Neural Networks and Deep Learning

Lecture 3 & 4

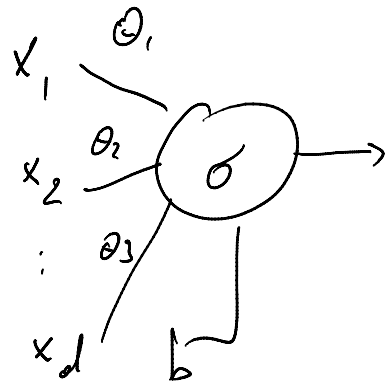
Jan Chorowski
Instytut Informatyki
Wydział Matematyki i Informatyki Uniwersytet Wrocławski

2019-2020

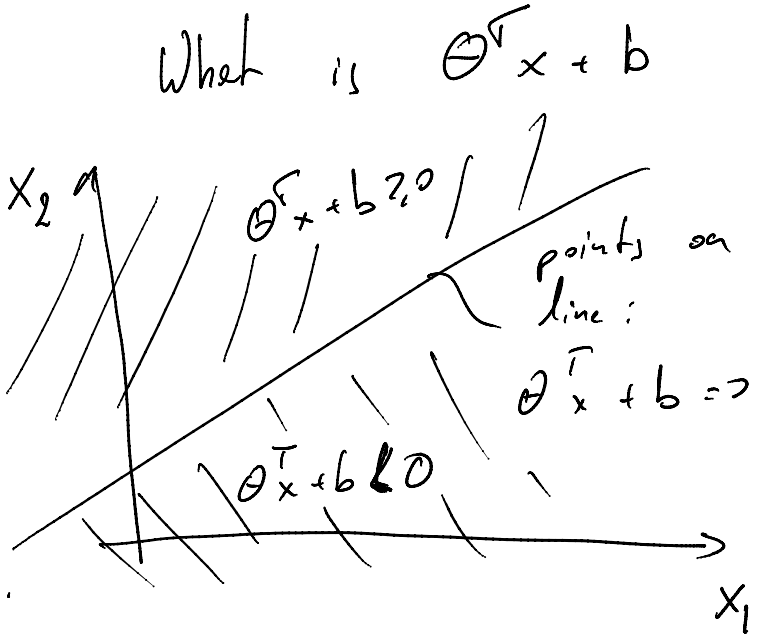
DEPTH MATTERS

Single neuron

$$f(x) = \sigma(\Theta^T x + b)$$

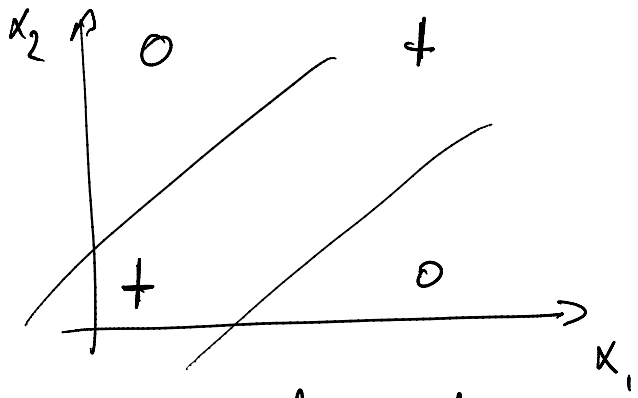


What is $\theta^T x + b$



↳ can split space with 1 hyper-plane.

The XOR Problem



Goal: separate + from o

need 2 lines to do it ?

=> Can't solve with logistic regression / one hidden neuron

The old approach: hand-design data transformation

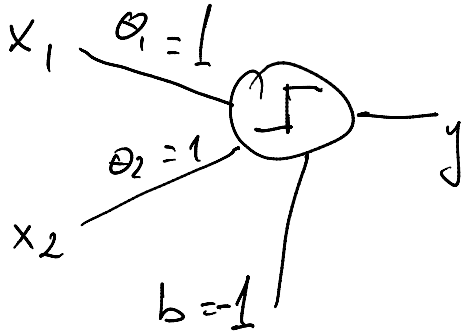
We want x_1 XOR x_2

Intro transformation $\phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) \rightarrow \begin{bmatrix} x_1 \text{ OR } x_2 \\ x_2 \text{ AND } x_2 \end{bmatrix}$

x_1 XOR x_2 $\equiv$ (x_1 OR x_2) AND NOT (x_1 AND x_2)

Logic networks

We need : · OR · AND · NOT



$$y = \begin{cases} 1 & \text{iff } \theta_1 x_1 + \theta_2 x_2 + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

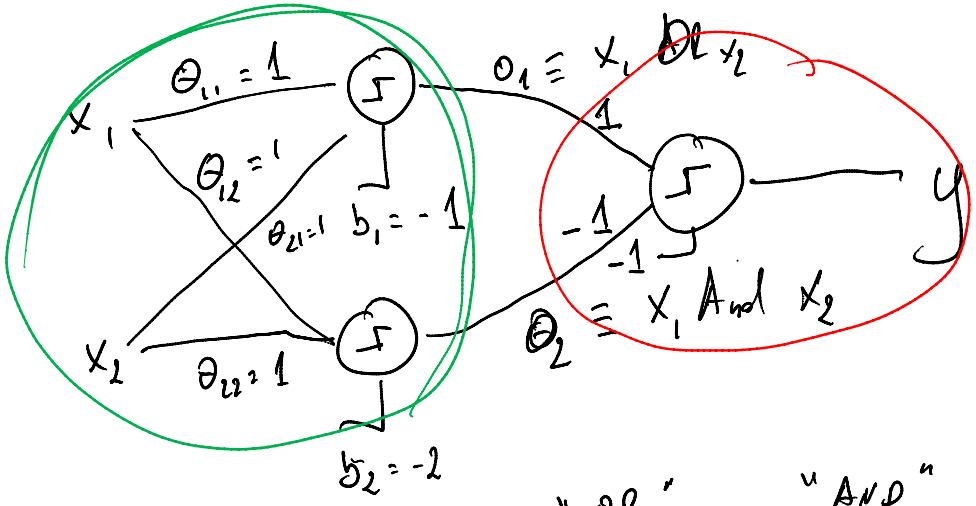
$b = -2$

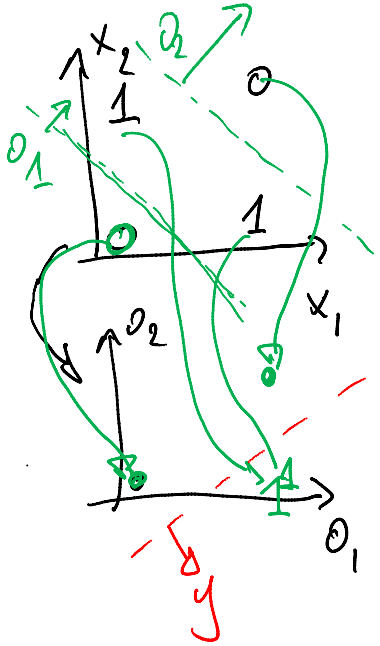
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

x_1 OR x_2

y	
0	x_1 AND x_2
0	
0	
1	

Full XOR network





x_1	x_2	"OR" o_1	"AND" o_2	y	
0	0	0	0	0	x_1 XOR x_2
0	1	1	0	1	
1	0	1	0	1	
1	1	1	1	0	

Hidden layer: a really big deal: Universal Approximation

Thm: ANY BOOLEAN FUNCTION HAS A DNF FORM

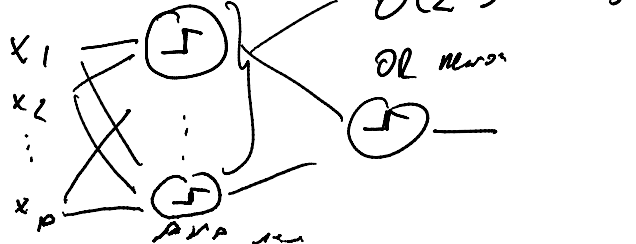
Truth table:

#	x_1	x_2	...	x_D	y
1	0	0	0 ...	0	y^1
2	0	0	0 ...	1	y^2
⋮					
2^D	1	1	1 ...	1	y^{2^D}

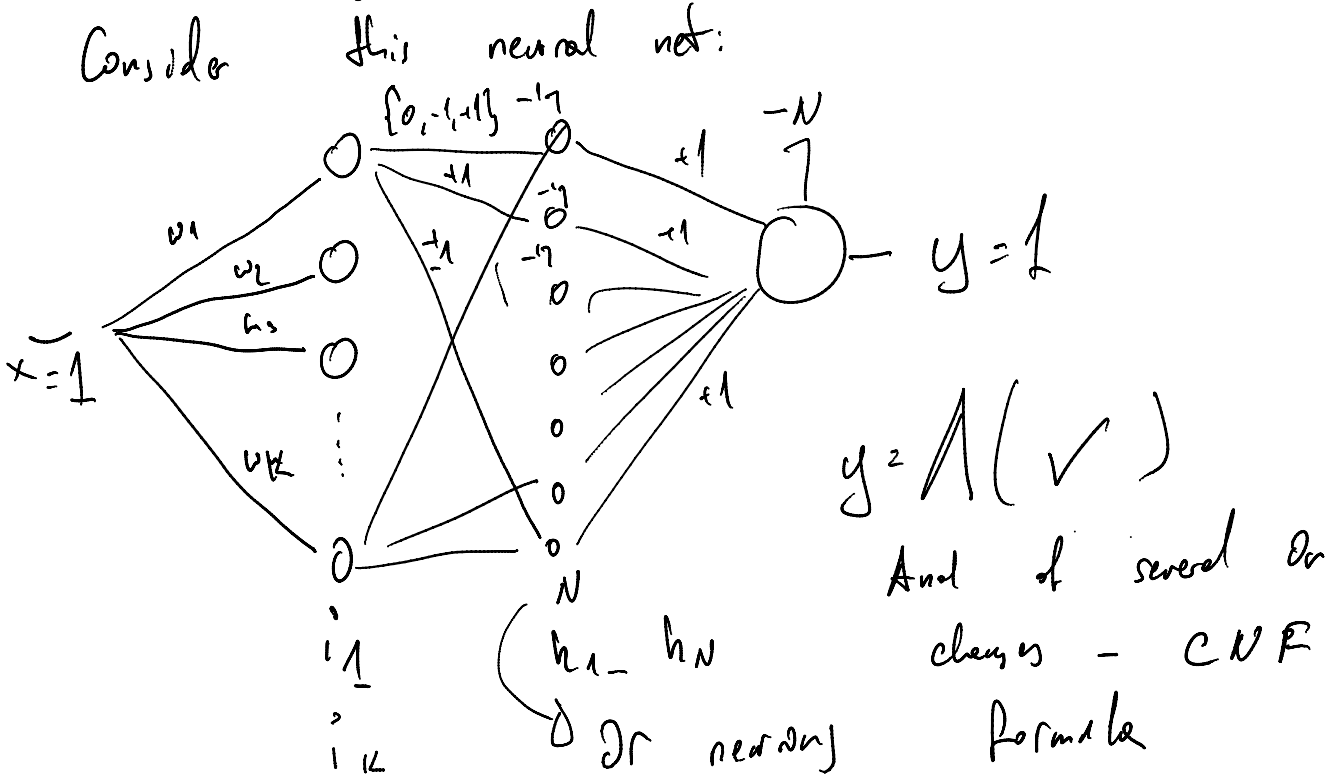
$$y = \bigvee_{i : y^{(i)} = 1} \left(x_1 = x_1^i \wedge x_2 = x_2^i \wedge \dots \right)$$

make a net

the value of x_1 in row i



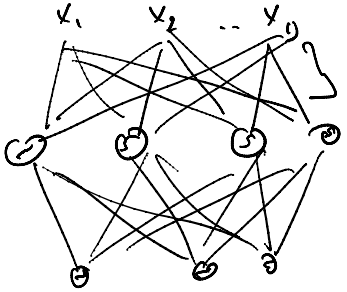
Hidden layer: a really big deal: Optimization Hardness



Opt loss = 0 ≡ Solved SAT

ERROR BACKPROPAGATION

Backprop intuitions



W^1

x

$A^1 = W^1 x$

$O^1 = \sigma(A^1)$

$A^2 = W^2 O^1$

$O^2 = \sigma(A^2)$

$\text{Loss} = L(O^2)$

forward pass

$\frac{\partial L}{\partial A^1} = \frac{\partial O^1}{\partial A^1} \frac{\partial L}{\partial O^1}$

$\frac{\partial L}{\partial O^1} = \frac{\partial A^2}{\partial O^1} \frac{\partial L}{\partial A^2} = \frac{\partial A^2}{\partial O^1} \frac{\partial O^2}{\partial A^2} \frac{\partial L}{\partial O^2}$

$\frac{\partial L}{\partial A^2} = \frac{\partial O^2}{\partial A^2} \frac{\partial L}{\partial O^2}$

Jacobian

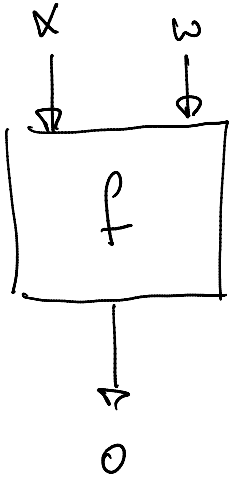
$\frac{\partial L}{\partial O^2}$

1

backward pass

$\frac{\partial O_i^2}{\partial A_j^2}$

Modular implementation



forward: $o = f(x, w)$

backward: $\frac{\partial L}{\partial x} = \frac{\partial o}{\partial x} \frac{\partial L}{\partial o}$

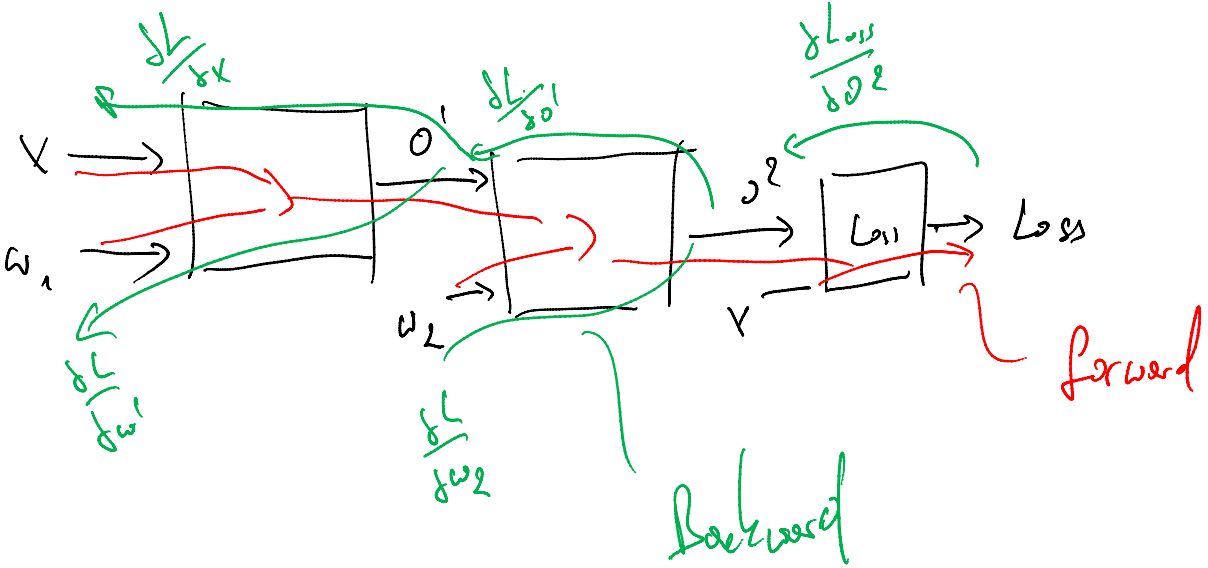
$\frac{\partial L}{\partial w} = \frac{\partial o}{\partial w} \frac{\partial L}{\partial o}$

e.g. a lin layer

$o = w x$

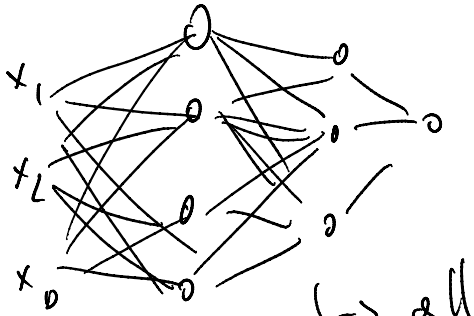
$\frac{\partial L}{\partial x} = W^T \frac{\partial L}{\partial o}$ $\quad \frac{\partial L}{\partial w} = \frac{\partial L}{\partial o} \cdot x$

Full Net Implementation



TRAINING INTUITIONS

Function fitting



$y = f_{net}(x; \theta)$

↳ all params : θ

We want: $\min_{\theta} \mathbb{E}_{x,y}[Loss(f_{net}(x;\theta), y)]$

Approx $\mathbb{E}$ on train set $\{(x^i, y^i) : i = 1 \dots N\}$

$\min_{\theta} L(\theta) = \frac{1}{N}\sum_i Loss(f_{net}(x^i;\theta), y^i)$

e.g. $L(\theta) = \frac{1}{N}\sum_i \frac{1}{2}(f_{net}(x^i;\theta) - y^i)^2$

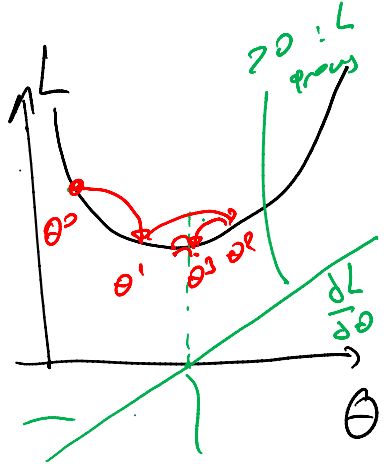
Gradient Descent

$$\min_{\theta} L(\theta) = \sum_i^n (f_{net}(x^i, \theta) - y^i)^2$$

$\theta \leftarrow \theta^0$

While not converged

$$\theta \leftarrow \theta - \alpha \frac{\partial L(\theta)}{\partial \theta} \quad *$$



at opt $\frac{\partial L}{\partial \theta} = 0$

↳ Grad Desc. stops by itself

fine print:

t : each GD step

one full pass over data

Perceptron

$y = \mathbb{1}(\theta^T x)$

$\theta \leftarrow \theta^0$

while not converged

$x^i, y^i \leftarrow$ sample (x, y)

$o^i \leftarrow \mathbb{1}(\theta^T x^i)$

if $o^i \neq y^i$

$$\theta \leftarrow \theta - \alpha (o^i - y^i) x^i$$

step size — error in its step — input

case 1: $y^i = 1$

$o^i \neq y^i \Rightarrow o^i = 0 \Rightarrow \theta^T x^i \leq 0$ too small

After update:

$$(\theta - \alpha \underbrace{(o^i - y^i)}_{0 - 1} x^i)^T x^i =$$

$$= \theta^T x^i + \alpha \underbrace{x^{iT} x^i}_{\geq 0} \geq \theta^T x^i$$

Stochastic Gradient Descent

We want: $\min_{\theta} \mathbb{E}_{x,y}\left[\text{Loss}(f_{\text{pred}}(x,\theta), y)\right]$

Grad Descent: approx $\mathbb{E}$ by average Loss on train data

Stochastic Grad Descent:

approx $\mathbb{E}$ by Loss on a single data sample

or on a "mini batch" of samples

Result: noise ↑ speed ↑

SGD for linear regression

$f_{NET}(x, \theta) = \theta^T x$ $\quad$ $Loss(f_{NET}(x, \theta), y) = \frac{1}{2}(\theta^T x - y)^2$

$$\frac{\delta Loss}{\delta \theta} = (\theta^T x - y)\, x$$

Algo: $\theta \leftarrow \theta^0$

while not converged:

$x^i, y^i \leftarrow$ sample single data sample

$o^i = \theta^T x$

$$\theta \leftarrow \theta - \alpha \frac{\delta Loss}{\delta \theta} = \theta - \alpha (o^i - y^i)\, x^i$$

very similar to Perceptron

step size, error, input

SGD: importance of α

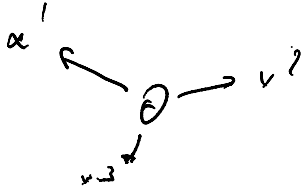
At optimum $\theta = \arg\min_{\theta} \mathbb{E}\left[\text{Loss}\left(f_{net}(x, \theta), y\right)\right]$

and $\mathbb{E}\left[\frac{\partial \text{Loss}}{\partial \theta}\right] = \frac{\partial}{\partial \theta}\mathbb{E}\left[\text{Loss}\right] = 0$

↑ average grad on all samples

- samples "pull" θ in many dir, average is 0

- SGD looks at a single sample => it never converges!



SGD: schedules for α

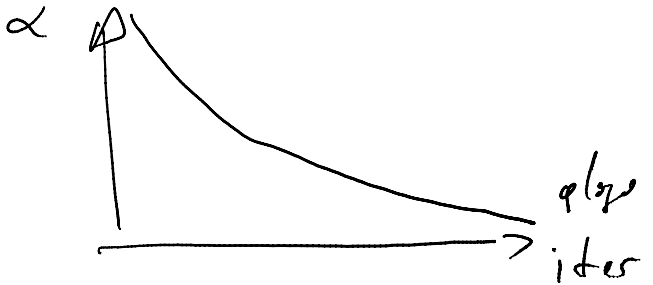
SGD needs a "learning rate schedule"

i.e. while True:

$x^i, y^i \leftarrow$ sample

$\theta \leftarrow \theta - \alpha \frac{\partial Loss}{\partial \theta}$

$\alpha \leftarrow$ a smaller value, e.g. 0.999 α



formal reqs (strongly convex)

$\sum_i \alpha_i = \infty$

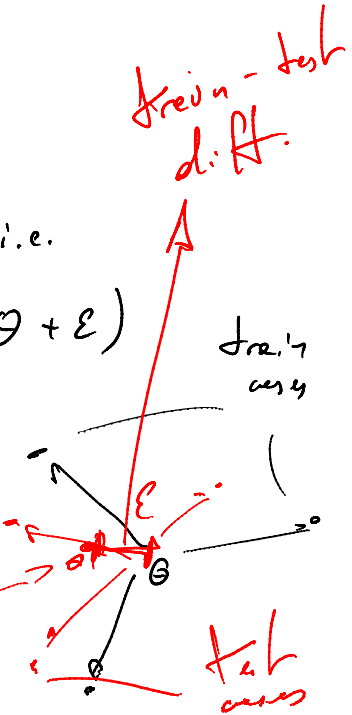
$\sum_i \alpha_i^2 < \infty$ e.g. $\alpha_i = \frac{1}{i}$

SGD: practical schedules for α

1 neural net's loss surface is not convex (good local-pts are similar)

- many local minima (current hyp: all similar)
- many saddle points / flat regions
- sharp vs flat minima
  - ⟶ want a flat minima, i.e.

$$f(x, \theta) \approx f(x, \theta + \varepsilon)$$

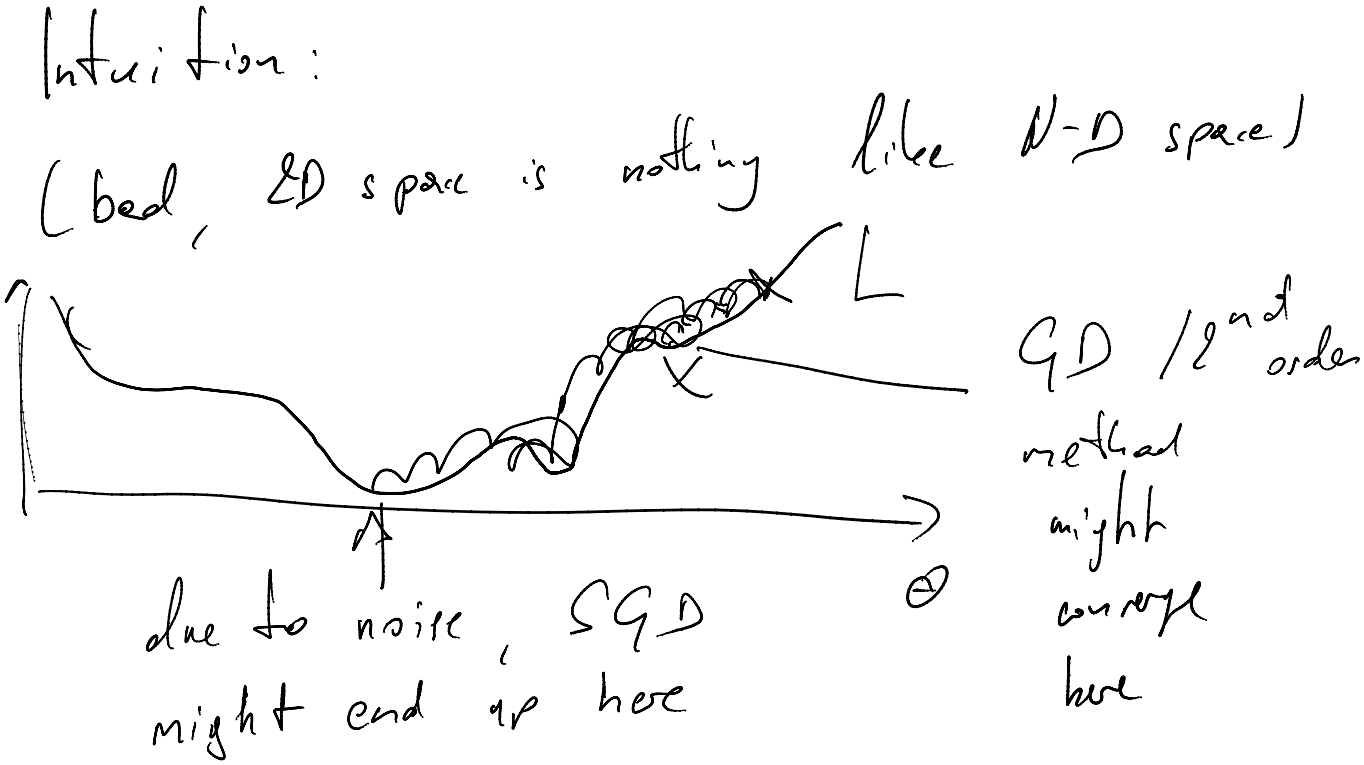


infinitesimal

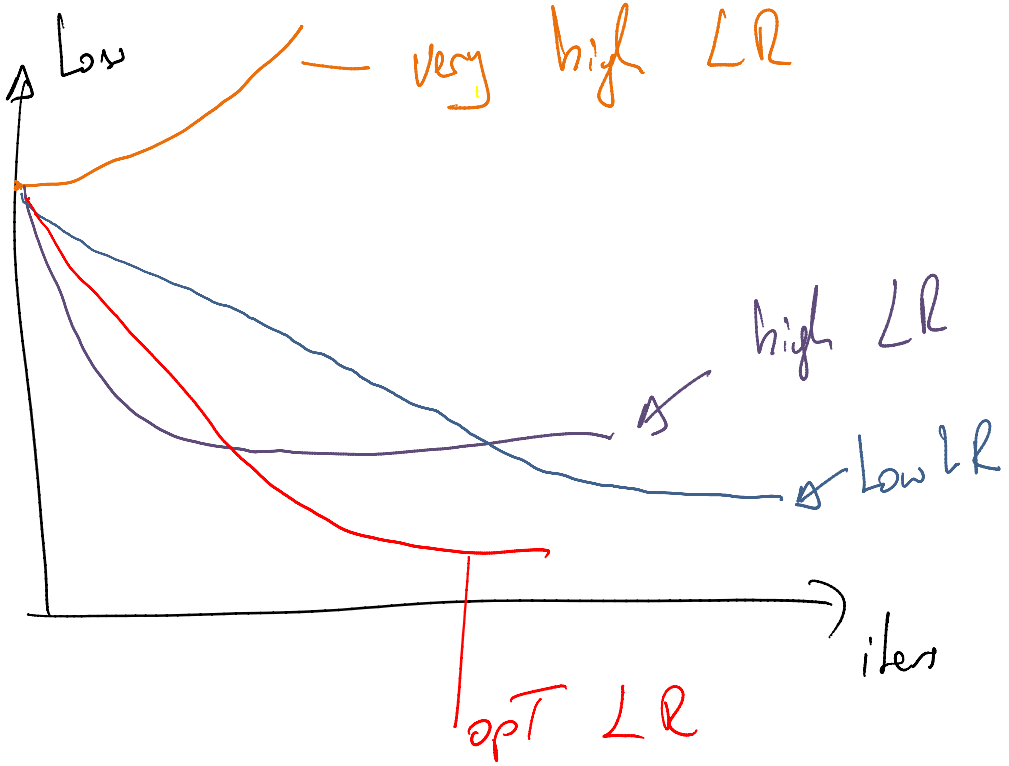
=> SGD noise:

- regularizes
- escapes bad/narrow local opts

SGD intuitions:

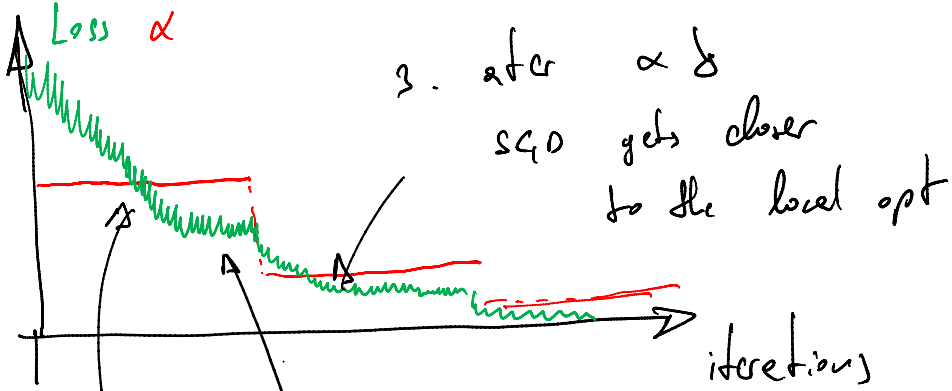


SGD: impact of learning rate



SGD: practical learning rates

- Stair case



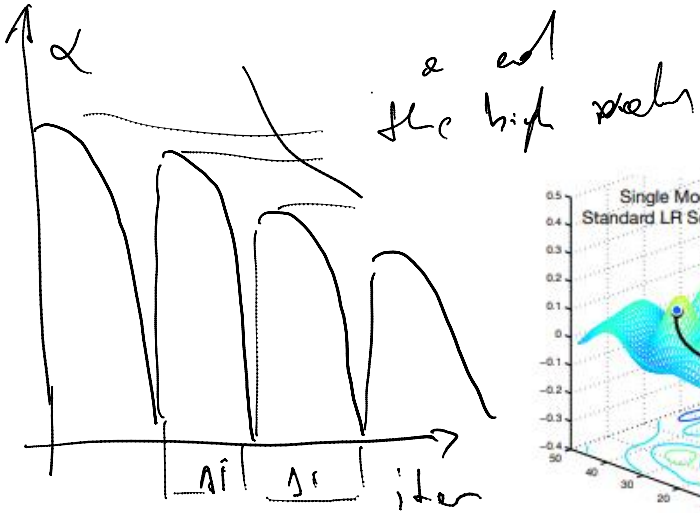
3. after α δ
SGD gets closer
to the local opt

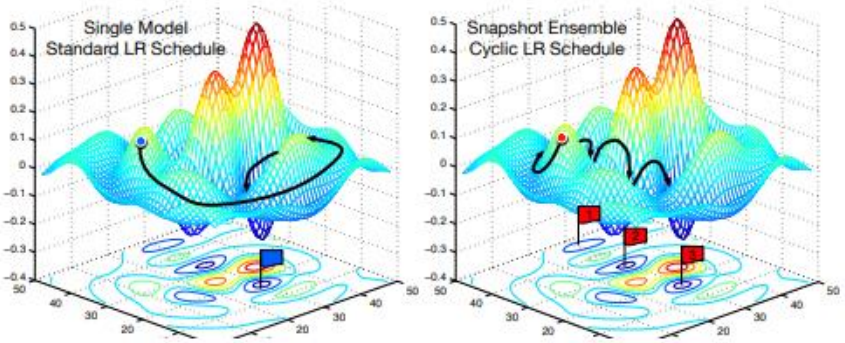
1 Quick progress

2 SGD jumps around, searches for wide local opt

SGD: fancy learning rate schedules

- https://arxiv.org/pdf/1506.01186.pdf
- https://arxiv.org/pdf/1704.00109.pdf



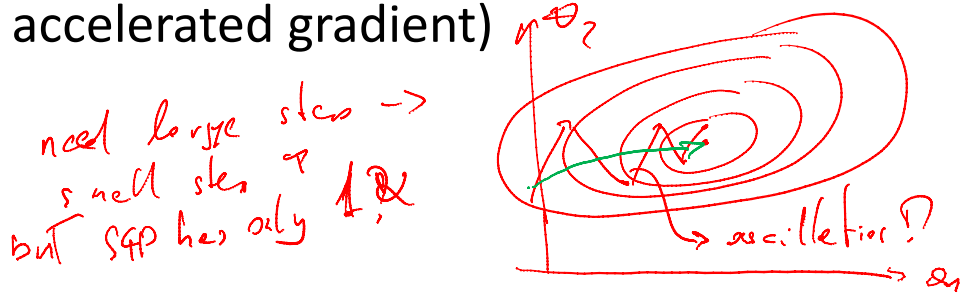


Speeding up SGD

- Momentum: https://distill.pub/2017/momentum/
  - Spread each update over many steps

$$V_t \leftarrow \mu_t V_{t-1} - \alpha_t \frac{\partial J}{\partial \Theta}$$
$$\Theta_t \leftarrow \Theta_{t-1} + V_t$$

  - Theoretical justification in the batch setting (Nesterov accelerated gradient)
  - Popular



Only use sign of the gradient

- Resilient Propagation (RProp, IRProp):
  - Use only the sign of the gradient
  - Rules for learning rate scaling
  - Only works in BGD
- RMSProp (Rprop for SGD)

$$r_t \leftarrow (1-\gamma)\left(\frac{\partial J}{\partial \Theta}\right)^2 + \gamma r_{t-1}$$

$$\Theta_t \leftarrow \Theta_t - \frac{\alpha_t}{\sqrt{r_t + \epsilon}}\frac{\partial J}{\partial \Theta}$$

- Keep a running average of the magnitude to get the sign

Adam – the recommended default

- RMSprop with momentum
- http://arxiv.org/abs/1412.6980

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Model averaging

- Use many models:
  - Train many models, and average (ensemble)
  - Train 1 net, but once in a while use high LR to jump to another local optimum ("snapshot ensemble")
- Stabilize the weights by Polyak averaging
  - Keep a running average of parameters:

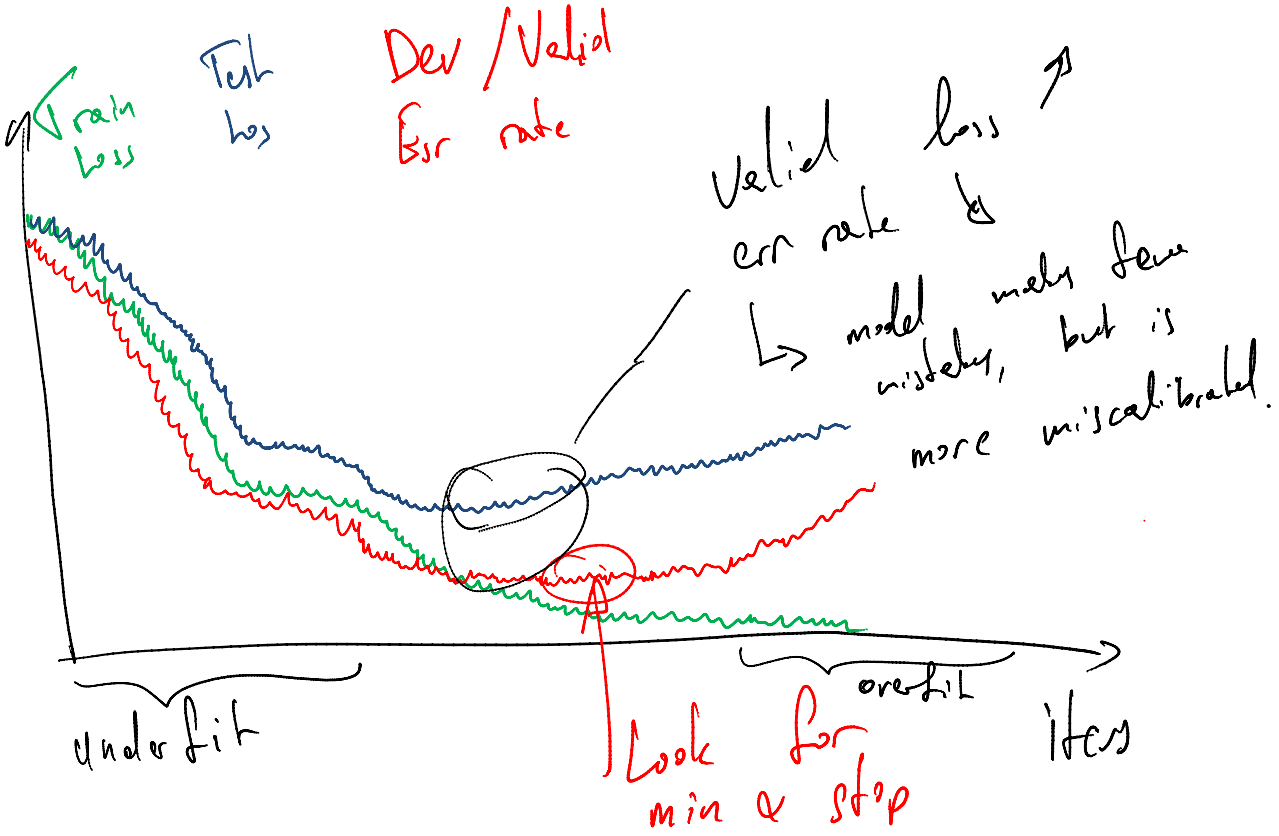
$$\Theta \leftarrow \text{normal train update}$$
$$\Theta^{ave} \leftarrow 0.995\Theta^{ave} + 0.005\Theta$$

Use Θ^{ave} for testing & profit!

What to monitor during training

- Loss
- Magnitude (mean and st. dev) of gradient for (best for each layer)
- Activation values (mean, st. dev.) for each layer (monitors whether units are “stuck”)
- Good practices:
  - Save the model during training.
  - Make the analysis from the saved models -> less clutter in the optimization code.

Early stopping



Have some patience!

With SGD we expect some steps in the wrong direction !?

- Patience algorithm:

  Set patience to some initial value

  While iters < patience

    if significant improvement of some criterion

      patience = max(patience, iter*2)

Can also use patience for learning rate decay!
E.g.:
https://pytorch.org/docs/stable/_modules/torch/optim/lr_scheduler.html#ReduceLROnPlateau

Neural Networks: scale matters

Consider a lin. layer: $O_1 = w_1 x$

Consider another one: $O_2 = \frac{1}{10} w_2 x$ with $w_2 = 10 w_1$

Q: How do they differ?

Forward → same operations: $O_2 = \frac{1}{10}(10 w_1) x = w_1 x$

Backward:

$$\frac{\partial \text{Loss}}{\partial w_1} = \frac{\partial L}{\partial o_1} x$$

$$\frac{\partial \text{Loss}}{\partial w_2} = \frac{\partial L}{\partial o_2} x \cdot \frac{1}{10} \quad !$$

$\Rightarrow$ grad step magnitude diverges

w_2: 10x larger, but 10x smaller grad ? 100x ??

Choose initial weights wisely

- Goal: the outputs of the neuron have 0 mean, st. dev 1.
- Assume inputs are normalized.

$$O = \sum_{i=1}^{n} w_i x_i$$

x_i: mean 0, stdev 1

w_i: mean 0, stdev $= \sigma$

$$\text{stdev}[O] = \sqrt{n \cdot \text{stdev}^2(w_i x_i)} = \sqrt{n}\,\sigma \Rightarrow \sigma \propto \frac{1}{\sqrt{n}}$$

n "fan-in" of a neuron

n index Random vars, each with variance σ^2

Then compensate for nonlinearity, e.g. ReLU: half neurons $= 0 \Rightarrow$ sum goes over $\frac{n}{2}$ neurons, $\sigma \propto \frac{\sqrt{2}}{\sqrt{n}}$

Choose initial weights wisely

- Goal: the outputs of the neuron has 0 mean, st. dev 1.
- Normalize inputs!
- Typical rules of thumb (for $\tanh()$ activ. function):
  - $b_{init} = \bar{0}$
  - $W_{k_{init}} \sim Uniform\left[-\frac{\sqrt{3}}{\sqrt{n_k}}, \frac{\sqrt{3}}{\sqrt{n_k}}\right]$
- Or also account for fan-out:
  - $W_{k_{init}} \sim Uniform\left[-\frac{\sqrt{6}}{\sqrt{n_k+n_{k+1}}}, \frac{\sqrt{6}}{\sqrt{n_k+n_{k+1}}}\right]$
- For ReLU use $\sqrt{2}$x larger ranges (since ½ units will be 0), so called Kaiming He initialization https://pytorch.org/docs/stable/nn.init.html#torch.nn.init.kaiming_uniform_

Glorot, Bengio, "Understanding the difficulty of training deep feedforward neural networks"

Batch normalization

http://arxiv.org/abs/1502.03167

- Normalize all activations in each minibatch!

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

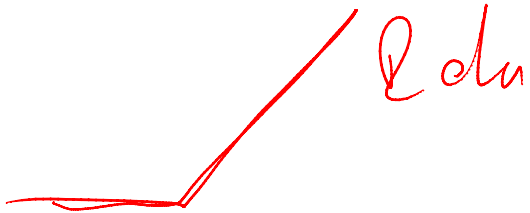
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- BN helps because it normalizes the scale of values that flow through the network.
  NB: it was originally said to prevent covariate shift (during training the upper layer's inputs are constantly changing), this is currently contested

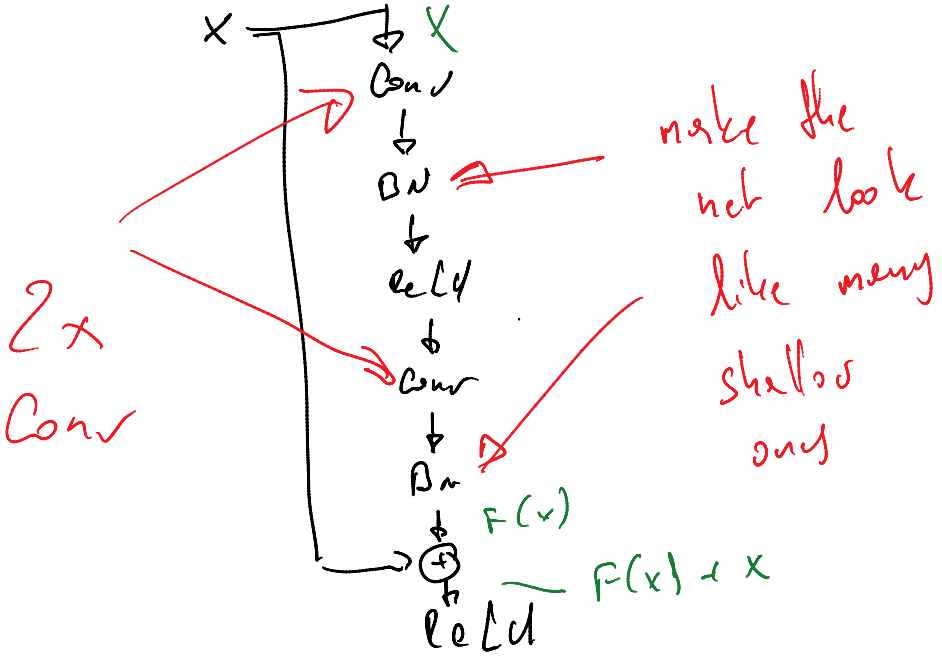
Avoid flat regions of the sigmoids

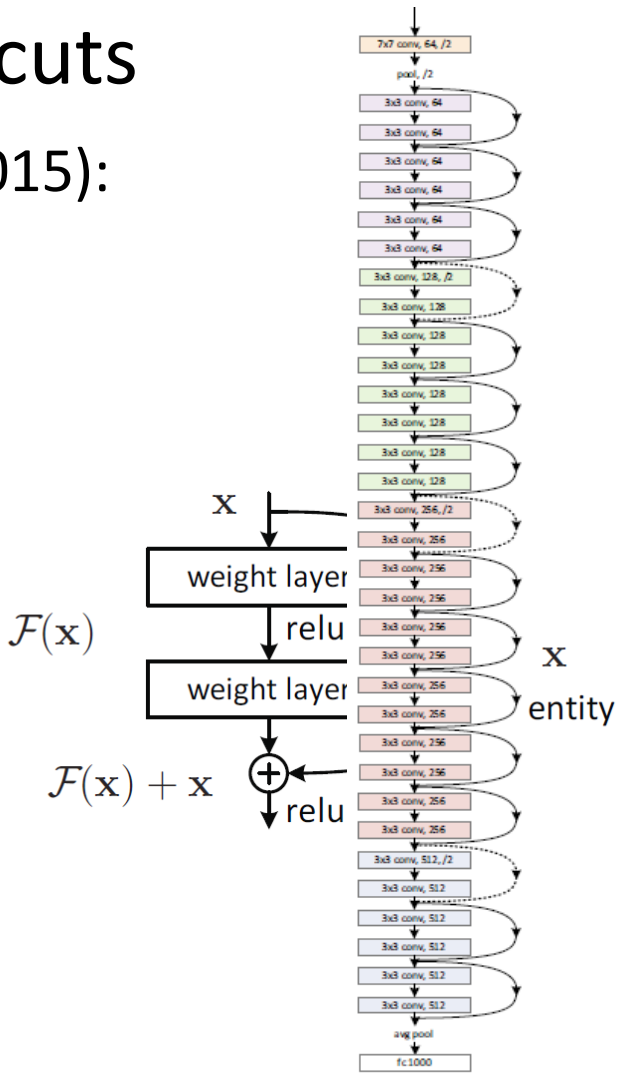
- For output layer and classification: use SoftMax and cross-entropy loss
- For inner layers:
  - Prefer tanh over log. sigmoid (tanh gives mean 0)
  - The contemporary default: ReLU
    http://machinelearning.wustl.edu/mlpapers/papers/icml2010_NairH10
  - Or Maxout http://arxiv.org/abs/1302.4389



Use gradient shortcuts

- Residual networks (SOTA Imagenet 2015): https://arxiv.org/abs/1512.03385
- Networks with more than 150 layers!

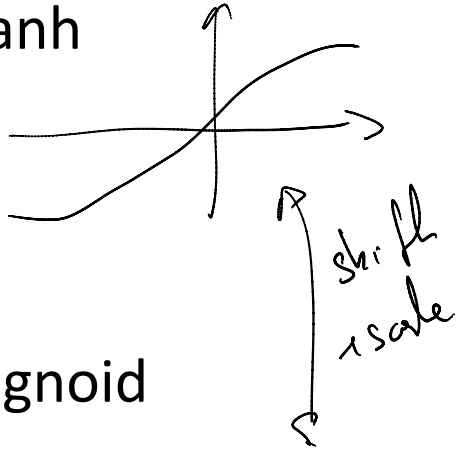




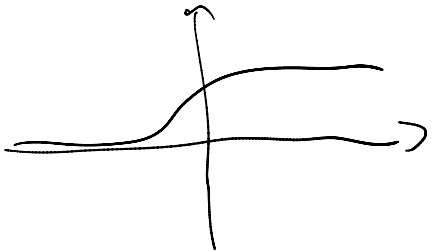
A set of activation functions

Classical:

Tanh

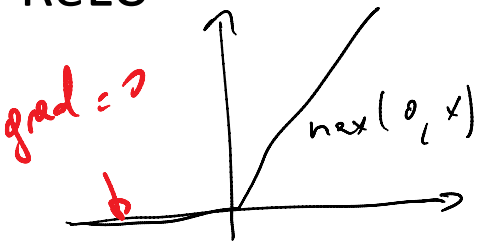


Signoid

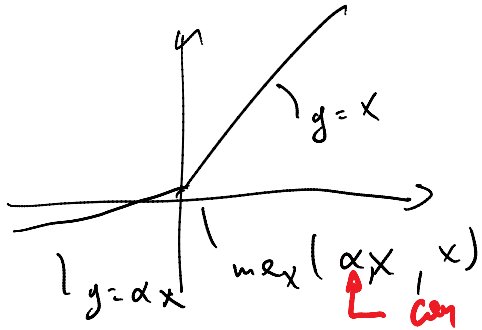


Contemporary

ReLU



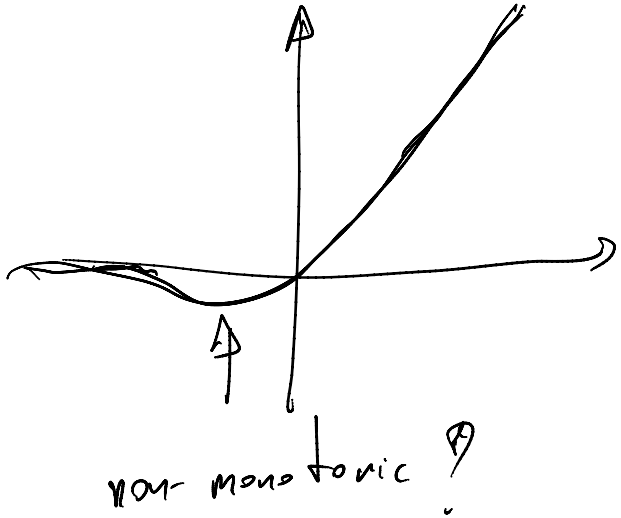
Leaky ReLU



Newcomers

Swish $x\sigma(\beta x)$

(arxiv.org/abs/1710.05941)



Babysitting the training process

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf from slide 61

Where to get more information

- "**Efficient backprop**" by Y. Le Cun
- „Practical Recommendation for Gradient-Based Training of Deep Architectures" Y. Bengio
- Slides by Geoff Hinton in his Coursera lectures:
  https://www.coursera.org/course/neuralnets
  and especially
  http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
- „Stochastic Gradient Descent Tricks" L. Bottou
- Slides from Stanford:
  http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf
  http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture7.pdf