random. This will add noise to predictions.

Why does Dropout work?

- it seems to be crazy randomly knoking out units in the network.
- why does it work as a regularizer?

Intuition: can't rely on any one feature. So has to spread out the weights.

⇒ shrinking L_2 norm of weights.



← ◎ In fact Dropout is very similar to L_2 Regularization.

◎ You can also have different values for keep-prop for different layer. If a layer is to big (⇒ may overfit) make keep-prop lower.

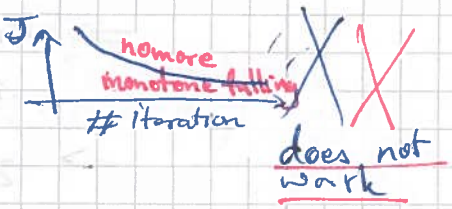
◎ Normally you don't dropout input layer. (keep-prop = 1)

◎ Dropout is often used in computer vision, where input size is too big (⇒ you almost never have enough data)

◎ But does not always generalize to other domains

◎ Down-side of Dropout :

Cost fnct(J) is no longer well-defined ⇒



⇒ first turn-off drop-out

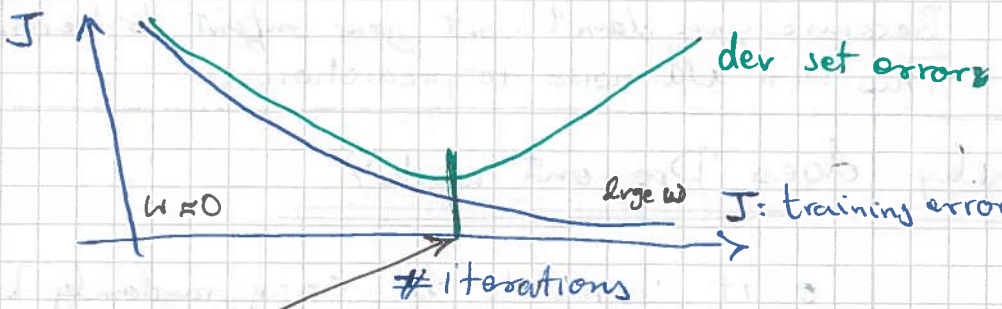
⇒ make sure your J is monotonically decreasing * and your algorithm works

⇒ then turn on drop-out

Other Regularization Methods :

- Data Augmentation !
  - → e.g. flipping images horizontally
  - → random distortion & transformations

- Early Stopping



Stop training here

: this works like L_2 regularization

mid-size w : $\|w\|_F^2$

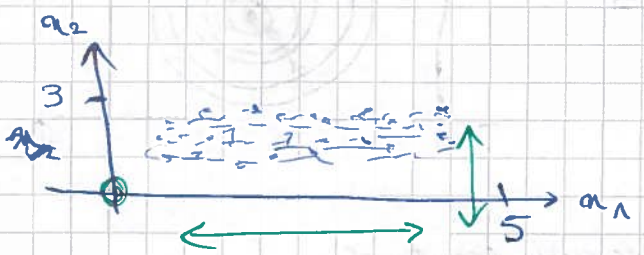
Down-side of Early stopping

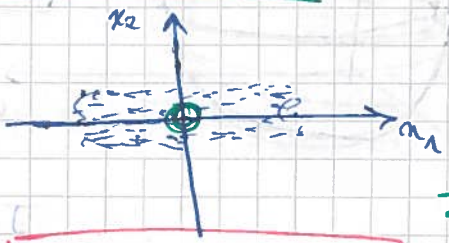
- ML is a process comprising of several steps:
  - optimizing the cost function J
    - Gradient descent
    - RMS ...
  - Not overfitting
    - regularization
- ML is easier if we focus on each problem/step separately (orthogonalization)
- Early stopping mixes the two tasks. It make the set of tools to use more complicated.
- An alternative can be using Regularization using different values of λ

Setting up the Optimization Problem:

Normalizing Inputs:

- a technique to speed-up the training





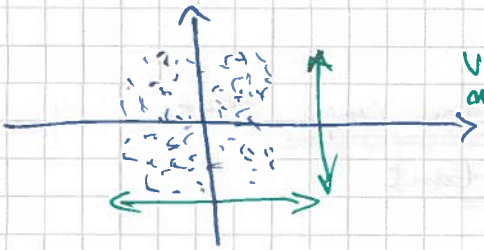
Zero mean

Subtract mean

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

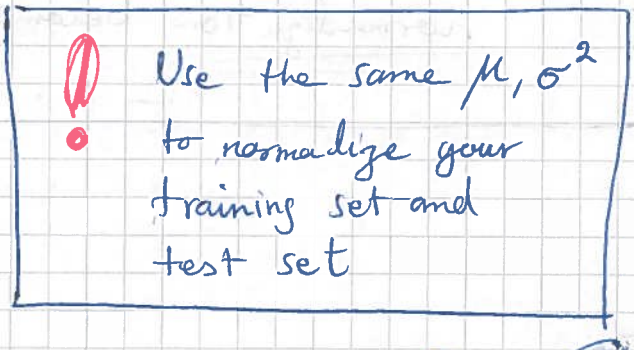
$$x^{(i)} := x^{(i)} - \mu \quad \text{for all the samples.}$$

Normalize Variance



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

$$x /= \sigma^2 \quad \text{for all samples}$$



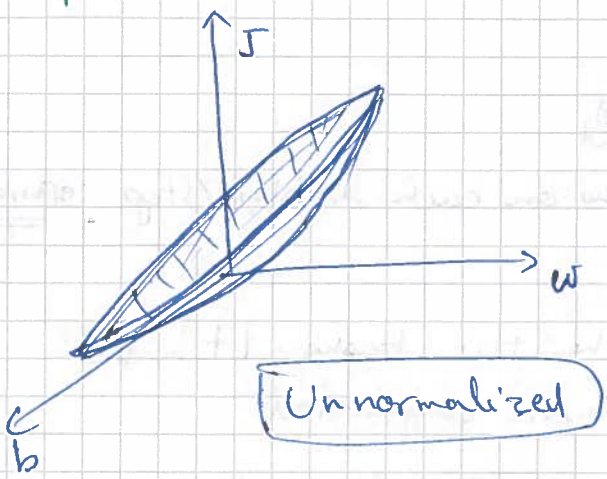
features have very different range

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

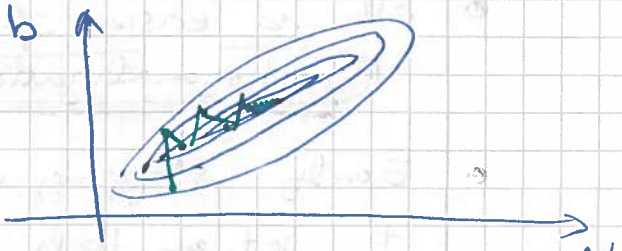
x_1 : 1 1000
x_2 : 0 1

$\Rightarrow$ w_1 & w_2 will have very different values

$\Rightarrow$ the cost function will be very elongated bowls



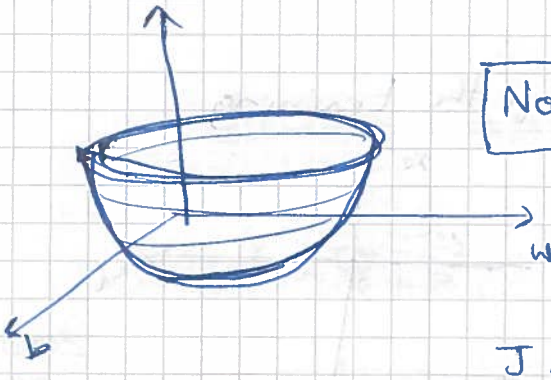
Unnormalized



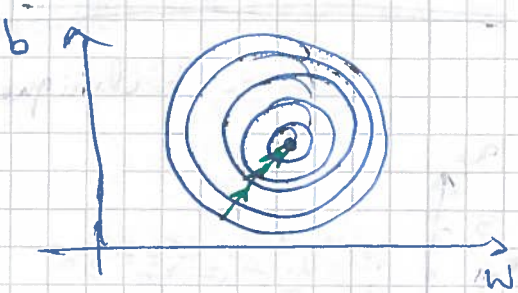
$\Rightarrow$ GD takes much longer to find the minimum
↳ have to use a very small learning rate

Wher as Normalization:

- features become very symetric



Normalized

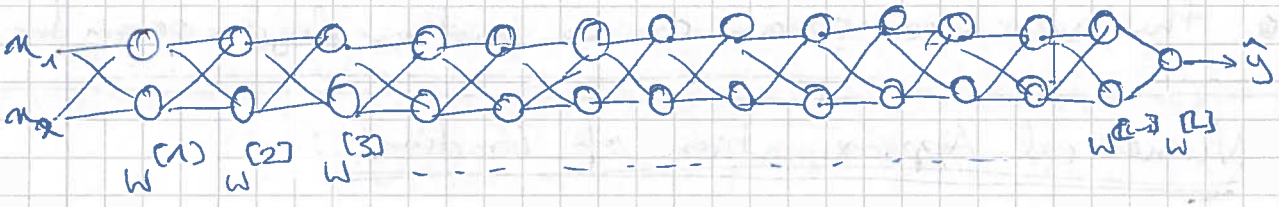


J has a more spherical counturs
$\Rightarrow$ you can have larger learning rate
$\Rightarrow$ faster GD

- If features have roughly the same range then normalization becomes less important.

Vanishing / Exploding Gradients:

- Here on example with weights w, but same argumentation is also valid for derivitives.



for simplicity, assume $g(z) = z$ (linear activation) & $b = 0$

$$\Rightarrow y = W^{[L]} W^{[L-1]} W^{[L-2]} \cdots W^{[3]} W^{[2]} W^{[1]} x$$

- now imagine each $W^{[l]}$ slightly larger than I (identity matrix)

$$W^{[l]} = \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}$$

$$\Rightarrow \hat{y} = W^{[L]} \begin{bmatrix} 1.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{L-1} x \Longrightarrow$$

Weights grow exponentially !!!

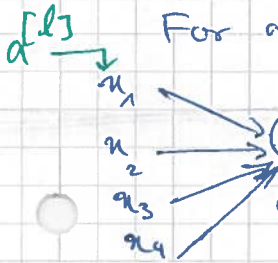
- on the other hand imagine each $W^{[l]} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$ $\Longrightarrow$

Weights decrease exponentially !!!

- Imagine now you had $L = 150$ layer!
- this problem was for a long time a huge barrier to train very deep networks.
- there is a partial solution: careful choice of initial weights

Weights Initialization Deep Networks:

For a single neuron:



$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

We want for large n $\longrightarrow$ smaller w_i because otherwise z explodes (it is a sum)

One good thing is make $\text{Var}(w_i) = \frac{1}{n}$

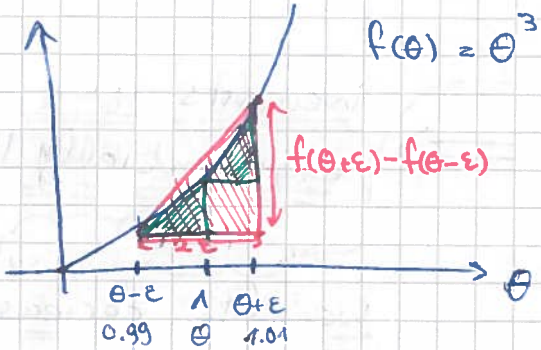
$\Rightarrow W^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$

for ReLU: $\frac{2}{n^{[l-1]}}$ this makes $\text{Var}(w_i) = \frac{2}{n}$

- if using a tanh activation: $\sqrt{\frac{1}{n^{[l-1]}}}$ (Xavier Initialization)
- another one: $\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
- this variance param can be another hyper_param for model

Numerical Approximation of Gradient:

- Gradient Checking can help make sure that the impl of Backprop is correct.
- First let's see how to numerically approximate the gradient.



- the larger triangle gives a much better approximation of derivative of f.

- two sided difference: derivative

$$\frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon} \approx g(\theta) = 3\theta^2 = 3$$

$$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$$

approx. error: 0.0001

Whereas:

- one sided difference

$$\hookrightarrow \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon} = 3.0301$$

approx error: 0.03 !

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta-\varepsilon)}{2\varepsilon}$$

Error: $O(\varepsilon^2)$

$\varepsilon = 0.01 \rightsquigarrow$ Error: 0.0001

Two-sided difference derivation is much more accurate.

where as:

$$f'(\theta) = \lim_{\varepsilon \to 0} \frac{f(\theta+\varepsilon) - f(\theta)}{\varepsilon}$$

Error: $O(\varepsilon)$

Gradient Checking:

- Debug and verify that your backprop is correct. It saves a lot of →

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

(concatenate)

$$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = J(\theta)$$

- Same for $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$: reshape into a big vector $d\theta$

(concatenate)

- Question is now:

Is $d\theta$ the gradient of $J(\theta)$? (slope)

Gradient Checking

Implementation of Grad Check:

$J(\theta) = J(\theta_1, \theta_2, \theta_3, \dots)$

for each i:

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

(rest remains the same)

this should be $\approx \frac{\partial J}{\partial \theta_i}$

compute $d\theta_{approx}$ for all i's → at the end you have two vectors and need to check:

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

Use the Euclidean distance

Check: $\frac{\| d\theta_{approx} - d\theta \|_2}{\| d\theta_{approx} \|_2 + \| d\theta \|_2} \approx$

just normalize by length of both vectors (just for the case the vectors are very different in size)

- 10^{-7} → great! :)
- 10^{-5} → a little concerned, careful :|
- 10^{-3} → worry, double check everything :|
- $> 10^{-3}$ → most probably a bug somewhere :(

Grad Check implementation notes:

- Don't use in training - only for debug.
  - → $d\theta_{approx}[i]$ for all i is very costly
  - → compute the derivatives and $d\theta$ during backprop. only when debugging turn on grad check and compute $d\theta_{approx}[i]$ for checking

- If algorithm fails grad check, look at components to identify the bug.

$db^{[l]}$ → very far off in $d\theta_{approx}$
$dW^{[l]}$ → ok
⇒ Problem in $db^{[l]}$.

- Don't forget regularization in definition of J if turned on

$d\theta$: grad of J w.r.t. θ

- Doesn't work with dropout.
  - → implement without dropout (grad check), then turn on dropout when you are sure every thing is Ok.

- Maybe (sometimes): Run at random init and then again after some training.

Course 2 - Improving Deep NN - Week 1 - Programming 1

Initialization

Goal:
- Understand different regularization methods
- Implement dropout
- Realize the role of regularization: Without regularization the model performs better on training set but worse on test set.
- Try / Understand both dropout and regularization on your model.

- **Initialization:**

A well chosen initialization ~~can speed up~~
- can speed up the convergence of GD
- increase the odds that GD converges to a lower ~~Error~~ training and generalization error.

- zero init.
- random init.
- He init : random values scaled according to a paper by He et al., 2015

- **Zero init:**
  - networks fails to break symetry
    - each neuron in each layer computes the same thing as if an NN with $n^{[l]} = 1$ for all layers
    - no more powerful than a linear classifier.
  - The weights $W^{[l]}$ should be initialized randomly to break symetry
  - It is ok to init $b^{[l]}$ to zeros.

- **Random init with large values:**
  - cost starts very high. This is because large random weights, make the last activation (sigmoid) output very near to 1 or 0; when that example is wrong, the Loss is very high. Indeed when $\log(a^{[3]}) = \log(0)$ the loss goes to infinity.

"inf" because "numerical round off"

  - Poor init leads to vanishing / exploding grads, which slows down the optimization.
  - Initializing weights to very large random values does not work.
  - But how small should the values be?

- **He init:** similar to Xavier init $\left(\sqrt{\frac{1}{layer\_dims[l-1]}}\right)$

(He et al., 2015) → $\sqrt{\frac{2}{layers\_dims[l-1]}}$

→ dimension of previous layer.

  - Recomended for ReLU layers
  - very good accracy.

model	train accuracy	Problem / comment
3-layer NN with zero init	50%	fails to break symetry
3-layer NN with large random init	83 %	too large weights
3-layer NN with He init	99 %	recommended method →

- Different init leads to different results
- Random init is used to break symetry and make sure different hidden units can learn different things
- Don't initilize to values that are too large
- He initialization works well for networks with ReLU activation.

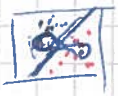
Course 2 - Improving Deep NN - Week 1 - Programming 2

Regularization

- Deep learning models have so much flexibility and capacity that overfitting can become a serious problem if training data is not big enough.
- It will perform well on the training data set, but the learned Network does not generalize well on new examples.

- Baseline model, Trainset accuracy: 94.8%
  Test set " : 91.5%

- The baseline non-regularized mode overfitts the training set.
  It is fitting the noisy points



- L2 Regularization to avoid overfitting & Modify the normal cost function

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

to

$$J_{regul.} = \underbrace{-\frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)\right)}_{\text{cross-entropy cost}} + \underbrace{\frac{1}{m}\frac{\lambda}{2}\sum_{l}\sum_{k}\sum_{j}W_{k,j}^{[l]2}}_{\text{L2 Regularization cost}}$$

- We must now change the backprop as well:
  All the gradients must be computed w.r.t. new cost function

- To each dW_1, dW_2, dW_3 we have to add the gradient of the regularization term.

$$\left(\frac{d}{dW}\left(\frac{1}{2}\frac{\lambda}{m}W^2\right)\right) = \frac{\lambda}{m}W$$

← • After L2 regularization ($\lambda = 0.7$): Train set accuracy: 93.83 % / Test set accuracy: 93 % } !

• The model is not fitting for noise anymore 

Notes

- the value of λ is a hyperparameter that we can tune using a dev set
- L2 Regularization makes the decision boundary smoother. If λ is too large, it is also possible to "oversmooth" resulting in a model with high bias

Smoothing the DB

- L2 Regularization relies on the assumption that models with smaller weights is simpler than a model with larger weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. It becomes too costly for the model to have large weights. This leads to a smoother model, where the output changes more slowly as the input changes.

• The implications of the L2 Regularization on:

- The cost function:
  A Regularization term is added to the cost.
- The backprop function:
  There are extra terms in gradients w.r.t. weight matrices.
- Weights end up smaller ("weight decay")
  Weights are pushed to smaller values.

Dropout

- randomly shut down some neurons in each iteration, with the probability (1 - keep-prob)
  - The dropped out neurons don't contribute to training in both the forward and backward propagation of iteration.
  - We actually modify the model at each iteration and train a different model that uses only a subset of neurons. This way, the neurons become less sensitive to activation of one other specific neuron, because that other neuron might be shut-down at any time.

→

← Impl: $D^{[l]} = [d^{[l](1)}\ d^{[l](2)} \dots d^{[l](m)}]$

$d^{[l]}$ same shape as $a^{[l]}$ → mask vector

$d^{[l]} = np.random.rand(n^{[l]}, 1)$

$d^{[l]} := (d^{[l]} < (1 - keep\text{-}prob))$

Forward:

- Mask matrix with the same shape as $A^{[l]}$
- For each iteration, and for each training sample, we shut down a different set of neurons of layer l.
- $A^{[l]} := A^{[l]} * D^{[l]}$ (Shutting down some neurons)
- $A^{[l]} := A^{[l]} / keep\text{-}prob$ (Assure that the result of the cost will have still the same expected value as without Dropout. ("inverted dropout")

Backward:

- Back prop:
  - In forward pass we shut down some neurons using $D^{[l]}$. In backward pass, we should shut down exactly the same neurons by reapplying $D^{[l]}$ to $dA^{[l]}$
  - Since $A^{[l]}$ was scaled by keep_prop in forward pass, we have to scale $dA^{[l]}$ by the same value in the backward pass

- With dropout: (keep-prob = 0.86) Trainset accuracy: 93% / Test set accuracy: 95 % } ! Dropout works great :)
- Dropout only in Training ! Not in testing !

model	train accuracy	test accuracy
3-layer NN without regularization	%95	91.5 %
3-layer NN with L2 regularization	%94	%93
3-layer NN with Dropout	%93	%95

Course 2 – Improving deep NNs – Week 1, Programming 3

Gradient Checking

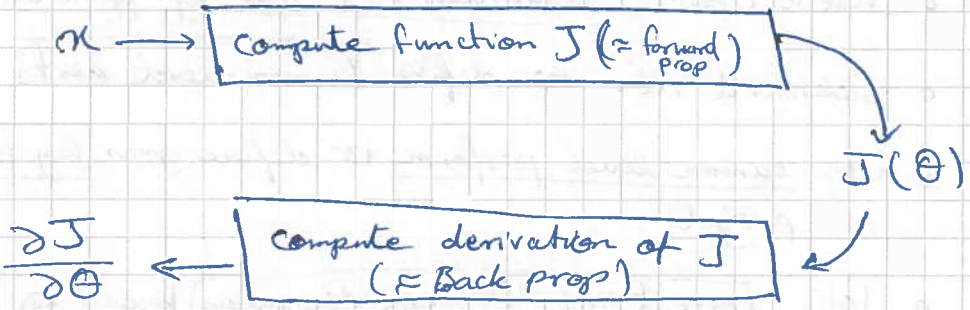
- implement Gradient checking from scratch.
- understand how to use differences formula to check the backprop impl
- Back prop and the differences formula should give the same results.
- Find out which params should be Gradient checked

- Back prop computes gradients $\frac{\partial J}{\partial \theta}$, where θ denotes the parameters of the model. J is computed using forward prop and the loss function

- We use the Forward prop to verify the implementation of back prop. Because forward prop can be implemented easier and we assume it does not have a bug.

$$\frac{\partial J}{\partial \theta} = \lim_{\varepsilon \to 0} \frac{J(\theta+\varepsilon) - J(\theta-\varepsilon)}{2\varepsilon} \approx \frac{J(\theta^+) - J(\theta^-)}{2\varepsilon} \approx \frac{J^+ - J^-}{2\varepsilon}$$

1- Dimensional Grad check :

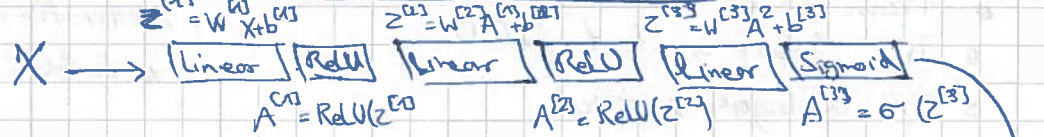


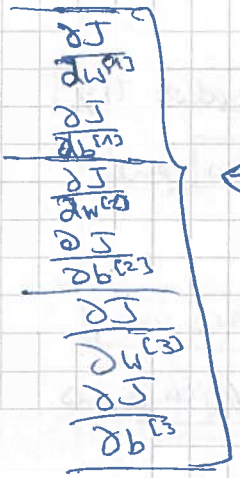
1. compute gradapprox $= \frac{J^+ - J^-}{2\varepsilon}$
2. compute grad using back prop.
3. compute the relative difference between gradapprox and grad:

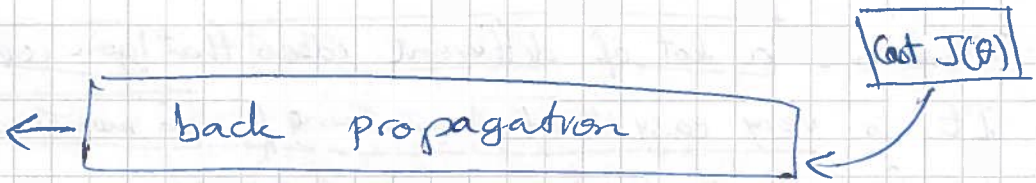
$$\text{difference} = \frac{\| \text{grad} - \text{gradapprox} \|_2}{\| \text{grad} \|_2 + \| \text{gradapprox} \|_2}$$

4. if difference is small, say $< 10^{-7}$ then you are ok.

N- Dimensional grad checking :







The implementation uses a function to flatten the whole parameters dictionary (W1, b1, W2, b2, ...) into one giant vector; and the whole gradients dictionary into a giant vector.

dictionary_to_vector()
gradients_to_vector()

And

vector_to_dictionary

Course 3 - C3 Structuring ML Projects

Week 1 — ML Strategy (1)

Learning Objectives:

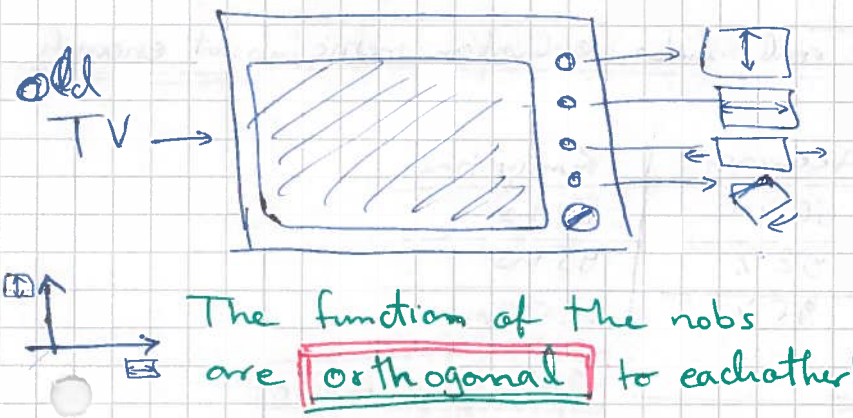
- Understanding Why a ML strategy is important.
- Apply satisfying and optimizing metrics to set up your goal for ML project
- Choose correct train/dev/test split of your dataset
- Understand How to define human-level performance
- Use human level perform to define your key priorities in ML projet
- Take correct ML Strategic decisions based on observations of performances and dataset.

Introduction : Motivation

- Imagine you cat classifier achieves 90% accuracy, but this is not enough.
- What to do? you could:
  - collect more data
  - collect more diverse training set
  - train longer with GD
  - Try Adam instead of GD
  - Try a bigger Network
  - Try smaller Network
  - Try dropout
  - Try L2 Regularization
  - Change Network Architecture
    - Activation function
    - # hidden units
    - ...
- There are a lot of different ideas that you could try.
- It is very easy to try something for 6 monthes only to figure out this does not help. :(
- ML Strategy gives you a quick and effective way to figure out which idea is worth persuing and which ones you can safely discard.

Orthogonalization:

Challenge:
- In ML System you could try many different things, such as hyperparameters.
- Having a very clear view on "what to change" in order to achieve an effect.



- Each nob does a distinct function
- If you had one single nob that did: $(0.1 \times \updownarrow + 0.5 \leftrightarrow + 1.7 \times \square + 0.2 \circlearrowleft)$ It would be impossible or very hard to tune the tv.

The functions of the nobs are **orthogonal** to eachother!

Chain of Assumptions in ML:

	nobs to tune
Fit training set well on cost function. (≈ human-level performance)	• bigger network • Adam ⋮
↓ Fit dev set well on cost function.	• Regularization • Bigger training set ⋮
↓ Fit test set well on cost function.	• Bigger dev set ⋮
↓ Perform well in real world. (Happy users)	• change either dev set or the cost function ⋮

- Each nob/function addresses a distinct/seperate problem
- **Early Stopping** could be a **bad nob**. Because it **simultaneously effects two things**:
  - How good you fit training set (you stop before you reach the best train set fitting)
  - Improve dev set performance.

Setting up your goals: Single number evaluation metric

- having **1 single real number evaluation metric** speeds up iteration an helps choosing algorithms easier.

$$\text{Precision} = \frac{TP}{TP+FP} \%$$ $$\text{Recall} = \frac{TP}{TP+FN} \%$$

$$\text{F1-score} = \frac{2}{\frac{1}{P}+\frac{1}{R}}$$: Average of precision and recall (harmonic mean)

Satisficing and Optimizing metrics:

Single Real number evaluation metric

- Some times just one single real number evaluation metric is not enough

Example:

Classifier	Accuracy	Running time
A	90 %	80 ms
B	92 %	95 ms
C	95 %	1,500 ms

{ maximize Accuracy ← Optimizing metric
subject to running time ≤ 100 ms ← Satisficing metric
↳ (Just has to be good enough)

Generally: If you have N metrics: 1 Optimizing, N-1 Satisficing

Another example: Wake Words / Trigger words

↳ Intrested in accuracy & # false positive.

⇒ { maximize accuracy ← Optimizing
s.t. ≤1 FP in every 24 hours. ← Satisficing
(at most)

Train/dev/Test distributions:

- The way we set up train/dev/test set can have a huge impact on the progress. It can also slowdown progress dramatically.

- make sure your dev & test sets come from "the same" distribution
↳ you can for example randomly shuffle the data into dev/test sets

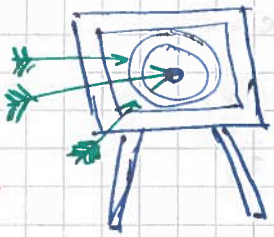
True Story: Wasted 3 month of work:

- Optimized on dev set on loan approval for medium income zip codes
- tested on low income zip codes.

- Choose the dev set and test set
  - from the same distribution
  - and reflect the data you expect to get in the future and consider important to do well on.

---

training set helps you hit the target well.



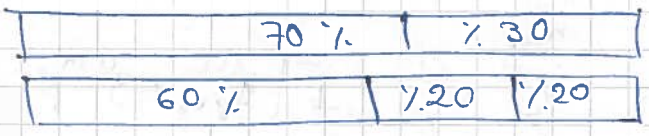
{dev set / test set} + evaluation metric

Defines the target !

Size of the dev and test sets :

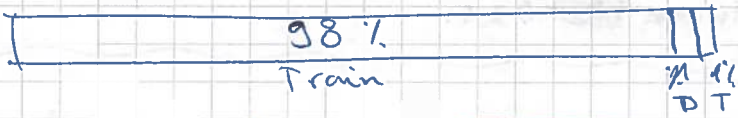
- guidelines of how to setup your dev and test sets are changing in the DL era.
- old way of splitting data

old guide lines



Small DS sizes
1000
10000 samples

- In DL era we have much larger DSs



large DS
1,000,000 samples

- Size of your test set : evaluation
  - test set gives you ~~confidence~~ on how good your final system is after you have developed it.
  - Big enough to give you slight confidence in the overall performance of your system.
  - Some times you don't even need a high confidence in the overall performance of the final system
    ⇒ all you need is train/dev set

  - Some time test/dev is used exchangeably.
  - Not recommended not having a test set

When to change dev/test sets and metrics?

Cat classifier example:

→ Metric classification error

Algorithm A: 3% error → * But lets pornographic images wrongly classified as cat!

Algorithm B: 5% error

⇒ { metric + Dev set : prefers A
  { you / users : prefer B

⇒ We must change the error metric, because it does not reflect our preference.

for example: If error metric was (indicator func.)

$$\text{Error} = \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} I\left\{ y_{pred}^{(i)} \neq y^{(i)} \right\}$$

(Just simply count the number of misclassifications)

we change it to

$$\text{Error} = \frac{1}{\sum_i w^{(i)}} \sum_{i}^{m_{dev}} w^{(i)} \, I\left\{ y_{pred}^{(i)} \neq y^{(i)} \right\} \quad \text{with} \quad w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ non-porn} \\ 10 & \text{if } x^{(i)} \text{ porn} \end{cases}$$

normalize so that the error remains between 0 & 1.

penalize wrong porn misclassification

Orthogonalization:

1. First define a metric to evaluate the two classifiers
2. Then worry separately about how to do well on this metric

1. Define the target
2. Worry about shooting it well SEPARETLY

- First define the metric
- then figure out how to advive it

Guideline: If doing well on your metric + dev/test set does not correspond to doing well on your application, change your metric and/or dev/test set.

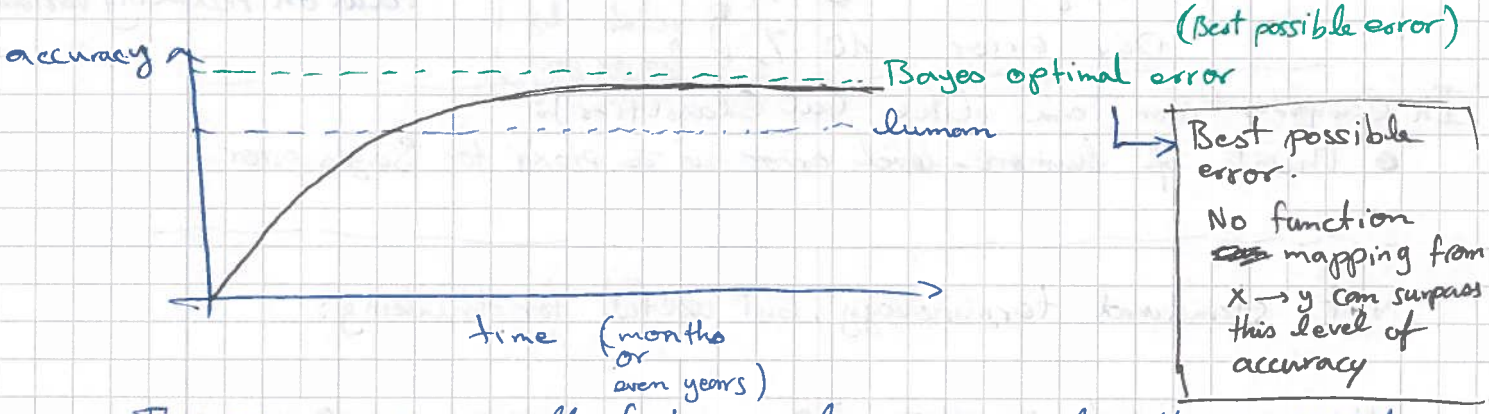
to speed up Progress:

- Set up a dev/test set and evaluation metric quickly and then iterate.
- do not work for too long without an evaluation metric and dev set.

Comparing to human-level Performance:

Why human level Performance?

- Because of advances in deep learning, ML is working much much better and has even become competitive to human performance in some areas.
- The workflow for setting up ML solution has become very efficient, so it is now natural to think of human-level performance.



- The progress is normally fast, as long as we haven't surpassed human-level performance. But after that the progress becomes very slow. It has two reasons:
  - 1. for many tasks the human-level performance is already very near to Bayes optimal error.
  - 2. After passing human-level performance, certain tools can hardly, or no longer be used to improve performance:
    - Getting labeled data from humans.
    - Gain insights from manual error analysis. (Why did a person get this right?)
    - Better analysis of bais/variance.

Avoidable Bias:

- We always want our training algorithm to do well on the training set. But sometimes we don't want it to do too well.
- Knowing the human-level performance can give us a good limit on how well we want our algorithm to perform on the training data. → But not too well.

Cat Classification Example:

human error	1%	big gap	⇒ Focus on reducing bias.
Training error	8%	small gap	⇒ Probably slight bias.
Dev error	10%		

Human error	7.5%	relatively small gap	⇒ Probably high variance Focus on reducing variance.
Training error	8%	relatively larger gap	
Dev error	10%		

In Computer Vision Tasks (like Cat Classifier):
- Think of human-level error as a proxy for Bayes error.

Not standard terminology, but useful for thinking:

Human error (≈ Bayes Error)	Avoidable Bias:	Difference between (≈Bayes error) and Training error.
Training error		
Dev error	Variance	

- By using the term "Avoidable Bias" we actually indicate that there is a level of error that we can't or basically don't want to surpass.

So, in above example we can say "Avoidable Bias is 0.5%" !

Understanding Human-level performance:

- Human-level error as a proxy for Bayes error.

Example of a medical image classification Task → 0.5%

a) untrained human 3% error b) typical doctor 1% error

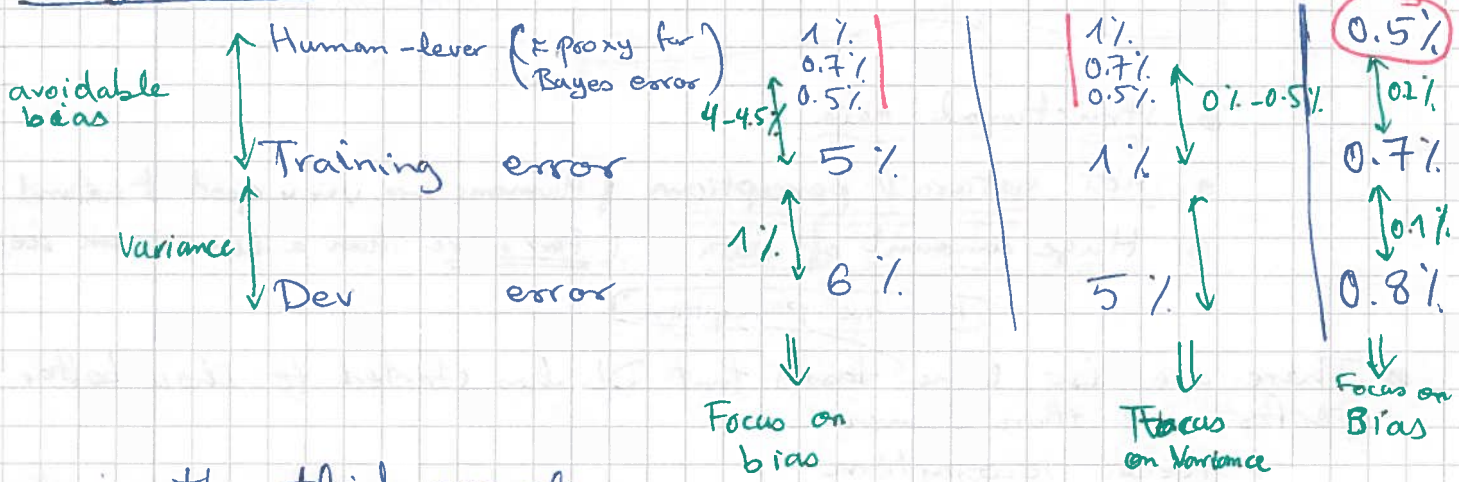
c) experienced doctor 0.7% error d) consensus opinion of a team of experien. doctors

← • Now with these values, what is the value for "human-level error"?

From d) ⟹ Bayes error ≤ 0.5 % error

⇒ We define human-level error (as a proxy for Bayes error) to be 0.5 %

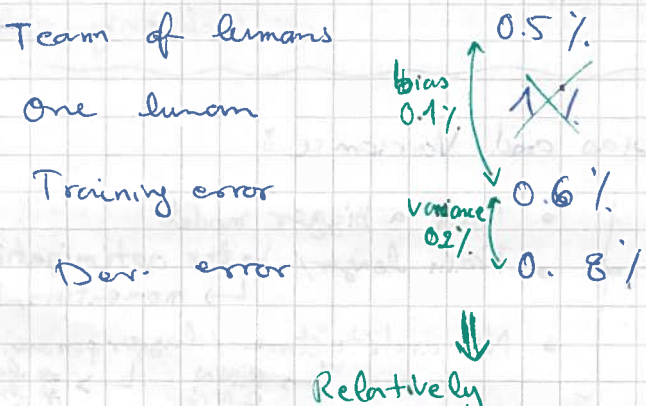
Why this matters? ⟶ Error Analysis Example.

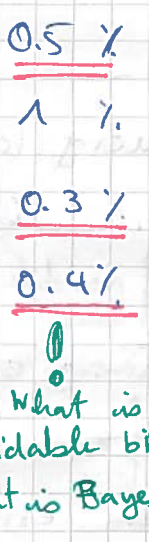


- in the third example it really matters to set the human-level error to 0.5 %. Because otherwise, (for example if you have set Human-level error to 0.7%) you would have missed the fact that you need to work on bias and would have worked (wasted time) on variance instead.

- Interestingly, as we approach to human-level performance (third example) it becomes harder to figure out what to do to improve performance.

Surpassing human-level performance:





- In second example we cannot know what to do. What to focus on? reducing bias? or reducing the variance?

- Moreover, as we have surpassed the human-level performance [illegible] human intuition [illegible] how to improve.

- Problems where ML significantly surpasses human-level performance:
  - online advertising
  - product recommendations
  - Logistics (predicting transit times)
  - loan approval

- Structural data
- not natural perception (Humans are very good at natural perception tasks)
- Huge amount of data (Far more than a human can see/process)

"natural perception"

- There are also some tasks that DL has started to show better performance than humans:
  - speech recognition
  - some image recognition tasks
  - some medical tasks, such as ECG, ...

Improving your model performance:

The two fundamental assumptions of supervised learning:

1. You can fit the training set pretty well
   ≈ Achieving low avoidable bias.
2. The training set performance generalizes pretty well to the dev/test sets.
   ≈ Achieving good low variance.

Reducing (avoidable) bias and Variance:

Human-level
↕ avoidable bias →
- Train a bigger model
- Train longer/better optimization algorithm
  - ↳ momentum, RMSProp, Adam, ...
- NN architecture / hyperparameter search
  - ↳ RNN, CNN
  - ↳ # layers, # hidden units, ...

Training error
↕ Variance →
- More data
- Regularization
  - ↳ L_2, Dropout, data augmentation
- NN architecture / hyperparameter search.

Dev error

From Quiz:

It is **not a problem** to have **different training and dev distribution**.
On the other hand, it would be **very problematic** to have **different dev and test set distribution**.

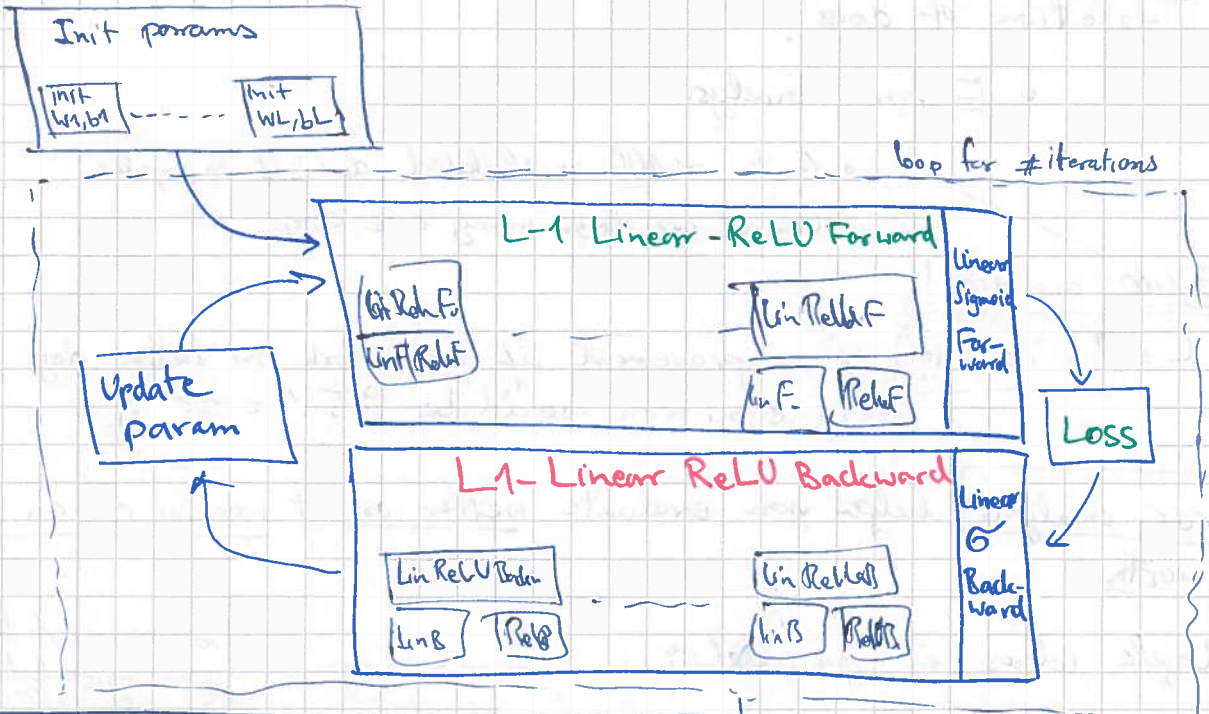
- training distribution ≠ dev distribution O.K.
- dev distribution ≠ test distribution Not O.K.

Course 1 - NNs and Deep Learning - Week 4 - Programming

Building your deep Neural Network step-by-step:

- develop an intuition for the **overall structure of a neural Network**
- write **functions** (e.g. forward prop, backward prop, logistic loss, etc...) that help you decompose your code and ease the process of building a neural Network.
- Initialize/update **parameters** according to your desired **structure**

First a 2-layer



For layer l the linear part is $Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
(followed by an activation)

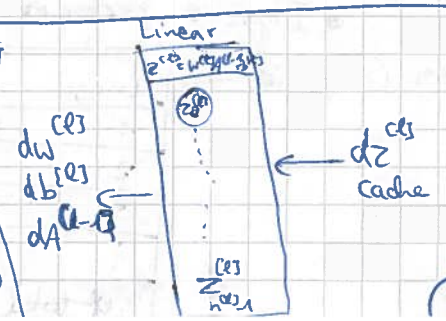
Suppose you have already calculated $dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}}$
You want to get $(dW^{[l]}, db^{[l]}, dA^{[l-1]})$

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]}$$

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]})$$



Course 3 - Structuring ML Projects - Week 2

Week 2 — ML Strategy 2:

Learning Objectives:
- Recognize bias, variance, and data-mismatch by looking at performance of your Algorithm on train/dev/test sets
- Understand what multi-task learning and transfer learning are

Error Analysis: Carrying out Error Analysis

- If you want your ML to do a task that humans can do, but your performance is still not as good as you wished,

  Manually examining the errors in your classification.

  can help giving you intuition on what to do next.

Example:

Cat Classifier: Look at "Dev Examples" to evaluate ideas.
- → Not working good: { accuracy: 90% / error: 10%
- → Some of dog pictures are misclassified as cats. Should you work on your classifier to improve detection of dogs?
  - → "Dev Examples" → mislabled.

- Error Analysis
  - Get ~100 mislabled dev set examples
  - Count up how many are dogs.

5/100 are dogs!
→ "Cieling" of improvement if you work on better dog recognition would be 9.5% error :|

- Error analysis helps you evaluate wether or not working on an idea is worth.
- Multiple ideas in parallel:
  - Fix picture of dogs being recognized as cats
  - Fix great cats (lions, panthers, ...) being misrecognized.
  - Improve performance on blurry images

As you work through your mislabled dev examples, you may even encounter new categories of error, such as "Instagram filters", ...

Error Categories →

Image	Dog	Big Cat	Blurry	Instagram	Comment
1	✓				Pitbull
2			✓		
3		✓	✓		rainy dog at zoo
⋮	⋮	⋮	⋮	⋮	
% of total	8%	43%	61%	12%	