Introduction to Machine Learning (IML)

LECTURE #12: DEEP LEARNING

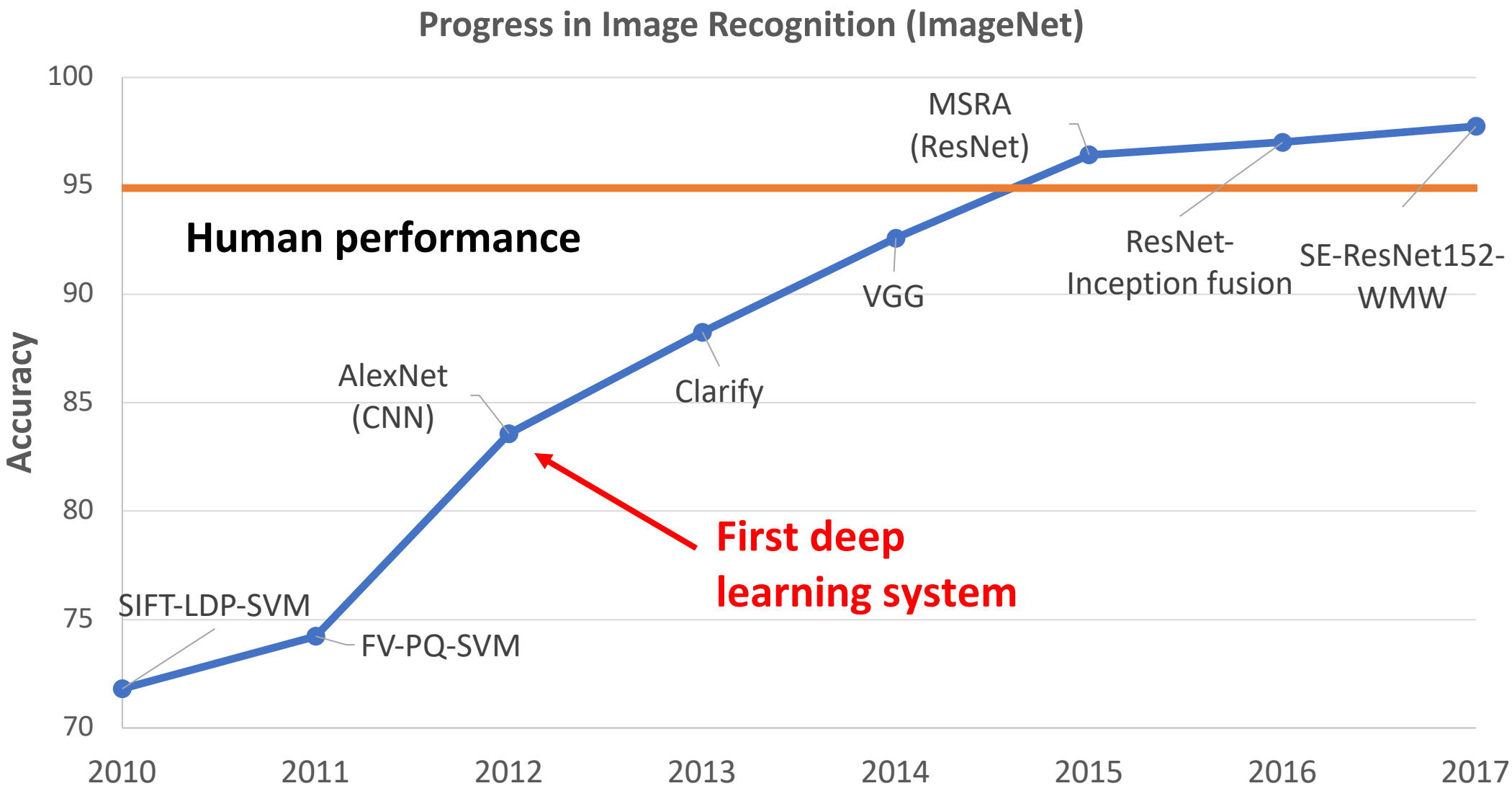
236756 – 2023-2024 WINTER – TECHNION

LECTURER: YONATAN BELINKOV

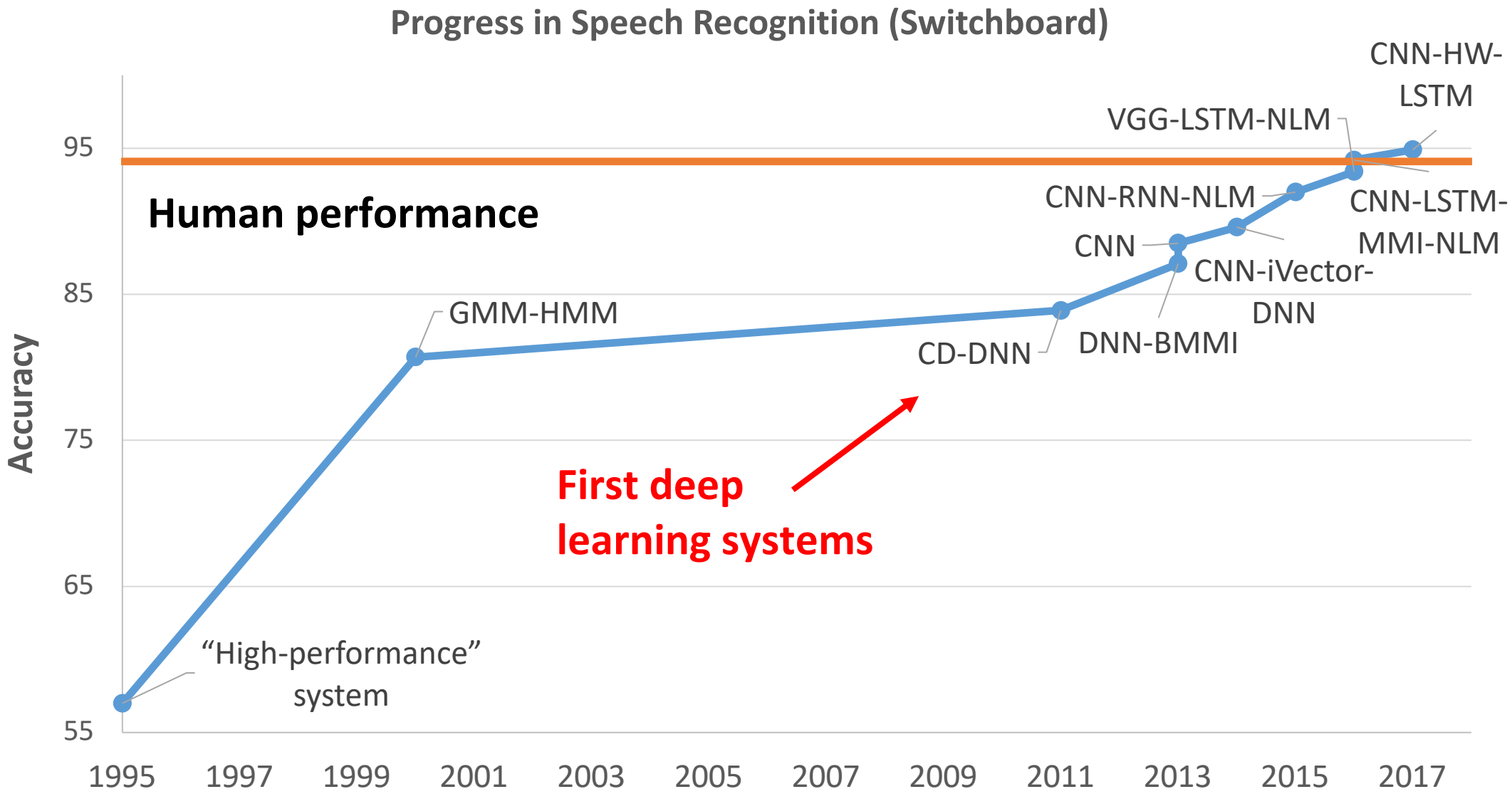
Today

- **Part III**: *more supervised learning*
  1. ~~Regression~~
  2. ~~Bagging and boosting~~
  3. ~~Generative models~~
  4. Deep learning (today)

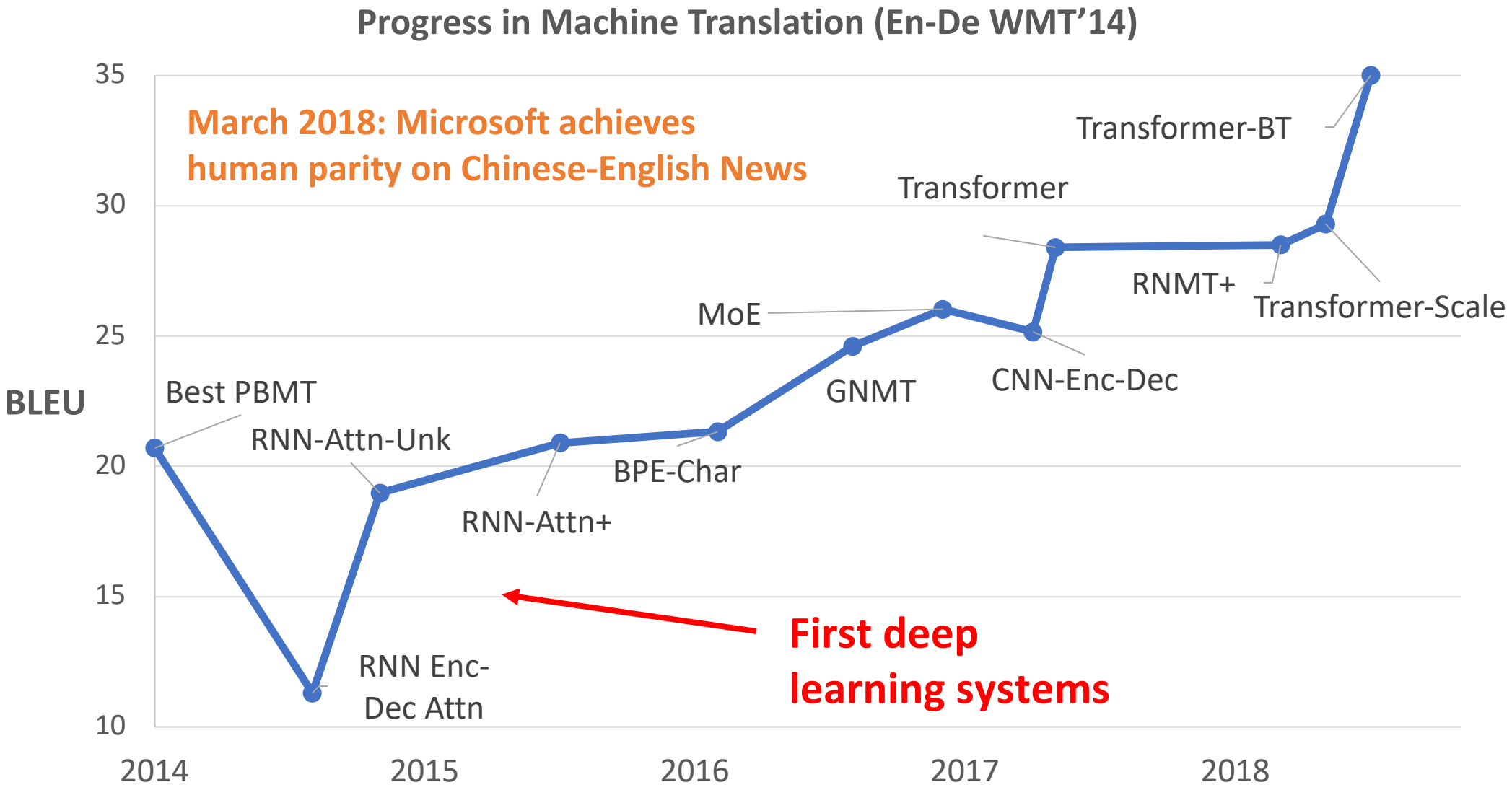
- **Goals for today:**
  - what it is (and what it isn't)
  - how it works (and how it breaks)
  - why it works (and when it doesn't)
- Be prepared for lots of jargon; we'll try to make sense out of it



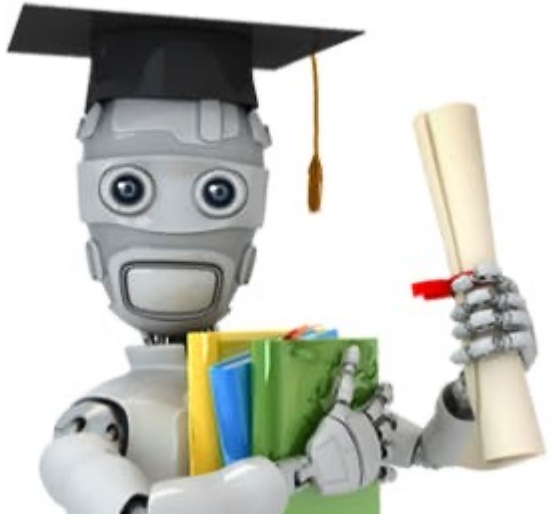
[Sources: eff.org, AI Index 2018]



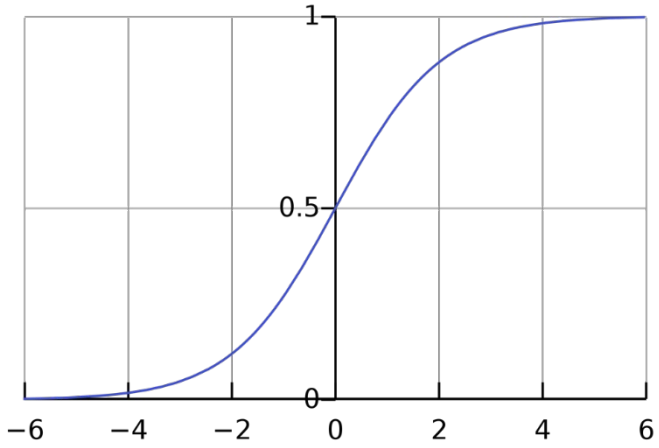
[Sources: eff.org, wer_we_are, George Saon's talk]



[Sources: eff.org, nlpprogress.com]



vs.



“the new electricity”

fancy logistic regression
(on steroids)

Meta-goal: tools for you to make your own conclusions

Representations

- Linear models: $h(x) = \text{sign}(w^\top x)$

Representations

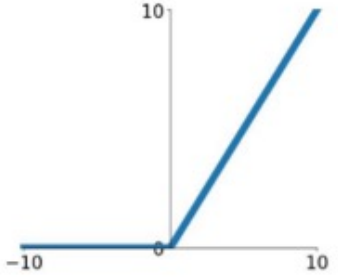
- Linear-in-representation models: $h(x) = \text{sign}\left(w^\top \phi(x)\right)$
- **Alternatives**:
  - linear: $\phi(x) = x$
  - hand-crafted: $\phi(x) = (\dots, g_i(x), \dots)$
  - kernels: $|\phi(x)| \gg |x|$
  - embedding: $|\phi(x)| \ll |x|$
  - **ensembles**: $\phi(x) = \left(h_1(x), \dots, h_k(x)\right)$ $\leftarrow$ $H(x) = \sum_i \alpha_i h_i(x) = \alpha^\top \phi(x)$
    - **bagging**: learn $\{h_i\}$ independently
    - **boosting**: learn $\{h_i\}$ sequentialy
    - **neural nets:** learn $\{h_i\}$ *end-to-end*

Logic gates: A motivating example

- Consider the perceptron decision rule: $\text{sign}(v^\top x + b)$
- Can a perceptron implement logic gates?
  - What values of v_1, v_2, b will implement AND? OR?
- Yes!
  - AND: set $v_1 = v_2 = 1, b = -1$
  - OR: set $v_1 = v_2 = 2, b = -1$
- What about XOR?
- Classic result: A linear model can't implement XOR (Minsky & Papert, 1969)

x_1	x_2	AND	OR	XOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

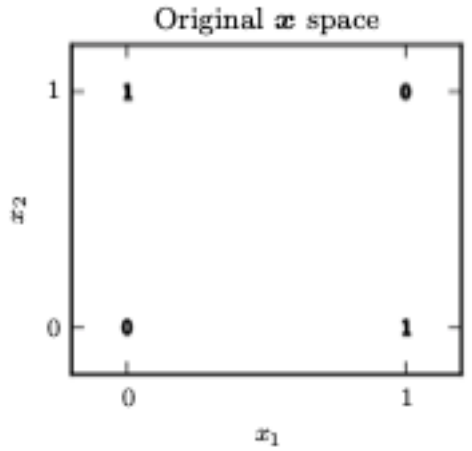
ReLU:
$\sigma(x) = \max\{0, x\}$



Logic gates: A motivating example

- Linear models like $h(x) = v^\top x + b$ can't implement XOR
- Idea: transform the feature space, such that a linear model works
- Define $h(x) = v^\top \phi(x) + b$, where $\phi(x) = \text{ReLU}(W^T x + c)$
  So: $h(x) = v^\top \max\{0, W^T x + c\} + b$
- It turns out, this is a solution:
  $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, c = [0; -1]^T, v = [1; -2]^T, b = 0$
- Why?

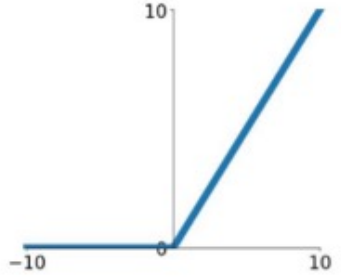
$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \xrightarrow{W} \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \xrightarrow{c} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{\text{RELU}} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{v} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Learned space

[Deep Learning Book]

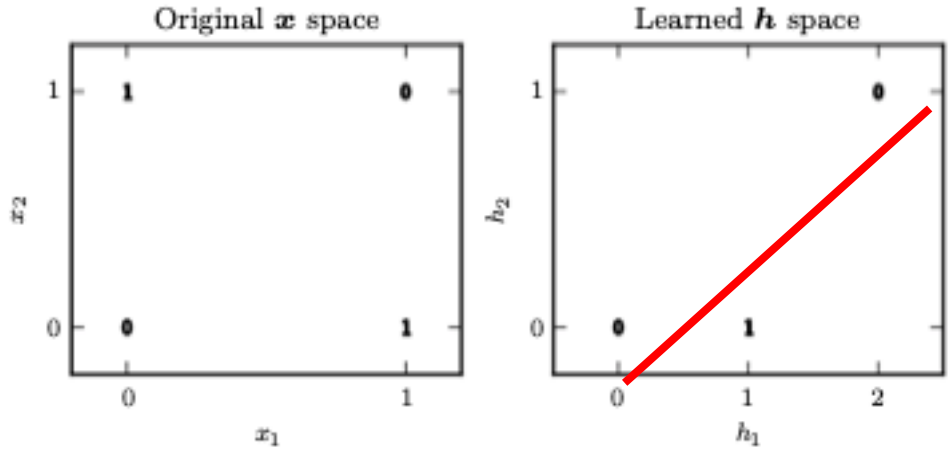
ReLU:
$\sigma(x) = \max\{0, x\}$



Logic gates: A motivating example

- Linear models like $h(x) = v^\top x + b$ can't implement XOR
- Idea: transform the feature space, such that a linear model works
- Define $h(x) = v^\top \phi(x) + b$, where $\phi(x) = \text{ReLU}(W^T x + c)$
  So: $h(x) = v^\top \max\{0, W^T x + c\} + b$
- It turns out, this is a solution:
  $W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, c = [0; -1]^T, v = [1; -2]^T, b = 0$
- Why?

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \xrightarrow{W} \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \xrightarrow{c} \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{\text{RELU}} \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \xrightarrow{v} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$



Learned space

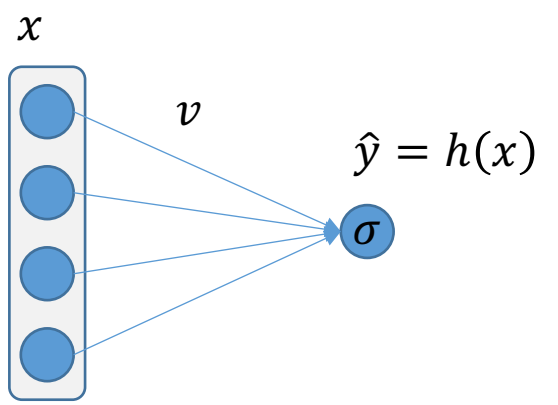
[Deep Learning Book]

Artificial Neural Networks

$$h(x) = \text{sign}(v^\top x) \approx \sigma(v^\top x)$$

non-linearity
(e.g., sigmoid)

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

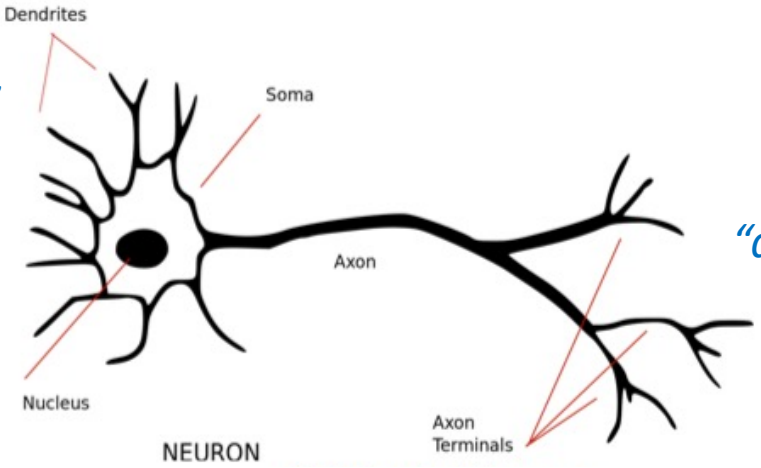


perceptron

$\approx$

very (very)
loose relation…

input =
“perception”



output =
“activation”

Artificial Neural Networks

representation

$$h(x) = \sigma\left(v^\top \phi(x)\right)$$

activation (=non-linearity)

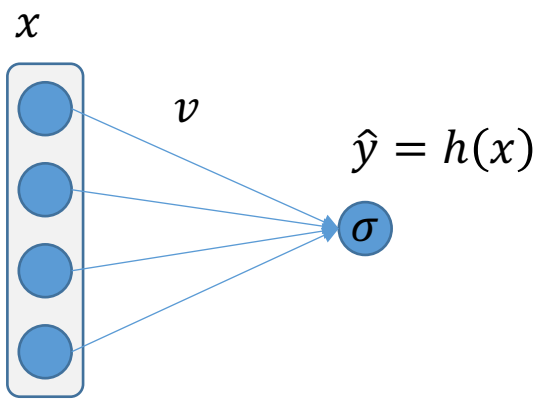
$$= \sigma(Wx)$$

d classifiers

$$\phi(x) = \left(h_1(x), \dots, h_d(x)\right)$$

$$= \left(\sigma(w_1^\top x), \dots, \sigma(w_d^\top x)\right)$$

$$= \sigma(Wx) = z$$



$$\sigma\left(\boxed{W}\ \boxed{x}\right) = \boxed{z}$$

$\mathbb{R}^{d\times d}$ $\mathbb{R}^d$ $\mathbb{R}^d$

Artificial Neural Networks

representation

$$h(x) = \sigma\left(v^\top \phi(x)\right)$$

activation (=non-linearity)

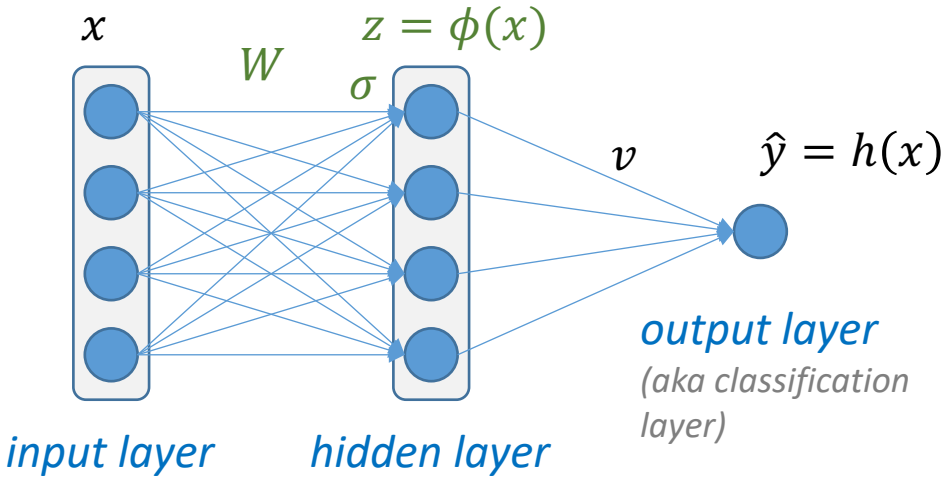
$$= \sigma(Wx)$$

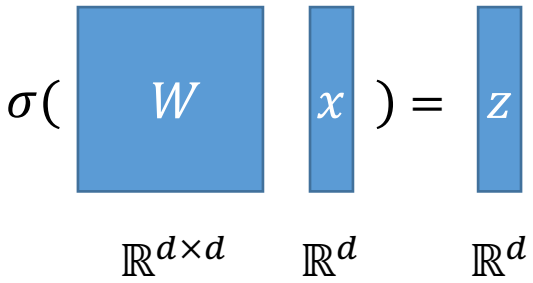
d classifiers

$$\phi(x) = \left(h_1(x), \dots, h_d(x)\right)$$

$$= \left(\sigma(w_1^\top x), \dots, \sigma(w_d^\top x)\right)$$

$$= \sigma(Wx) = z$$



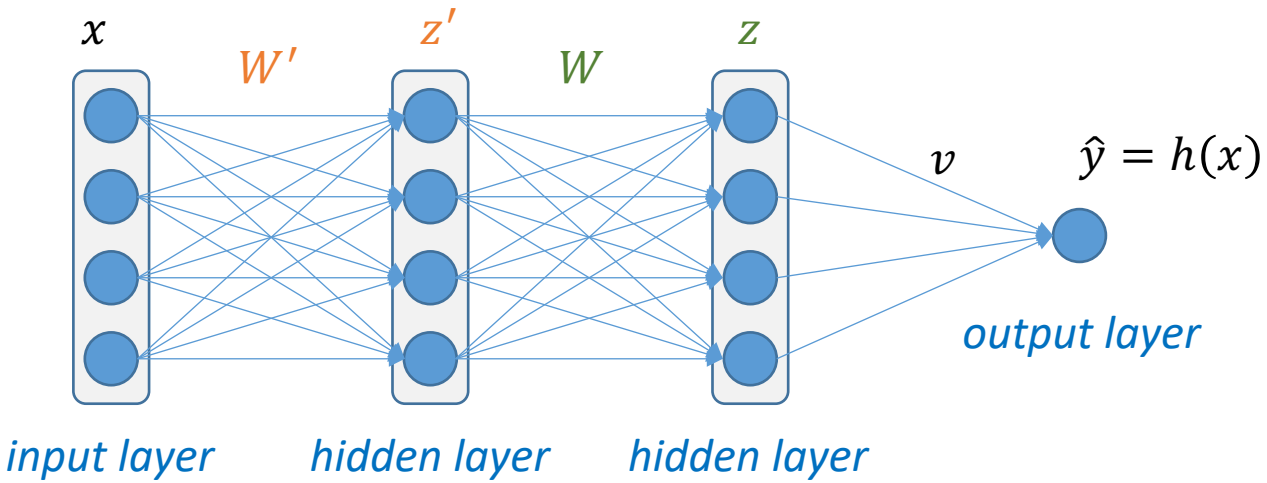


Artificial Neural Networks

$$h(x) = \sigma\left(v^\top \phi(x)\right)$$
$$= \sigma(W\phi'(x))$$
$$= \sigma(W'x)$$

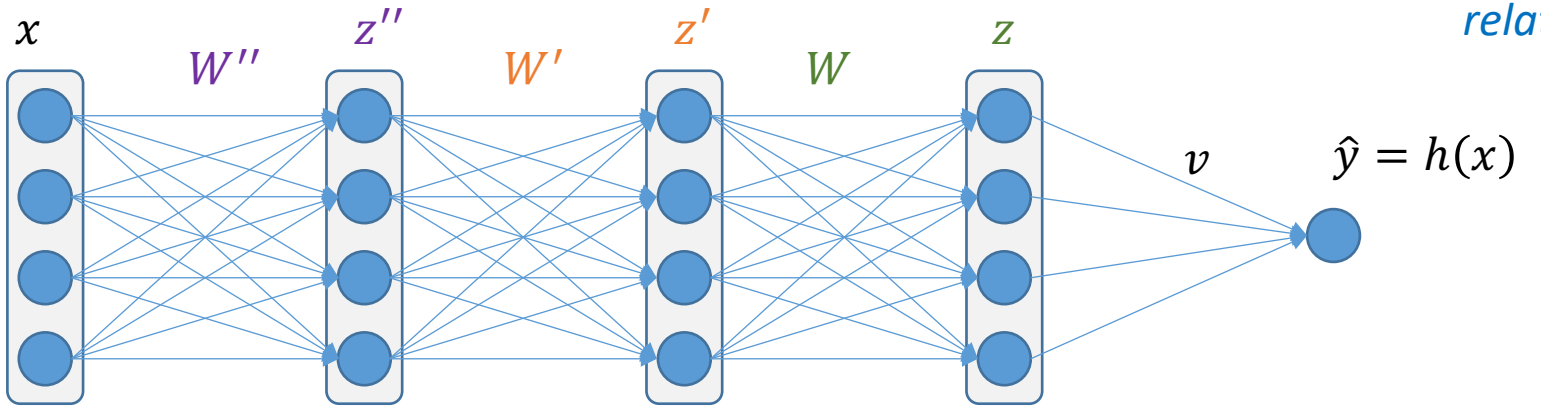
d "classifiers"

$$\phi'(x) = \left(h'_1(x), \ldots, h'_d(x)\right)$$
$$= \left(w'^\top_1 x, \ldots, w'^\top_d x\right)$$
$$= W'x$$

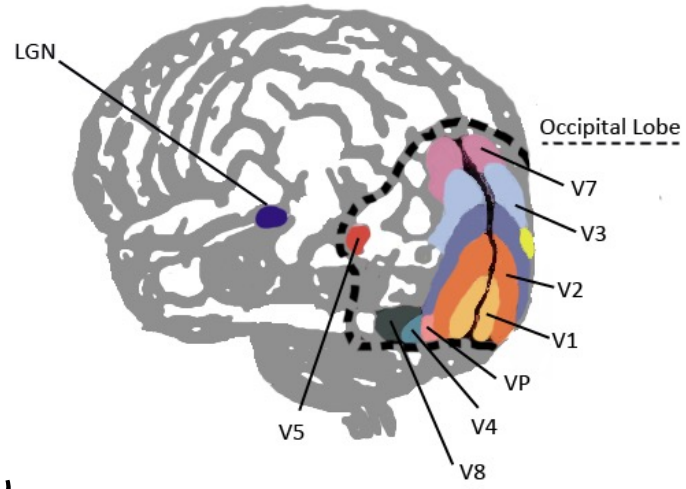


Artificial Neural Networks

$$h(x) = \sigma\left(v^\top \phi(x)\right)$$
$$= \sigma(W\phi'(x))$$
$$= \sigma(W'\phi''(x))$$
$$= \sigma(\dots)$$



multi-layered perceptron (MLP)



≈

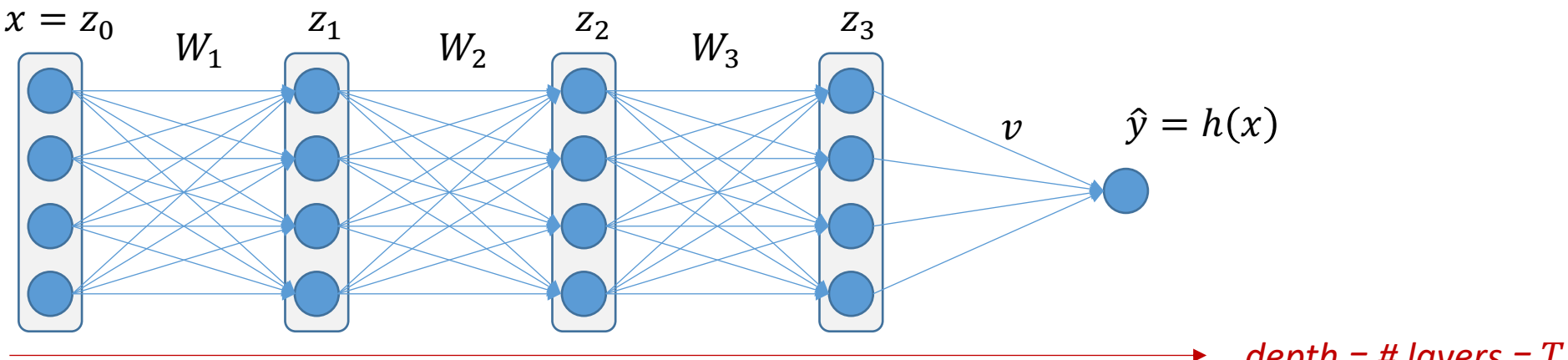
even looser relation...

Artificial Neural Networks

$$h(x) = \sigma\left(v^\top \phi_T(x)\right), \qquad \phi_t(x) = \sigma\left(W_t \phi_{t-1}(x)\right), \qquad \phi_0(x) = x$$

- or -

$$h(x) = \sigma(v^\top \sigma(W_T \sigma(W_{T-1} \sigma(\dots \sigma(Wx)))))$$



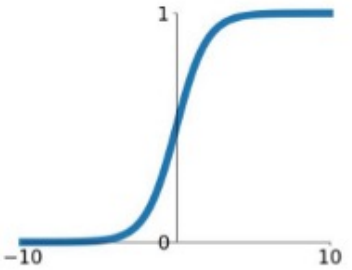
multi-layered perceptron (MLP)

Modeling

- Flexible model class – can vary:
  - depth (# hidden layers)
  - width (# dimensions per layer; can vary)
  - activations
  - parameters (connections)
  - …
- **Today**: focus on fully-connected, fixed width
  - Will preview other architectures

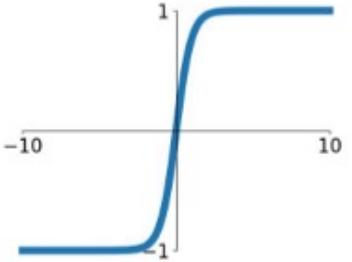
sigmoid:

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



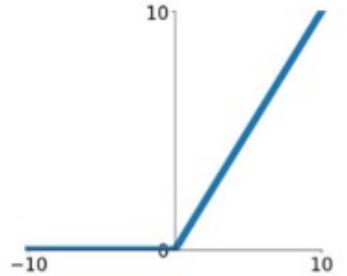
tanh:

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



ReLU:

$$\sigma(x) = \max\{0, x\}$$

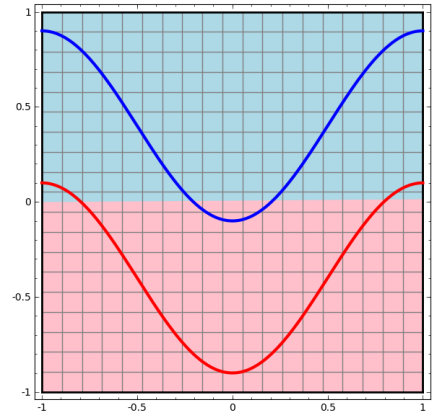


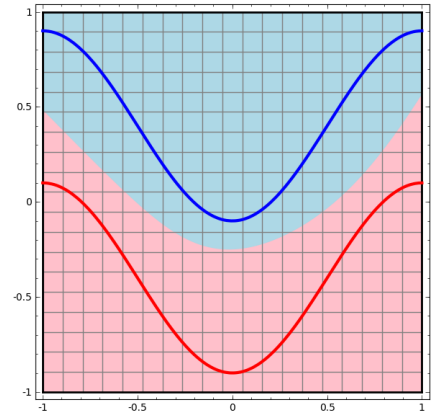
Neural representation

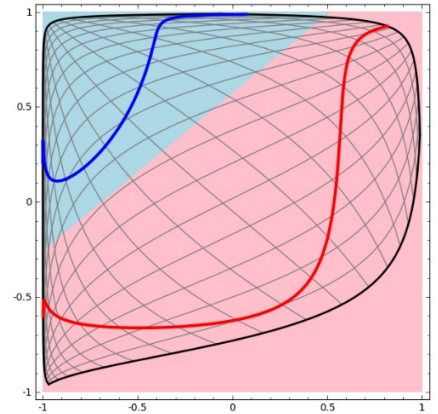
- **Objective**: $\underset{v,\{W_t\}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} \ell\big(y_i, h(x_i)\big)$

  where: $h(x) = \sigma\big(v^\top \phi_T(x)\big), \quad \phi_t(x) = \sigma\big(W_t \phi_{t-1}(x)\big) \in \mathbb{R}^d$

- Typically ℓ is cross-entropy – like logistic regression!
  - Recall: $\ell\big(y, h(x)\big) = -y \log h(x) + (1-y)\log(1-h(x))$
- Note ϕ are optimized for prediction, though indirectly
- Can be interpreted as classifiers, but only loosely
- More precise to think of them as "bending space"
  (or as moving points in space, at least for fixed-width networks)



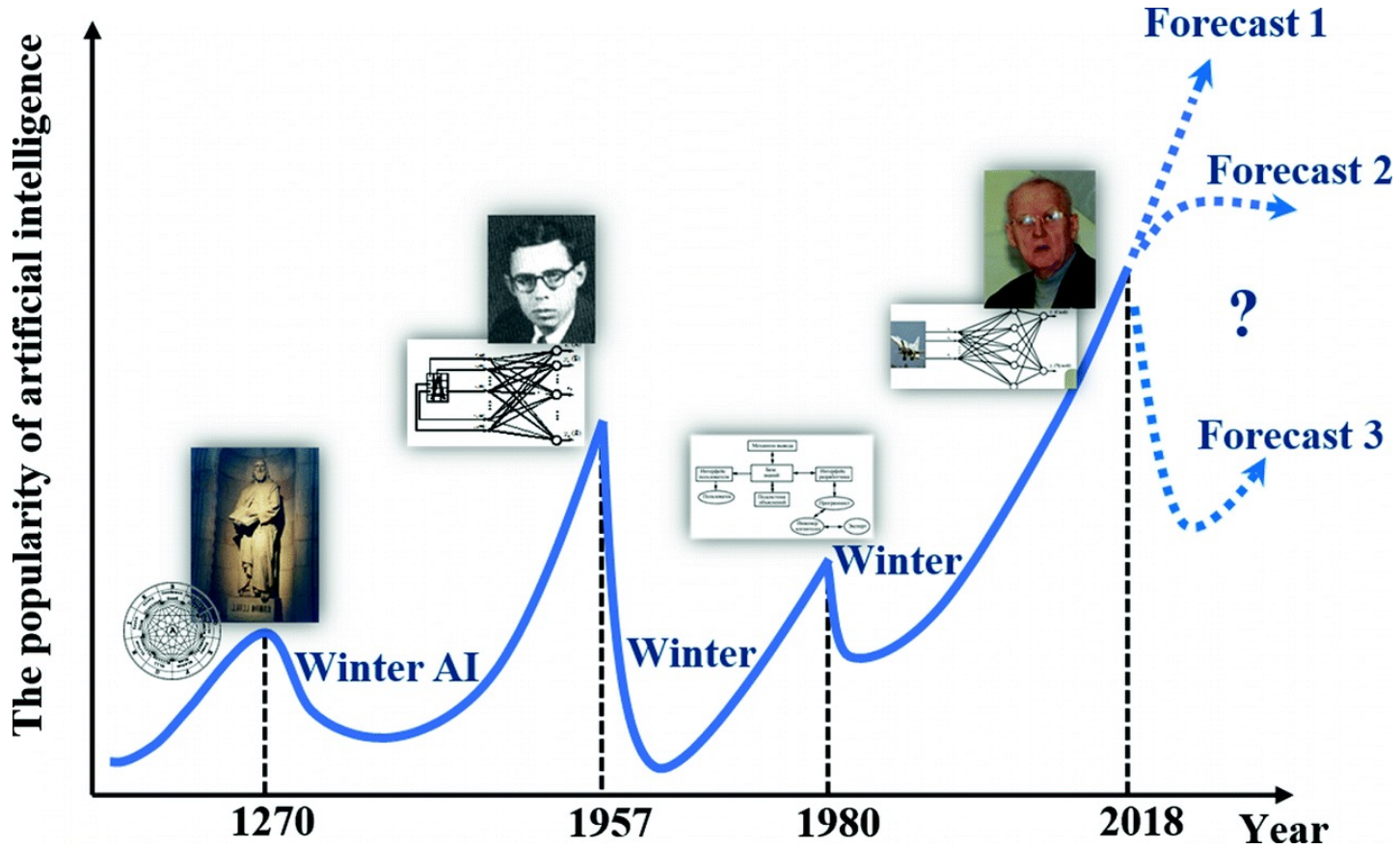




http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/

Why deep

- XOR (= need at least one hidden layer) result triggered “AI Winter”



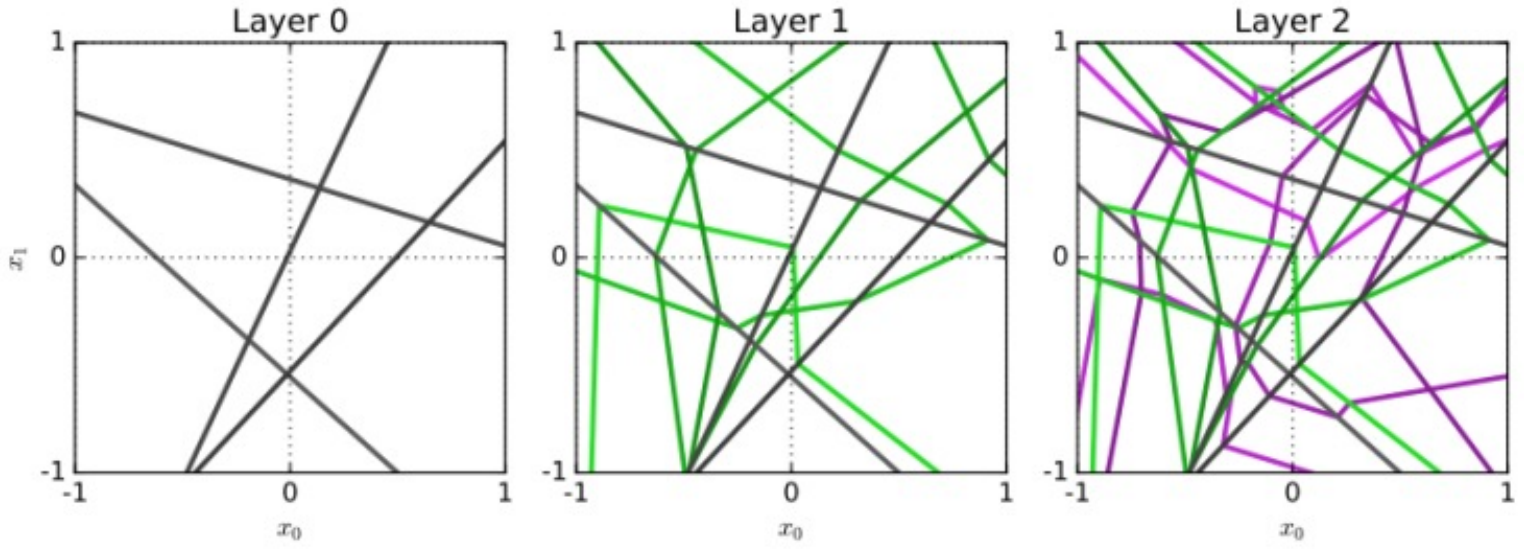
https://link.springer.com/chapter/10.1007/978-3-030-22493-6_2

Why deep

- XOR (= need at least one hidden layer) result triggered “AI Winter”
- But turns out even a single hidden layer is quite powerful
- **Classic result**:
  NNs with one hidden layer are universal approximators – can express any* function
  (*Lipschitz continuous; up to arbitrary precision)
- **Corollary**: anything k hidden layers can do, a single hidden layer can do too
  (in terms of expressivity; in terms of optimization, depth causes all kinds of trouble)
- But – **another classic result**: width needs to be *exponential* (in d)
- **Corollary:** practically, depth is very helpful

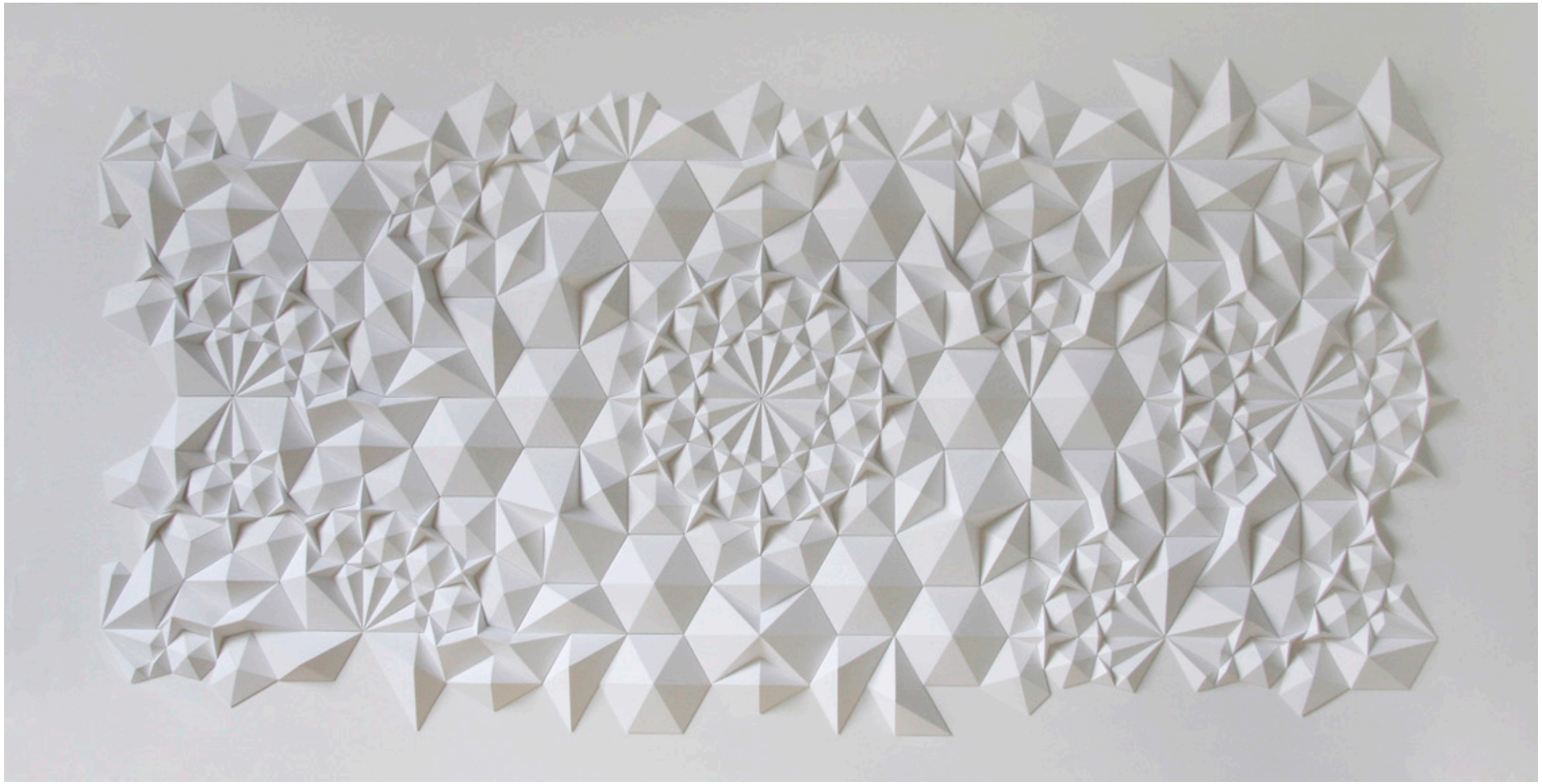
Why deep

- Intuitively, depth exponentiates capacity (vs width, which grows polynomially)
- (Width remains important! It sets the “embedded dimension”)
- Depth is great, but adds structure (or constraints, or symmetries):



- Paper folding analogy: https://www.youtube.com/watch?v=e5xKayCBOeU [@7:38]

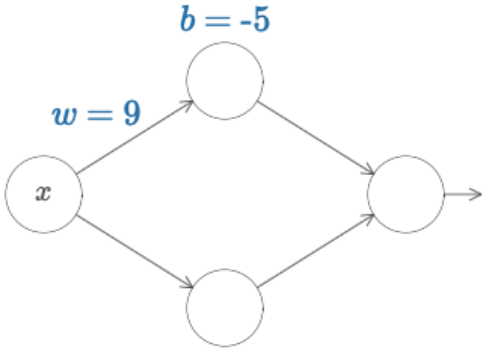
https://neuralrant.wordpress.com/2017/05/04/neural-networks-dynamics/

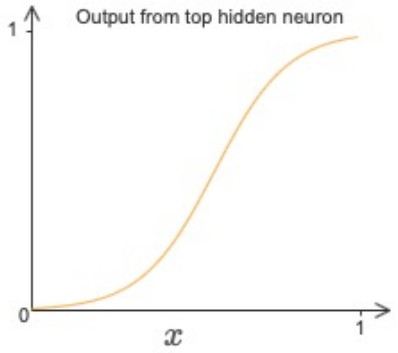


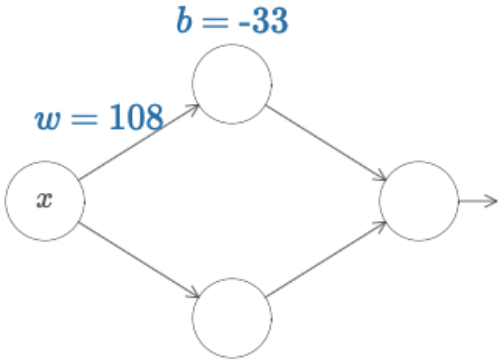
structure is great – if you know how to use it! (more on this later)

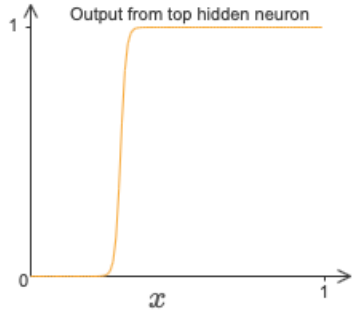
Universal approximation theorem

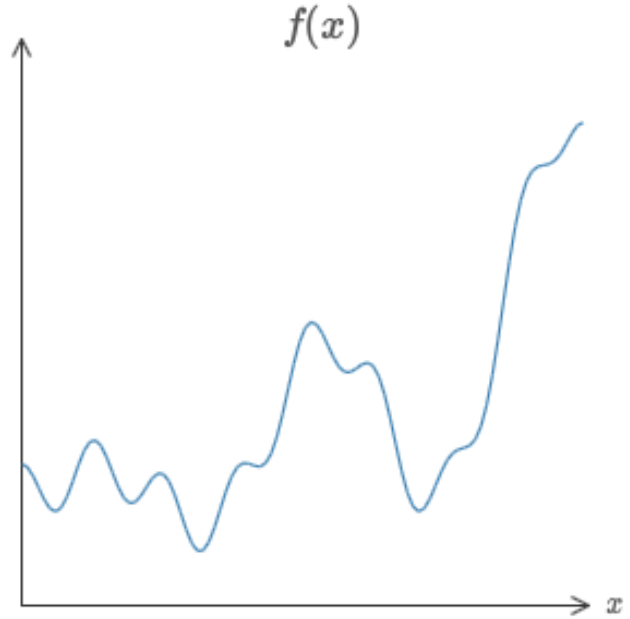
- Theorem (informal): NNs with one hidden layer can express any* function
  (*Lipschitz continuous; up to arbitrary precision)
- **Proof sketch**:
  - Each neuron can approximate a step function







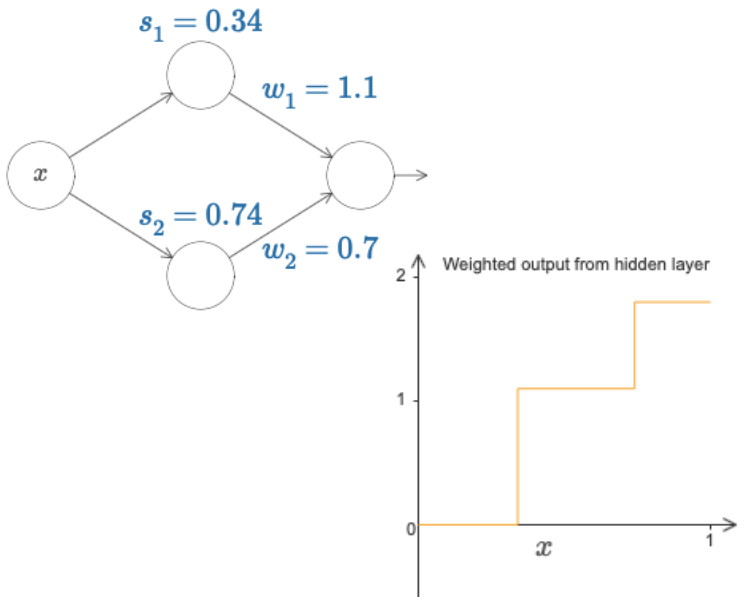


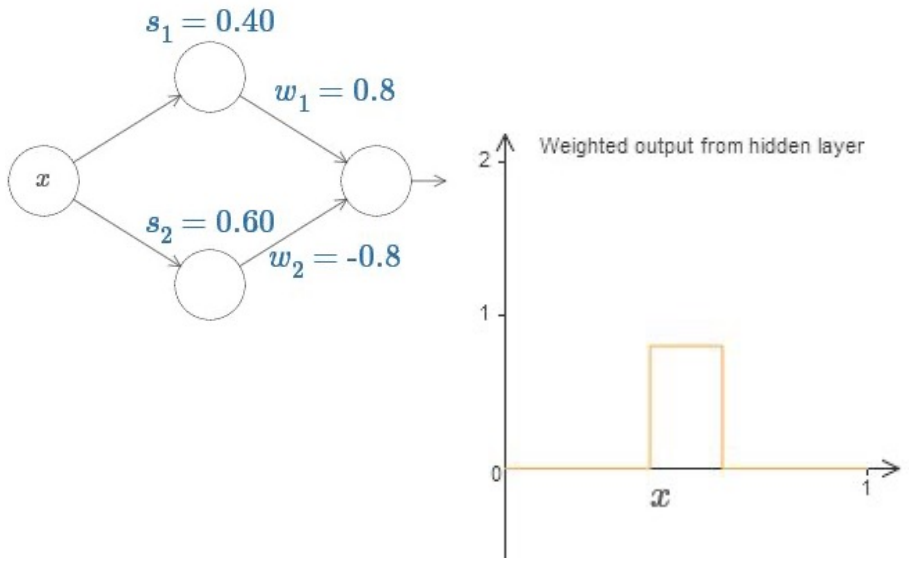


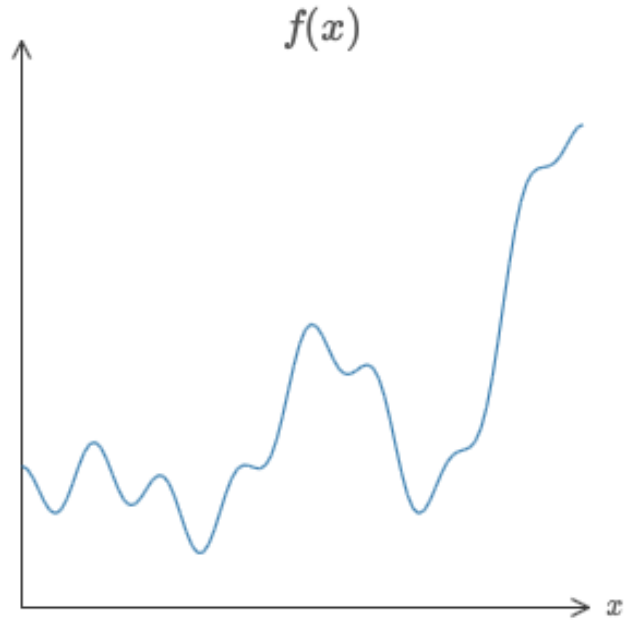
http://neuralnetworksanddeeplearning.com/chap4.html

Universal approximation theorem

- Theorem (informal): NNs with one hidden layer can express any* function
  (*Lipschitz continuous; up to arbitrary precision)
- **Proof sketch**:
  - Each neuron can approximate a step function
  - Two neurons approximate multi-step functions, or bumps



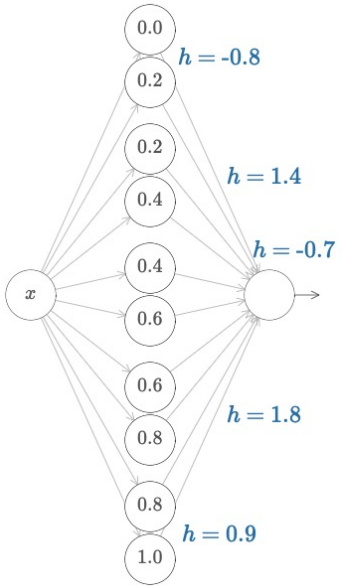


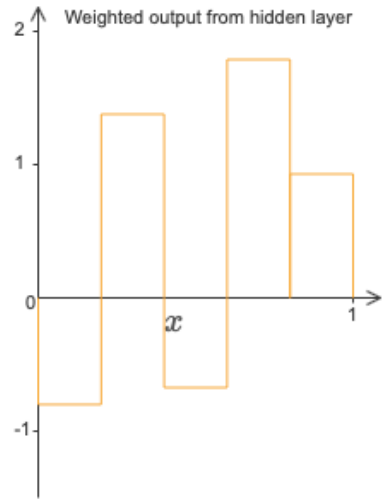


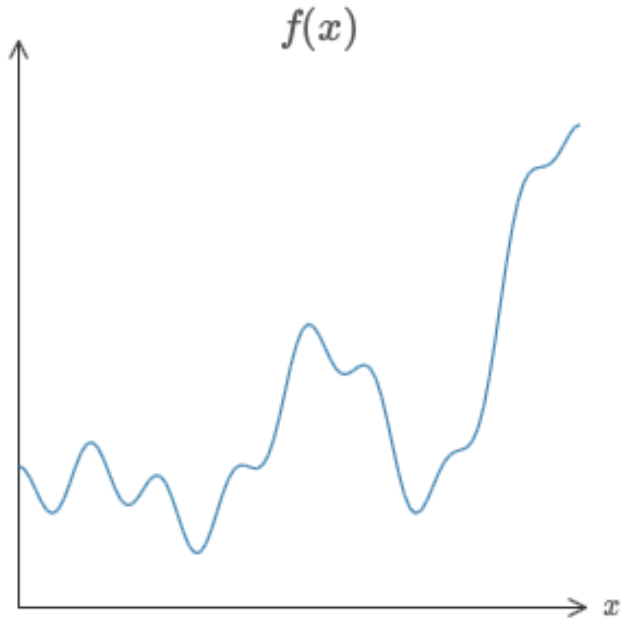
http://neuralnetworksanddeeplearning.com/chap4.html

Universal approximation theorem

- Theorem (informal): NNs with one hidden layer can express any* function
  (*Lipschitz continuous; up to arbitrary precision)
- **Proof sketch**:
  - Each neuron can approximate a step function
  - Two neurons approximate multi-step functions, or bumps
  - Multiple pairs of neurons make multiple bumps



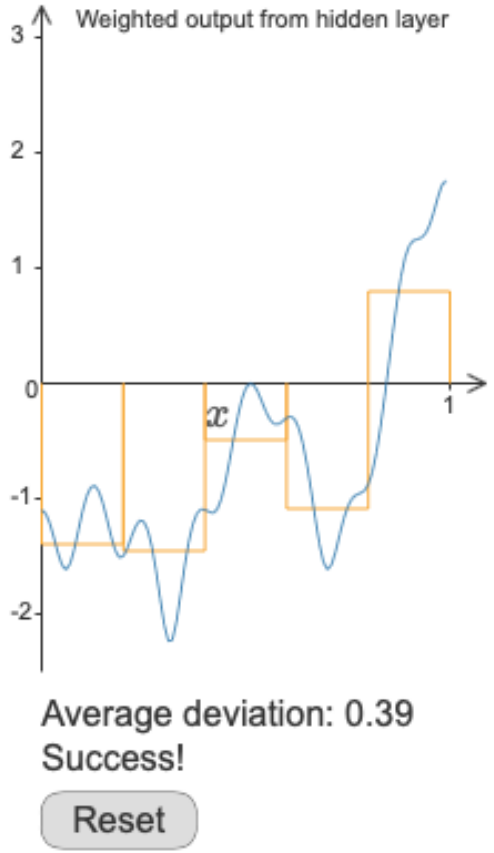




http://neuralnetworksanddeeplearning.com/chap4.html

Universal approximation theorem

- Theorem (informal): NNs with one hidden layer can express any* function
  (*Lipschitz continuous; up to arbitrary precision)
- **Proof sketch**:
  - Each neuron can approximate a step function
  - Two neurons approximate multi-step functions, or bumps
  - Multiple pairs of neurons make multiple bumps
  - Add lots of bumps and adjust their heights to approximate the function



http://neuralnetworksanddeeplearning.com/chap4.html

The works

Optimization

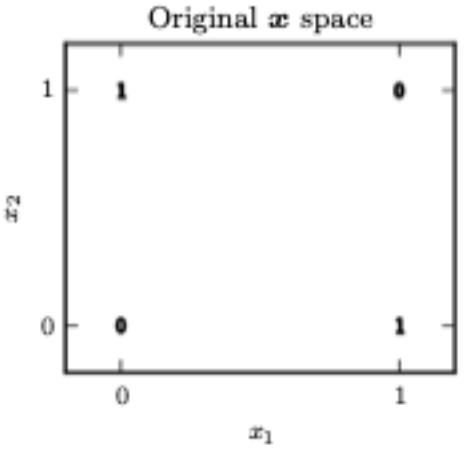
- Learning objective:

$$\underset{v,\{W_t\}}{\operatorname{argmin}} \frac{1}{m}\sum_{i=1}^{m} \ell\big(y_i, h(x_i)\big), \qquad h(x) = \sigma\big(v^\top \phi_T(x)\big), \quad \phi_t(x) = \sigma\big(W_t \phi_{t-1}(x)\big),\ \phi_0(x) = x$$

- Objective is non-convex (even if loss is convex, activations are not)
- But it is (sub-)differentiable – can use gradient descent!
- **Need**: $\frac{\partial L}{\partial v}$ ($\in \mathbb{R}^d$), $\frac{\partial L}{\partial W_T}$, $\frac{\partial L}{\partial W_{T-1}}$ ($\in \mathbb{R}^{d\times d}$), … , $\frac{\partial L}{\partial W_1}$ → lots of gradients to compute!
- Luckily, can be done efficiently using **chain rule** (aka backpropagation)

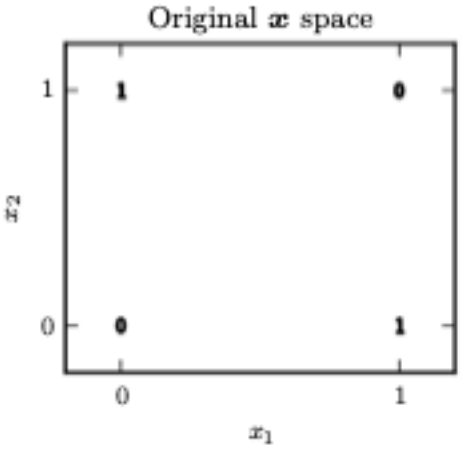
Computing gradients

- Back to our XOR-calculating network: $h(x) = v^{\top}\phi(x) + b$, where $\phi(x) = \mathrm{ReLU}(W^T x + c)$
- Let's ignore the biases and consider: $h(x) = v^T \mathrm{ReLU}(W^T x)$
- Objective: $L(h) = \sum_i \ell\big(y_i, h(x_i)\big) = \sum_i \frac{1}{2}\big(y_i - h(x_i)\big)^2 = \sum_i \frac{1}{2}\big(y_i - v^{\top}\mathrm{ReLU}(W^T x)\big)^2$
- $\partial L = \sum_i \partial \ell_i$, so we need individual elements in the sum: $\partial \ell(y, h(x))$
- $\frac{\partial l}{\partial v} = (y - v^T \mathrm{ReLU}(W^T x))\mathrm{ReLU}(W^T x)$
- $\frac{\partial l}{\partial W} = \big(y - v^T \mathrm{ReLU}(W^T x)\big)\mathrm{ReLU}'(W^T x) x v^T$



Computing gradients

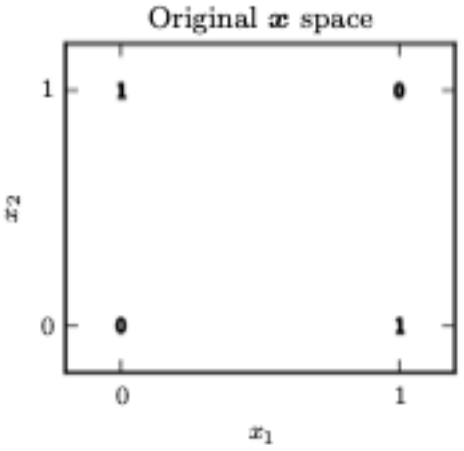
- Back to our XOR-calculating network: $h(x) = v^{\top}\phi(x) + b$, where $\phi(x) = \mathrm{ReLU}(W^T x + c)$
- Let's ignore the biases and consider: $h(x) = v^T \mathrm{ReLU}(W^T x)$
- Objective: $L(h) = \sum_i \ell\big(y_i, h(x_i)\big) = \sum_i \frac{1}{2}\big(y_i - h(x_i)\big)^2 = \sum_i \frac{1}{2}\big(y_i - v^{\top}\mathrm{ReLU}(W^T x)\big)^2$
- $\partial L = \sum_i \partial \ell_i$, so we need individual elements in the sum: $\partial \ell(y, h(x))$
- $\frac{\partial l}{\partial v} = (y - v^T \mathrm{ReLU}(W^T x))\mathrm{ReLU}(W^T x)$
- $\frac{\partial l}{\partial W} = \big(y - v^T \mathrm{ReLU}(W^T x)\big)\mathrm{ReLU}'(W^T x) x v^T$
- Notice: repeated calculations and values we get from the "forward pass"



Computing gradients

$$\ell(y, h(x)) = \frac{1}{2}\left(y - v^T \mathrm{ReLU}(W^T x)\right)^2$$

- Computing gradients the normal way, we got:
  - $\frac{\partial l}{\partial v} = (y - v^T \mathrm{ReLU}(W^T x))\mathrm{ReLU}(W^T x)$
  - $\frac{\partial l}{\partial W} = \left(y - v^T \mathrm{ReLU}(W^T x)\right)\mathrm{ReLU}'(W^T x) x v^T$
- Let's do this again, but this time more efficiently
- Denote a $\coloneqq W^T x$, so $h(x) = v^T \mathrm{ReLU}(a)$ and $\ell\left(y, h(x)\right) = \frac{1}{2}\left(y - v^T \mathrm{ReLU}(a)\right)^2$
- Then $\frac{\partial l}{\partial v} = (y - v^T \mathrm{ReLU}(a))\mathrm{ReLU}(a)$
- By chain rule: $\frac{\partial l}{\partial W} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial W}$
  - $\frac{\partial l}{\partial a} = \left(y - v^T \mathrm{ReLU}(a)\right)\mathrm{ReLU}'(a) v^T$
  - $\frac{\partial a}{\partial W} = x$
  - $\frac{\partial l}{\partial W} = \left(y - v^T \mathrm{ReLU}(a)\right)\mathrm{ReLU}'(a) x v^T = \left(y - v^T \mathrm{ReLU}(W^T x)\right)\mathrm{ReLU}'(W^T x) x v^T$



Computing gradients: a more general case

- Loss: $\ell\big(y, h(x)\big) = \frac{1}{2}(y - h(x))^2 = \frac{1}{2}(y - v^T\phi(x)\ )^2$ (ignored the final non-linearity for simplicity)
- We need $\frac{\partial \ell}{\partial v}$, $\frac{\partial \ell}{\partial W_T}$, $\frac{\partial \ell}{\partial W_{T-1}}$, … → let's use W' for W_T, W'' for W_{T-1}, …

- $\phi(x) = \sigma\big(W\phi'(x)\big) = \sigma(a)$
- $\phi'^{(x)} = \sigma\big(W'\phi''(x)\big) = \sigma(a')$
- $\phi''^{(x)} = \cdots$

- $\frac{\partial \ell}{\partial v} = \big(y - v^T\phi(x)\big)\phi(x)$

Chain rule:

- $\frac{\partial l}{\partial W} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial W}$
- $\frac{\partial l}{\partial W'} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial W'}$
- $\frac{\partial l}{\partial W''} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial a''}\frac{\partial a''}{\partial W''}$
- …

Computing gradients: a more general case

- Loss: $\ell\big(y, h(x)\big) = \frac{1}{2}(y - h(x))^2 = \frac{1}{2}(y - v^T\phi(x)\,)^2$ (ignored the final non-linearity for simplicity)
- We need $\frac{\partial \ell}{\partial v}$, $\frac{\partial \ell}{\partial W_T}$, $\frac{\partial \ell}{\partial W_{T-1}}$, … → let's use W' for W_T, W'' for W_{T-1}, …

- $\phi(x) = \sigma\big(W\phi'(x)\big) = \sigma(a)$
- $\phi'^{(x)} = \sigma\big(W'\phi''(x)\big) = \sigma(a')$
- $\phi''^{(x)} = \cdots$

- $\frac{\partial \ell}{\partial v} = \big(y - v^T\phi(x)\big)\phi(x)$

Chain rule:

- $\frac{\partial l}{\partial W} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial W}$
- $\frac{\partial l}{\partial W'} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial W'}$
- $\frac{\partial l}{\partial W''} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial a''}\frac{\partial a''}{\partial W''}$
- …

Can save a lot of calculations!

Computing gradients: a more general case

- Loss: $\ell\big(y, h(x)\big) = \frac{1}{2}(y - v^T\phi(x)\,)^2 = \frac{1}{2}(y - v^T\sigma(a))^2$
- $\phi(x) = \sigma\big(W\phi'(x)\big) = \sigma(a)$
- $\phi'^{(x)} = \sigma\big(W'\phi''(x)\big) = \sigma(a')$
- $\phi''^{(x)} = \cdots$

Chain rule:

- $\frac{\partial l}{\partial W} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial W}$
- $\frac{\partial l}{\partial W'} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial W'}$
- $\frac{\partial l}{\partial W''} = \frac{\partial l}{\partial a}\frac{\partial a}{\partial a'}\frac{\partial a'}{\partial a''}\frac{\partial a''}{\partial W''}$
- …

Can save a lot of calculations!

- $\frac{\partial l}{\partial a} = \frac{\partial}{\partial a}\frac{1}{2}(y - v^T\sigma(a))^2 = \cdots$ → **compute once, share by all gradients**

computed on forward pass:

- $\frac{\partial a}{\partial W} = \frac{\partial}{\partial W}W\phi'(x) = \phi'(x)$
- $\frac{\partial a'}{\partial W'} = \frac{\partial}{\partial W'}W'\phi''(x) = \phi''(x)$
- …

New computations

- $\frac{\partial a}{\partial a'} = \frac{\partial}{\partial a'}W\phi'(x) = \frac{\partial}{\partial a'}W\sigma(a') =_{\text{ReLU}} \begin{cases} W & a' > 0 \\ 0 & a' \le 0 \end{cases}$
- $\frac{\partial a'}{\partial a''} = \frac{\partial}{\partial a''}W'\phi''(x) = \frac{\partial}{\partial a''}W'\sigma(a'') =_{\text{ReLU}} \begin{cases} W' & a'' > 0 \\ 0 & a'' \le 0 \end{cases}$
- …

Back Propagation

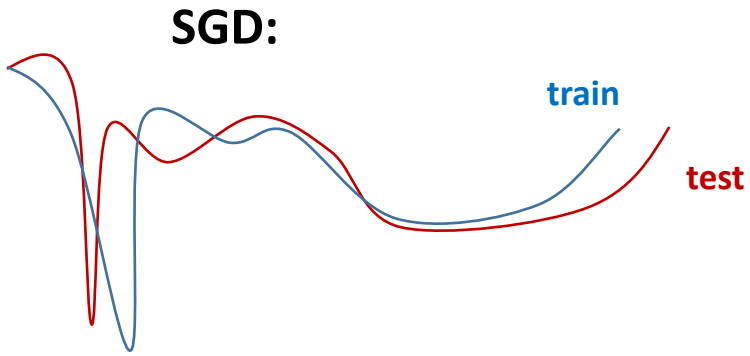
forward pass:

- for $t = 1, \dots, T-1$:
  - compute $h_t(x)$
- compute $h(x)$

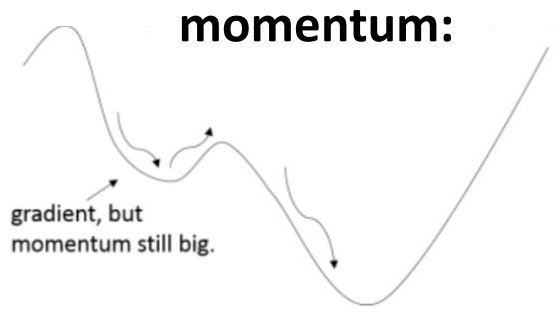
backward pass:

- $\frac{\partial \ell(h)}{\partial v} \leftarrow \big(y - v^\top h_{T-1}(x)\big) h_{T-1}(x)$
- compute $\delta_{T-1} = \frac{\partial \ell(h)}{\partial a}$
- for $t = T-1, \dots, 1$:
  - $\frac{\partial \ell(h)}{\partial W_t} \leftarrow \delta_t h_{t-1}(x)$ # $h_0(x) = x$
  - $\delta_{t-1} = \delta_t \frac{\partial a_t}{\partial a_{t-1}}$

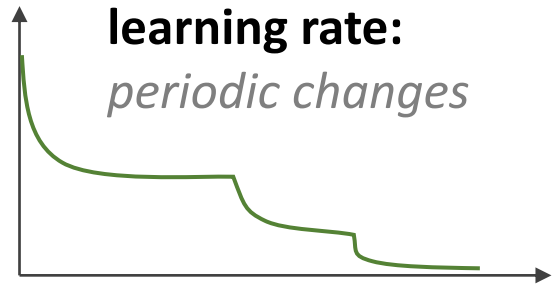
Optimization: non-convexity



- Non-convex optimization means **local minima**
- The deeper you go, the more local minima you get
- Some local minima are better than others!



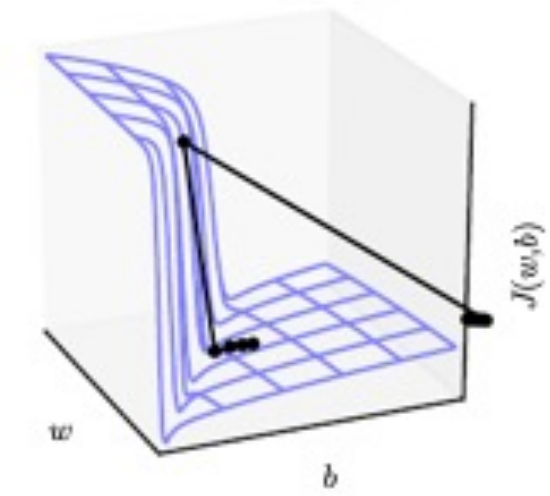
- Heuristics for ending up in good local minima:
  1. Use mini-batch SGD – noise helps "escape" small basins
  2. Apply momentum or advanced optimizers (e.g., ADAM)
  3. Learning rate crucial: trial and error, interim switching
  4. Run multiple random parameter initializations
     (pay attention to what's reported – average or max!)
  5. Many, many other tricks



- Much effort towards understanding optimization landscape
  (**example**: turns out you should actually worry more about saddle points)

Optimization: depth

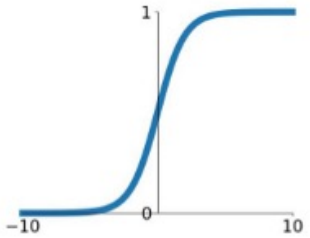
- Very deep networks suffer from vanishing or exploding gradients



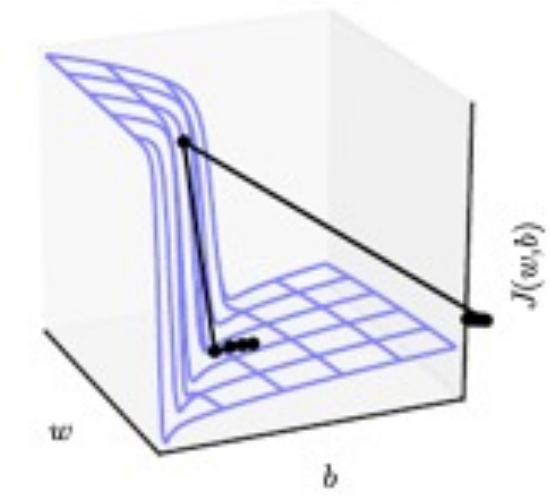
Optimization: depth

sigmoid:

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



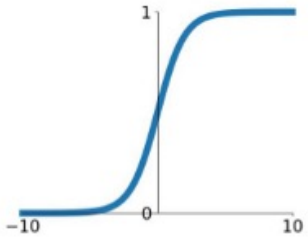
- Very deep networks suffer from vanishing or exploding gradients
- Reasons
  - Repeated applications of activations with small gradients (e.g., sigmoid)
  - Repeated applications of the same weight matrices
    Suppose W is used t times with decomposition $W = V\ \mathrm{diag}(\lambda)V^{-1}$, then gradients are scaled by $\mathrm{diag}(\lambda)^t$
    (Does not usually happen in simple feed-forward networks, common in other architectures)



Optimization: depth

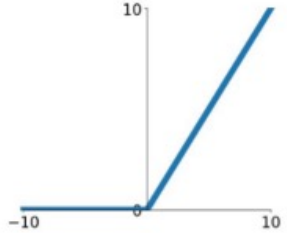
sigmoid:

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

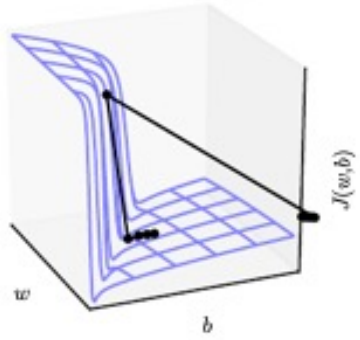


ReLU:

$$\sigma(x) = \max\{0, x\}$$



- Very deep networks suffer from vanishing or exploding gradients
- Reasons
  - Repeated applications of activations with small gradients (e.g., sigmoid)
  - Repeated applications of the same weight matrices
    Suppose W is used t times with decomposition $W = V\ \mathrm{diag}(\lambda)V^{-1}$, then gradients are scaled by $\mathrm{diag}(\lambda)^t$
    (Does not usually happen in simple feed-forward networks, common in other architectures)
- Solutions
  - Use different activations: ReLU solves vanishing gradients, but not exploding ones

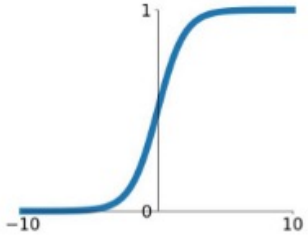


Optimization: depth

- Very deep networks suffer from vanishing or exploding gradients
- Reasons
  - Repeated applications of activations with small gradients (e.g., sigmoid)
  - Repeated applications of the same weight matrices
    Suppose W is used t times with decomposition $W = V\ \mathrm{diag}(\lambda)V^{-1}$, then gradients are scaled by $\mathrm{diag}(\lambda)^t$
    (Does not usually happen in simple feed-forward networks, common in other architectures)
- Solutions
  - Use different activations: ReLU solves vanishing gradients, but not exploding ones
  - Gradient clipping helps with exploding gradients: $if\ \|g\| > p: g \leftarrow \frac{gp}{\|g\|}$

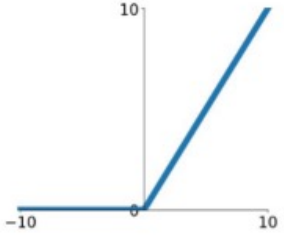
sigmoid:

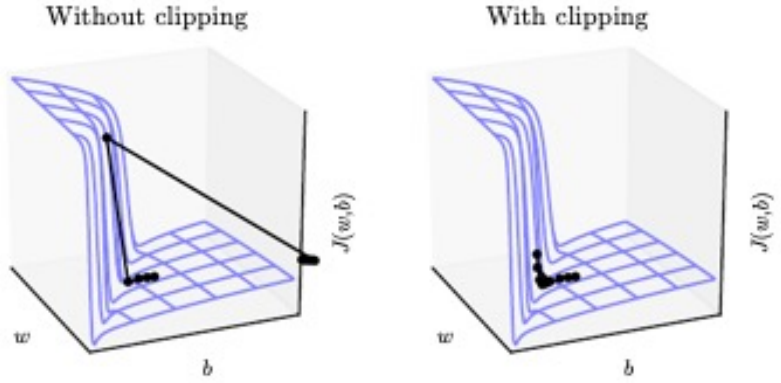
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



ReLU:

$$\sigma(x) = \max\{0, x\}$$

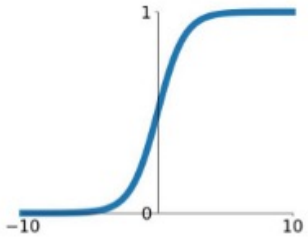




Optimization: depth

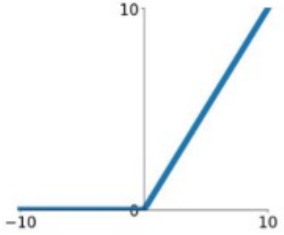
sigmoid:

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

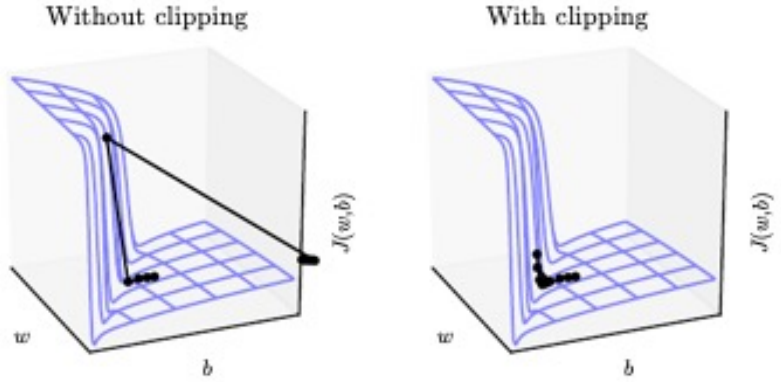


ReLU:

$$\sigma(x) = \max\{0, x\}$$



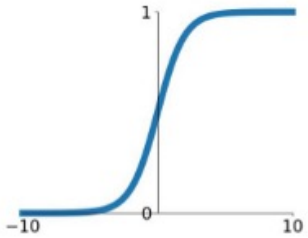
- Very deep networks suffer from vanishing or exploding gradients
- Reasons
  - Repeated applications of activations with small gradients (e.g., sigmoid)
  - Repeated applications of the same weight matrices
    Suppose W is used t times with decomposition $W = V\ \mathrm{diag}(\lambda)V^{-1}$, then gradients are scaled by $\mathrm{diag}(\lambda)^t$
    (Does not usually happen in simple feed-forward networks, common in other architectures)
- Solutions
  - Use different activations: ReLU solves vanishing gradients, but not exploding ones
  - Gradient clipping helps with exploding gradients: $if\ \|g\| > p: g \leftarrow \frac{gp}{\|g\|}$
  - Smart weight initialization also helps



Optimization: depth

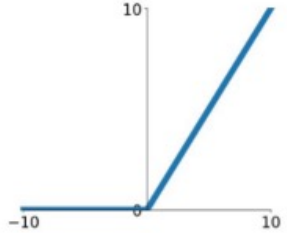
sigmoid:

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

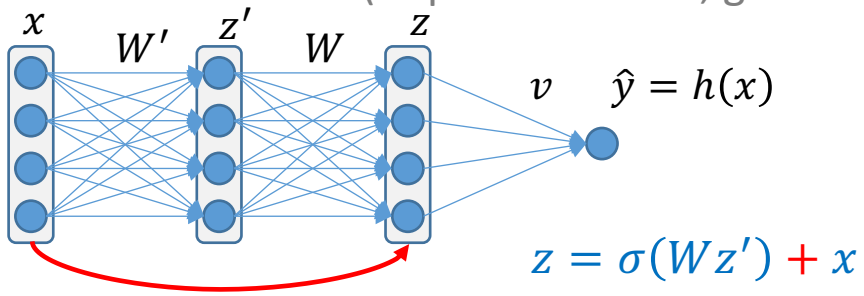


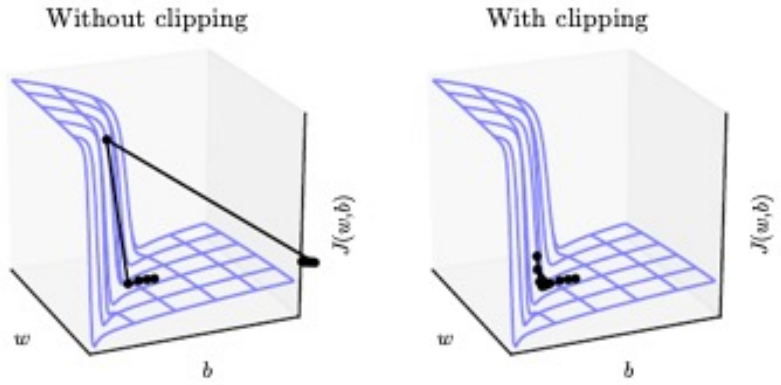
ReLU:

$$\sigma(x) = \max\{0, x\}$$



- Very deep networks suffer from vanishing or exploding gradients
- Reasons
  - Repeated applications of activations with small gradients (e.g., sigmoid)
  - Repeated applications of the same weight matrices
    Suppose W is used t times with decomposition $W = V\ \mathrm{diag}(\lambda)V^{-1}$, then gradients are scaled by $\mathrm{diag}(\lambda)^t$
    (Does not usually happen in simple feed-forward networks, common in other architectures)
- Solutions
  - Use different activations: ReLU solves vanishing gradients, but not exploding ones
  - Gradient clipping helps with exploding gradients: $if\ \|g\| > p: g \leftarrow \frac{gp}{\|g\|}$
  - Smart weight initialization also helps
  - Architectural tricks (skip connections, gated networks)





Optimization: reasons for optimism

- **Algorithmically, backprop consists of:**

1. **Gradient computations:**
   - used to be done tediously by hand
   - nowadays, completely abstracted away by autodiff (e.g., as in TensorFlow or PyTorch)
   - Provides many useful differentiable computational building blocks
2. **Matrix/tensor operations:** (lots and lots of them)
   - In principle, highly parallelizable
   - Luckily, are prevalent in computer graphics
   - Modern learning algorithms utilize **Graphical Processing Units (GPUs)**
   - Fast, parallelizable, and scalable (typically 5x-50x speedup)

- **Conclusion:** training neural nets has become practical and straightforward

TF playground demonstration

https://playground.tensorflow.org

Generalization (in theory)

- Modern neural networks are **big**. Should we expect them to generalize?
- **Theorem**: (won't prove)
  Let $\mathcal{H}_{NN}$ be class of sigmoidal-activation neural networks with N units and M connections (i.e., parameterized weights), then:

$$M^2 \leq VC(\mathcal{H}_{NN}) \leq N^2M^2$$

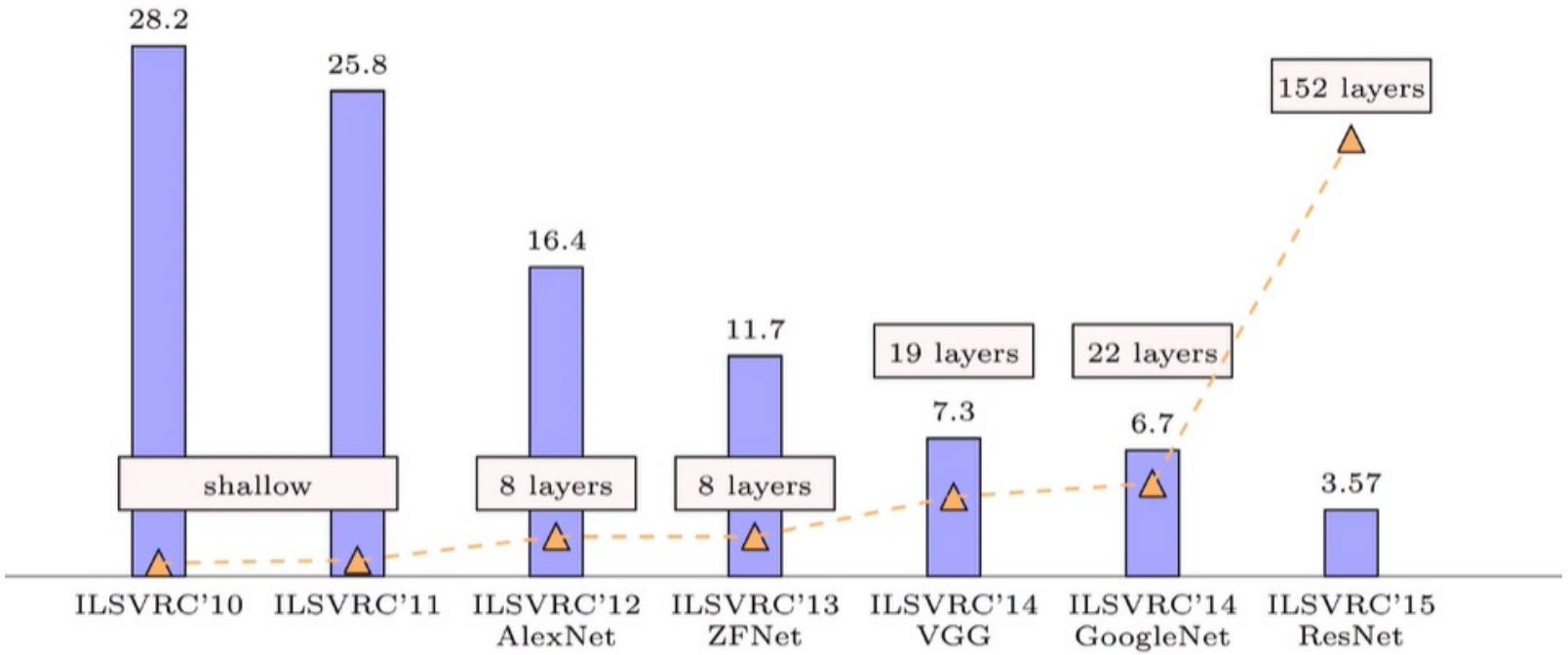
- VC is large, and grows quickly with network size
- (Interestingly, the bound is agnostic to width, depth, or connectivity structure; only depends on num. units and parameters!)
- Need to carefully control for overfitting

Controlling overfitting

- Can use explicit regularization (e.g., norm on weights), but not very popular
- Common *implicit* regularization tricks:
  1. **Early stopping** (using held-out validation set; like we saw)
  2. **Drop-out:**
     at each epoch (=SGD step), temporarily discard a random subset of weights
     (typical drop-out rate (percentage of discarded edges) values: 10%-50% - large!)
- **Intuition for drop-out**:
  - Prevents network from relying extensively on specific weights
  - Thus, prevents network from being tailored to training data (sort of…)
  - Can be thought of as **implicitly training an ensemble of network models**, each corresponding to a partial connectivity pattern (sub-graph)

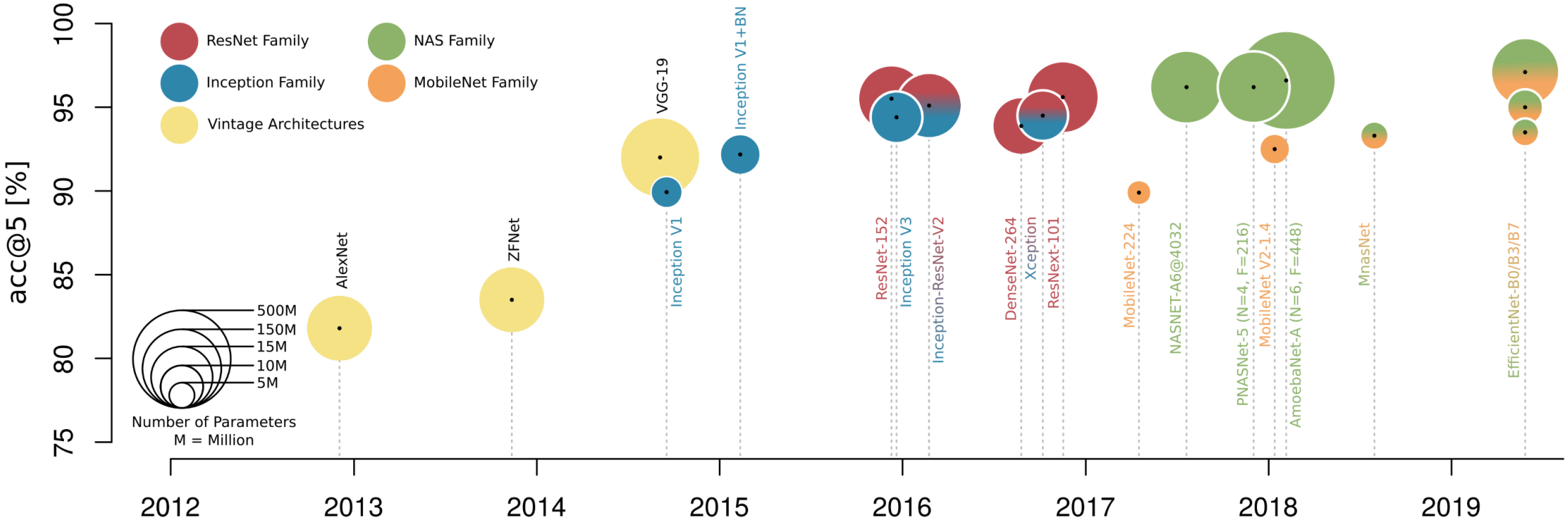
Generalization (in practice)

- VC says bad generalization, but empirically, even huge networks don't always overfit (this likely goes far beyond any effect attributed to intentional regularization)



Generalization (in practice)

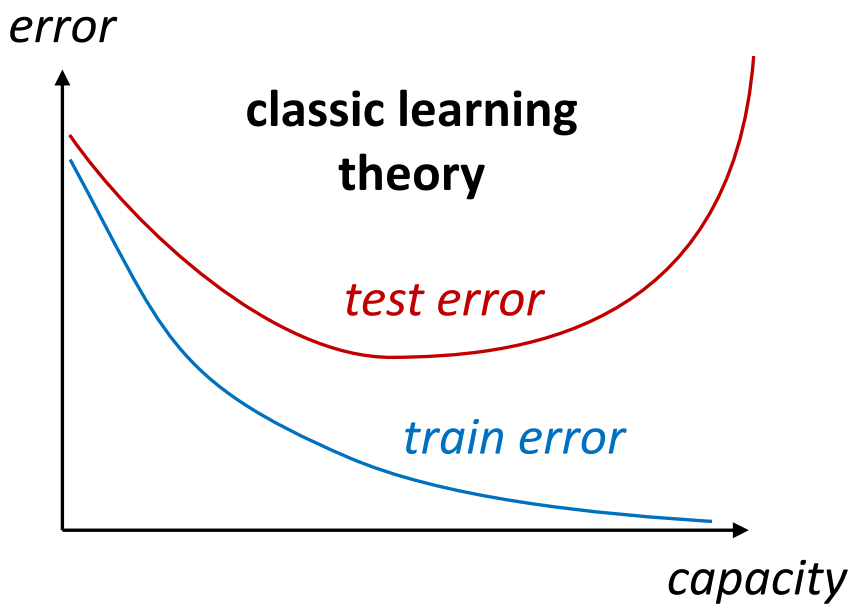
- VC says bad generalization, but empirically, even huge networks don't always overfit (this likely goes far beyond any effect attributed to intentional regularization)



- Surprising from the standpoint of classic statistical learning theory!

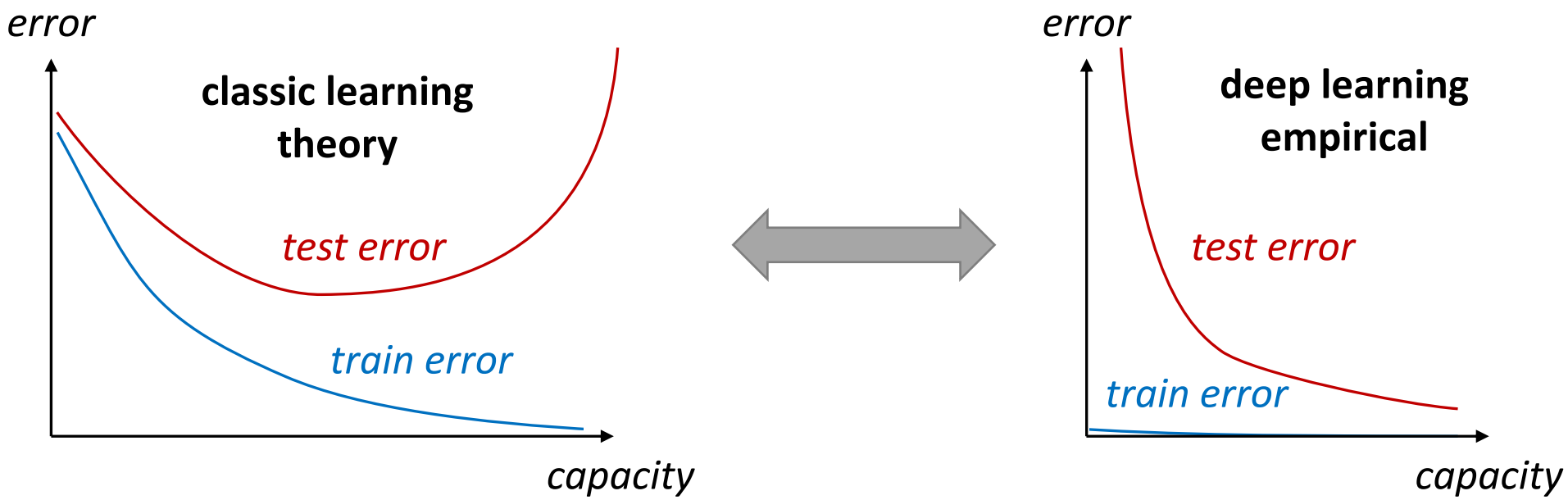
Theory! (?)

- Some research argues that the classic theory just doesn't apply
- (This is likely an overstatement, but also not completely wrong)
- **Example**: deep double-descent



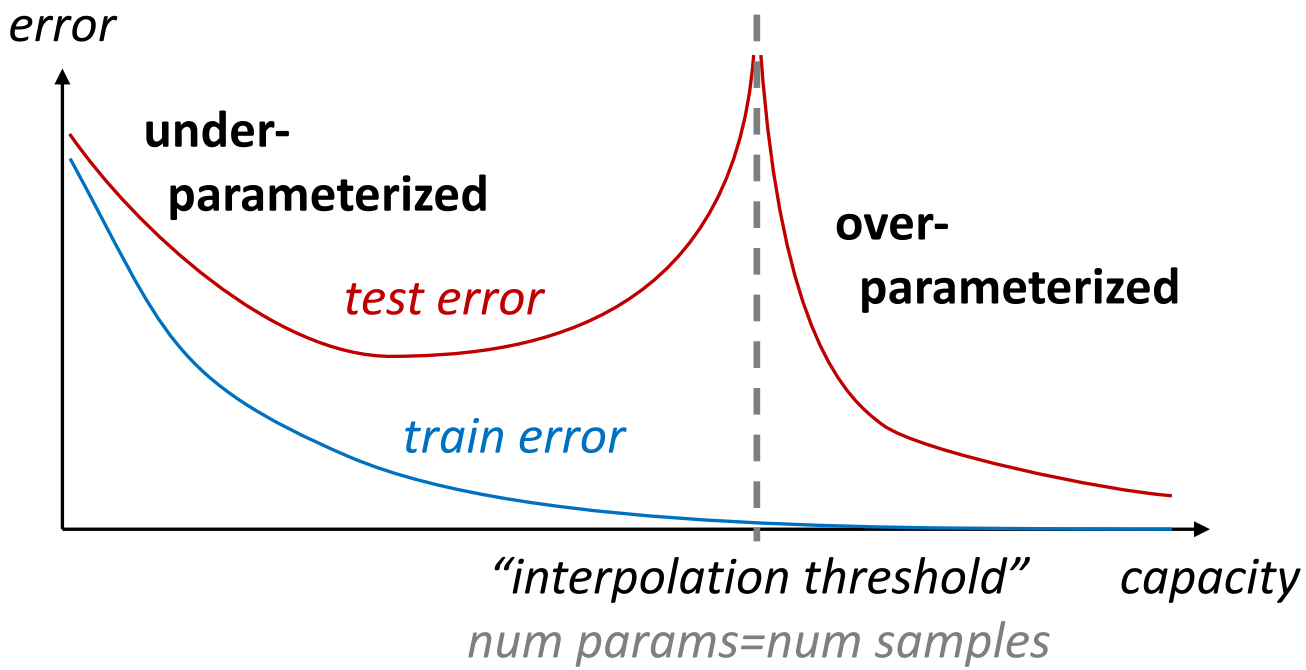
Theory! (?)

- Some research argues that the classic theory just doesn't apply
- (This is likely an overstatement, but also not completely wrong)
- **Example**: deep double-descent



Theory! (?)

- Some research argues that the classic theory just doesn't apply
- (This is likely an overstatement, but also not completely wrong)
- **Example**: deep double-descent



on the one hand:

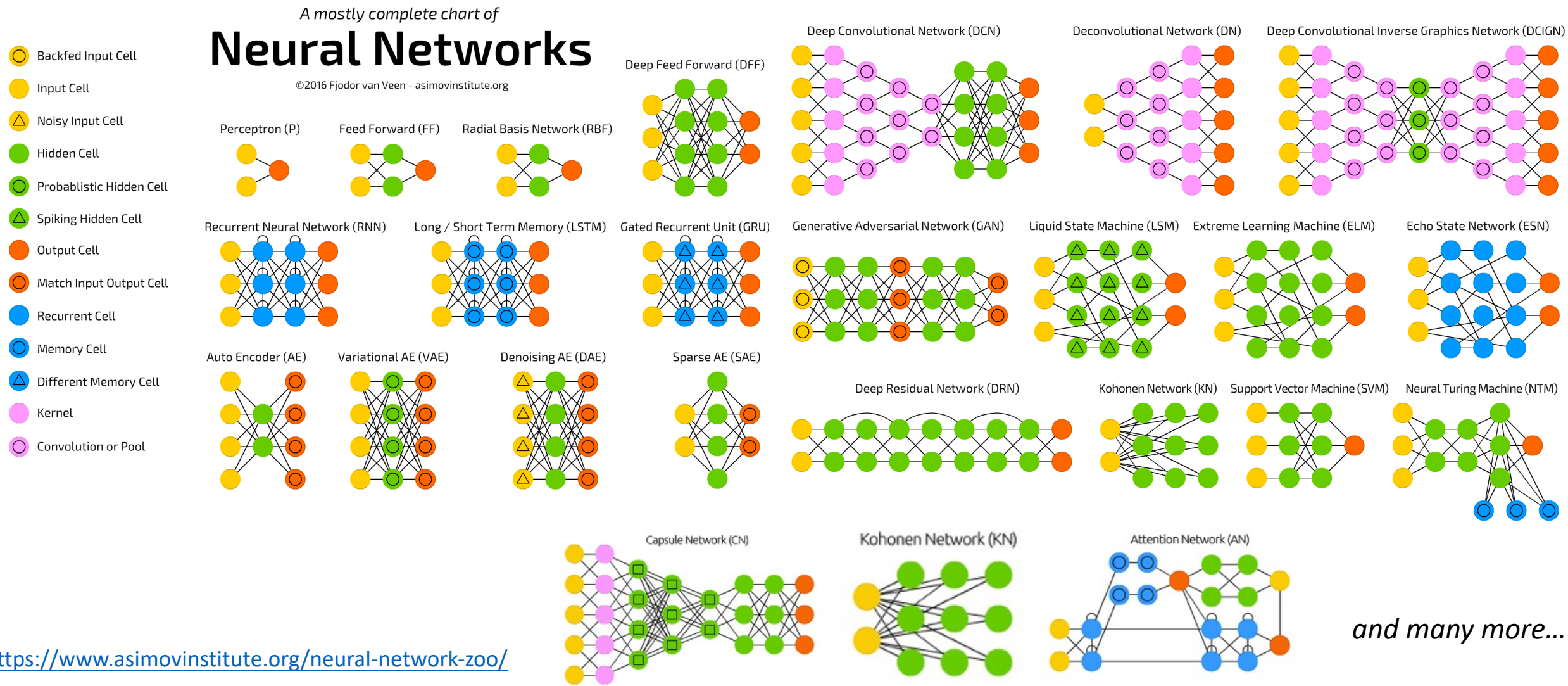
- *empirically observed*
- *theoretically established (for deep)*

on the other:

- *empirically disproved*
- *theoretically known (for linear!)*

Many ideas – but no unifying, agreed-upon, concrete theory (yet!)

Other Architectures



and many more…

https://www.asimovinstitute.org/neural-network-zoo/

Common Architectures

In the past (until ~5 years ago):

- **For 2D data** (images): convolutional neural networks (CNNs)
- **For sequential data** (text): recurrent neural networks (RNNs)

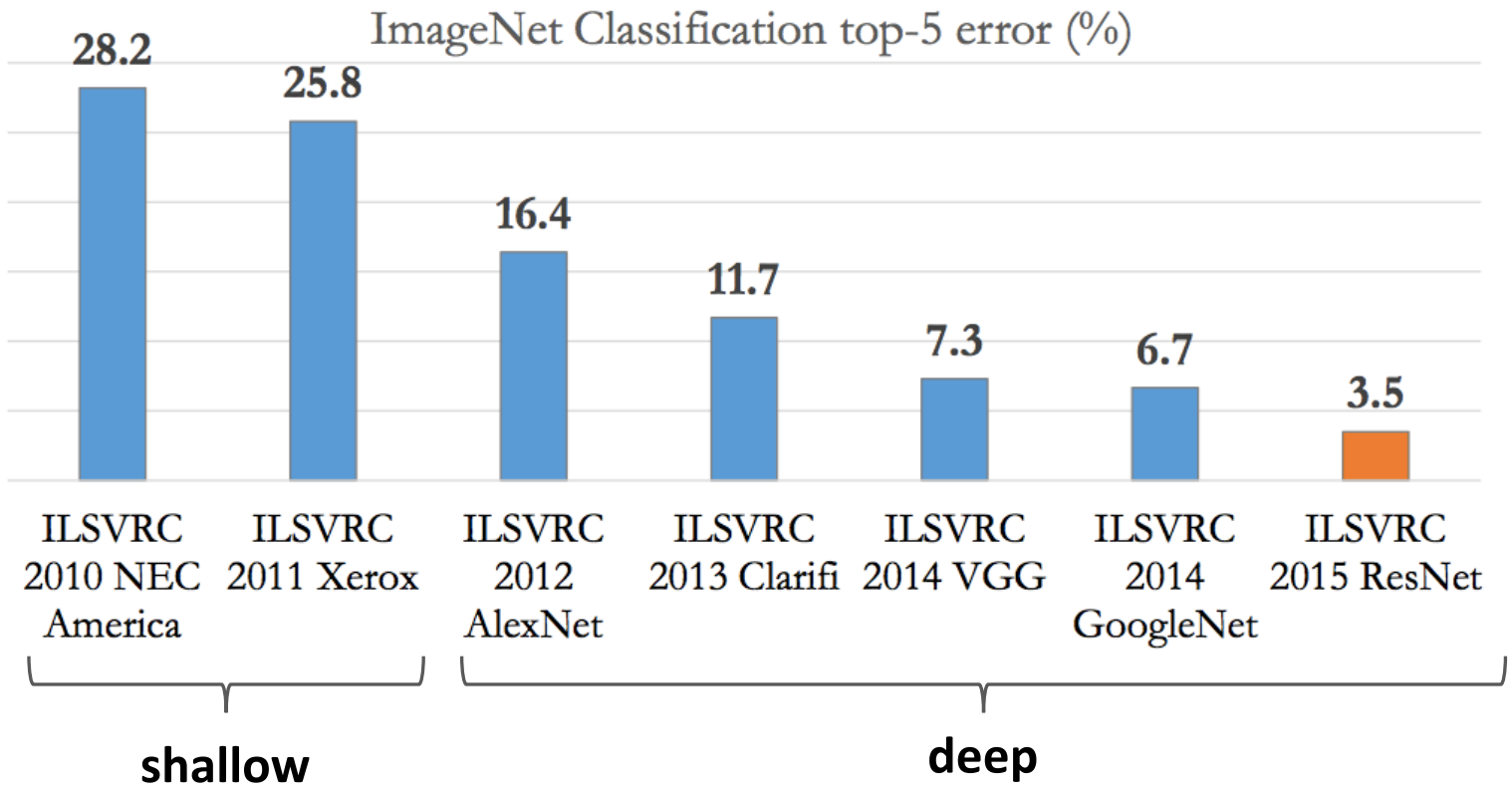
Nowadays:

- **For everything**: Transformers

Tomorrow?

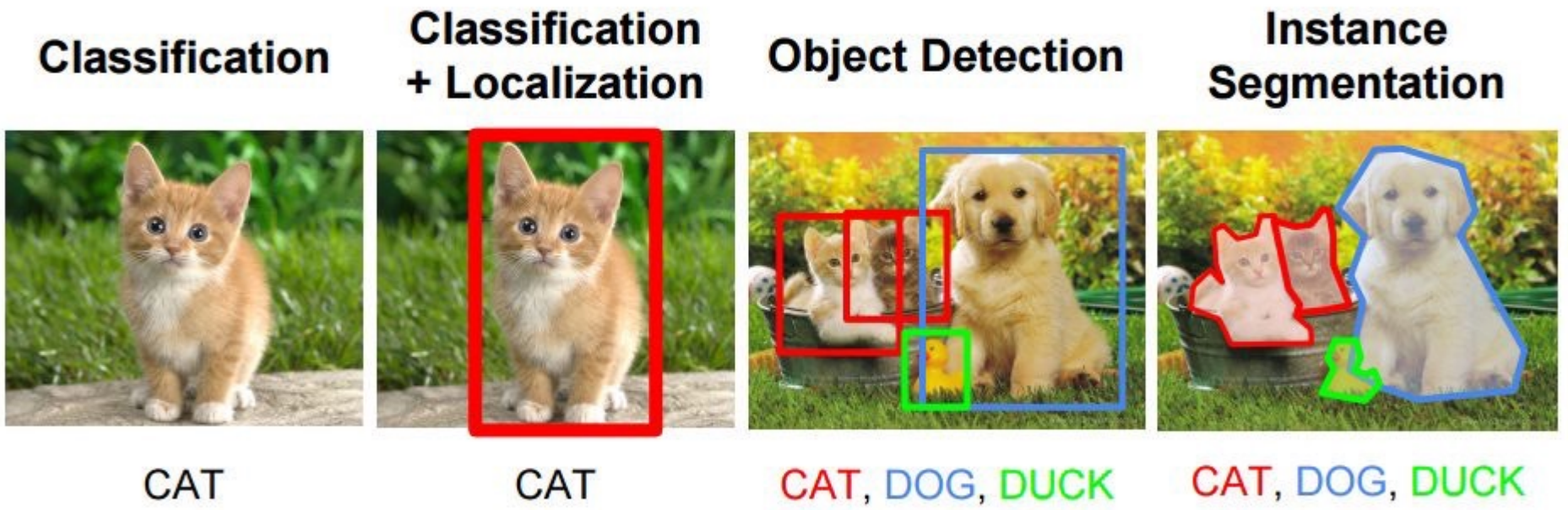
Neural nets for vision

- Modern neural networks are renowned for their performance on image recognition:



Neural nets for vision

- Modern neural networks are renowned for their performance on image recognition:



https://twitter.com/MikeTamir

Neural nets for vision

- Modern neural networks are renowned for their performance on image recognition:

- Let’s examine how they do this for **image recognition** (=classification)

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

Neural nets for image recognition

- **Input**: $x =$ natural image, small (128x128x3 pixels)
- **Output**: "cat" ($y = +1$) or "dog" ($y = -1$)

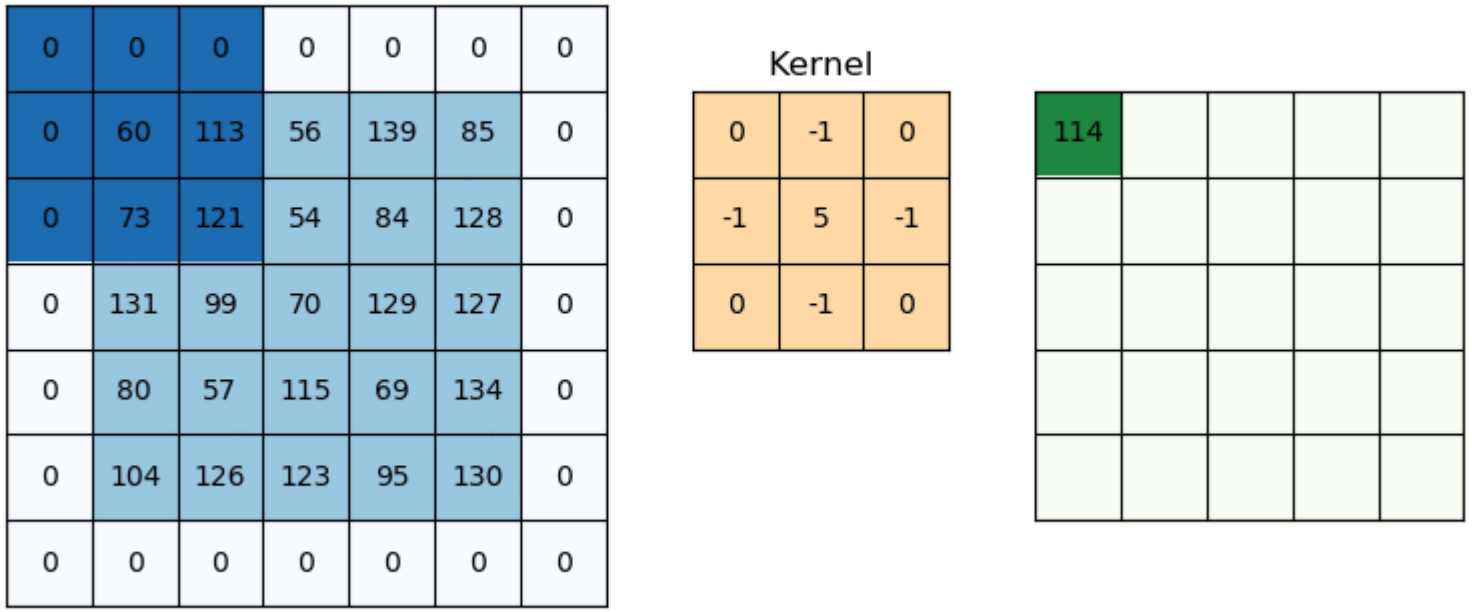
- Let's train a fully-connected neural network!
- **Problem**: $d = 49{,}152 \Rightarrow |W| = d^2 = 2{,}415{,}919{,}104$
- And that's just for <u>one layer</u>!
- Totally *infeasible*.

Neural nets for image recognition

- Fully connected networks view images as vectors
- But images have spatial structure (that vectors don't capture)
- **Example**: real world images are **translation invariant**: "moving" objects within an image does not change their essence
- This is a form of domain knowledge – useful to encode in a model
- Enter **Convolutional Neural Networks (CNNs)**

Convolutional Neural Networks

- **Idea**: apply same local “convolution” filter to all “patches”



Convolutional Neural Networks

- Each hXw patch/filter/kernel has $h * w$ parameters (e.g., 3*3=9)
- Typically use multiple (hundreds or more) of such filters in each layer
  - Here we see just one

0	0	0	0	0	0	0
0	60	113	56	139	85	0
0	73	121	54	84	128	0
0	131	99	70	129	127	0
0	80	57	115	69	134	0
0	104	126	123	95	130	0
0	0	0	0	0	0	0

Kernel

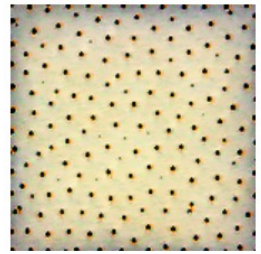
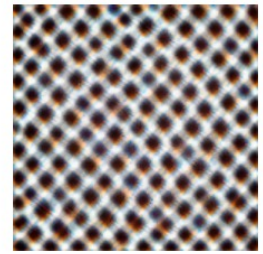
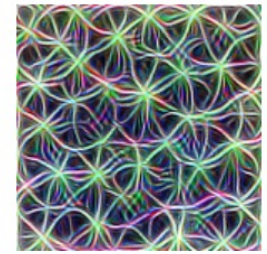
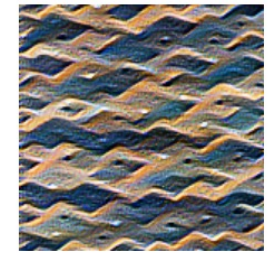
0	-1	0
-1	5	-1
0	-1	0

114				

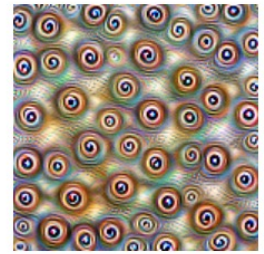
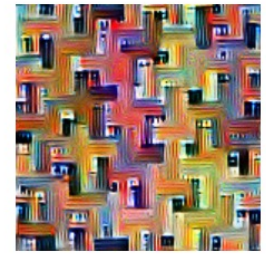
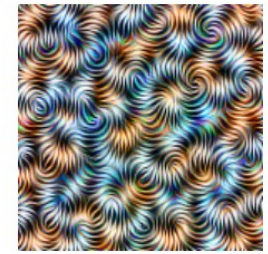
Convolutional Neural Networks

- **CNNs work well on natural images because they:**
  - ➢ use a fraction of possible connections (by applying local filters)
  - ➢ drastically reduce the number of learnable parameters (by weight sharing/tying)
  - ➢ do so in a way that preserves input structure and invariance
- **Thus, CNNs can:**
  - enjoy the benefits of deep models (multiple layers, non-linearities, etc)
  - but using a manageable number of parameters
- Some studies suggest that different filters have different meanings!

Layer 3a

Layer 3b

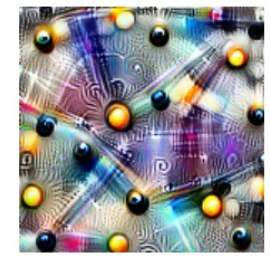





Layer 4b

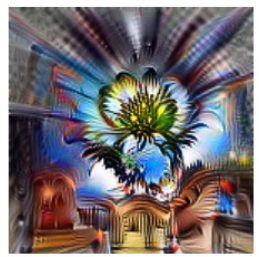

Architecture


Fluffy rope


Trees


Billiard balls

Layer 4c


Palm trees


Wheels


Dogs on leash


Houses

Layer 5a


Candles


Balls


Brass instruments


Traffic lights

https://distill.pub/2017/feature-visualization/appendix/

Neural nets for language

- Challenges
  - Input is discrete: how to feed a text into a neural network?
  - Output is (sometimes) discrete: how to make a neural net generate text?
  - Input (and output) is not only discrete, it has structure
    - “The dog chased the man” vs. “The man chased the dog”
    - “The cat sat on the mat” vs. “The dog sat on the mat”

- We’ll show here a **small** sample of possible solutions

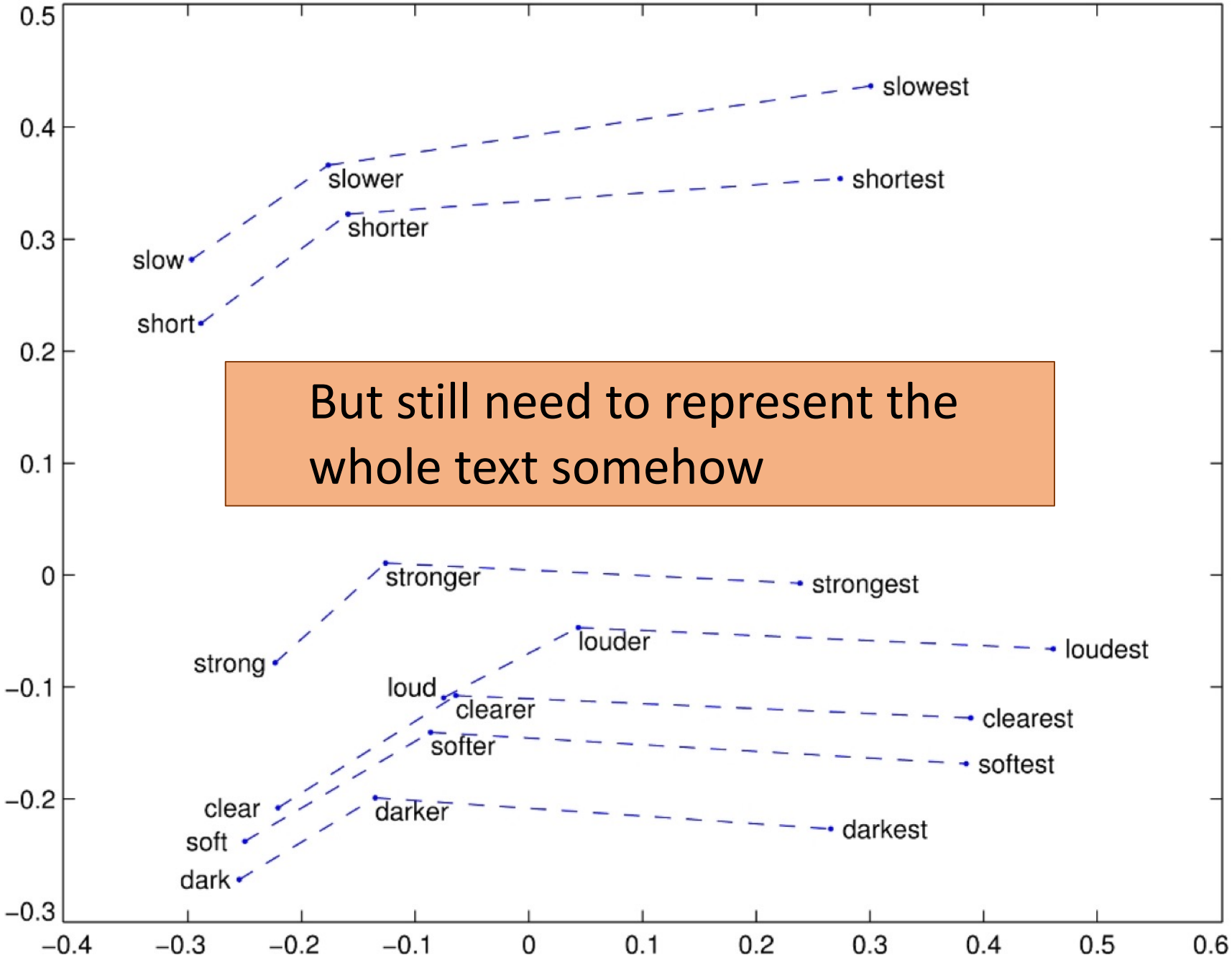
Modeling text

"The cat sat on the mat"

- How could we represent a text in vector space?
  - We've seen the bag-of-words representation last week

$$x_T = [0 \dots 0 \dots 0\ 1\ 0 \dots 0\ 1\ 0 \dots 0\ 2\ 0 \dots 0] \in \mathbb{R}^{|V|}$$

  - The number of possible such examples is exponential in $|V|$, so estimating probabilities based on text counts is intractable
    - Naïve Bayes provided one solution to this problem
- Idea: words can be "embedded" in vector space
  - Each word u in the vocabulary would have a vector $e_u \in \mathbb{R}^d$, where $\mathrm{d} \ll |V|$
  - We've descreased the dimensionality
  - And this has other benefits: e_{dog} and e_{cat} can be close
  - Think of $\{e_u\}$ as more parameters to be learned by a neural network (via backpropagation and gradient descent)



GloVe embeddings

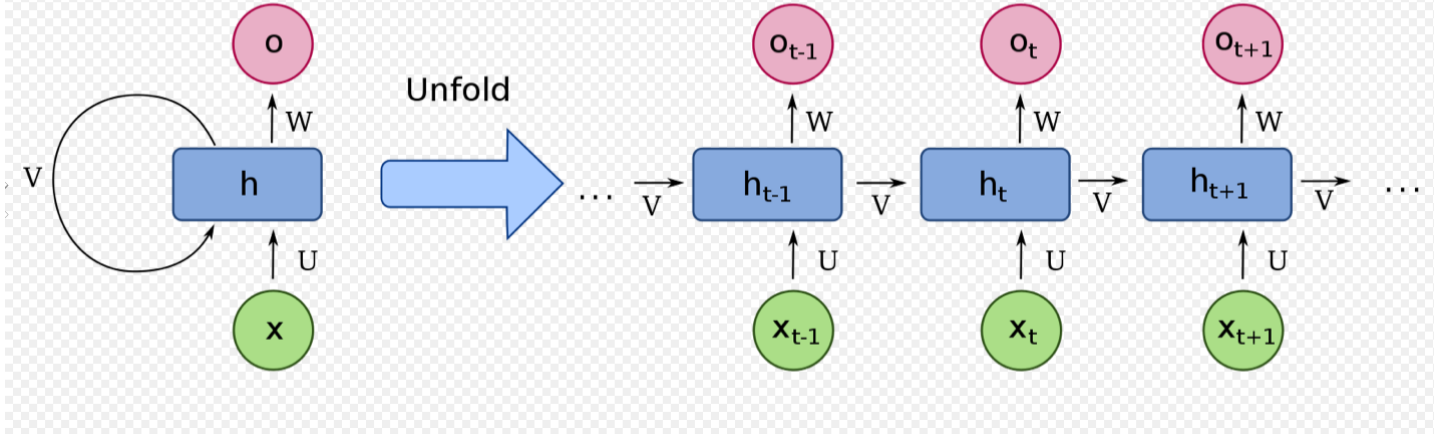
Modeling text

"The cat sat on the mat"

- Now a text $u_1, u_2, \ldots, u_T$ is a sequence of word vectors $e_{u_1}, e_{u_2}, \ldots, e_{u_T}$
  - These need to be input to some model
- Possibilities for aggregating them
  - Concatenate
    - Problem: invariable size of input, may grow substantially for long texts
  - Average
    - Problem: ignores structure (word order, word importance)
    - But for many tasks is a pretty good baseline
  - What else?

Recurrent neural networks

- **Key idea**: traverse the text from start to end and maintain a "state" of the representation up to time t
- Computation at time t: $\boldsymbol{h}_t = \sigma(U\boldsymbol{x}_t + V\boldsymbol{h}_{t-1})$
  - $\boldsymbol{h}_t$ is the hidden state (different from the function h_t mentioned earlier)
  - Here x_t is e_{u_t} , namely, the vector of the t-th word
  - We can classify by adding output layer $\check{y} = \sigma(\boldsymbol{v}\boldsymbol{h}_T)$ or $\check{y}_t = \sigma(\boldsymbol{v}\boldsymbol{h}_t)$
  - U, V, and $\boldsymbol{v}$ are parameters to learn with backpropagation and SGD



[Figure: Wikipedia]

Recurrent neural networks

- Advantanges
  - Can be applied to arbitrarily long texts
  - Can capture word order and structure
  - Reasonable number of parameters
  - Led to many breakthroughs in NLP
- Shortcomings
  - Vanishing and exploding gradients make optimization tricky
  - Hard to stack multiple layers
  - Difficult to parallelize the computation
  - Some of these are mitigated by RNN variants like LSTM

Transformers (very rough presentation)

- Consider the average representation

"The dog chased the man" ⟶ $\frac{1}{5}(v_{The} + v_{dog} + v_{chasee} + v_{the} + v_{man}) = \frac{1}{T}\sum_i v_i$

- Problems:
  1. Every word has equal importance
  2. Order-invariant
- Solutions
  1. Word-level weights: $\frac{1}{T}\sum_i \alpha_i v_i$
  2. Position encoding: $\frac{1}{T}\sum_i \alpha_i (v_i + p_i)$

α dynamic score for word *i*

p_i vector for position *i*

Transformers

- Advantages
  - Efficiently parallelizable
  - Optimization works much better than RNNs
  - Lead to superior results in many cases
  - Work well for many domains, including vision

Is Attention All You Need?

Current Status: Yes

Time Remaining: 1005d 22h 41m 13s

Proposition:

On January 1, 2027, a Transformer-like model will continue to hold the state-of-the-art position in most benchmarked tasks in natural language processing.

For the Motion

Jonathan Frankle
@jefrankle
Harvard Professor
Chief Scientist Mosaic ML

Against the Motion

Sasha Rush
@srush_nlp
Cornell Professor
Research Scientist Hugging Face 🤗

Discussion

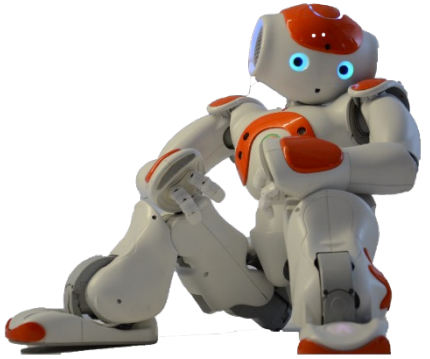
Discussion

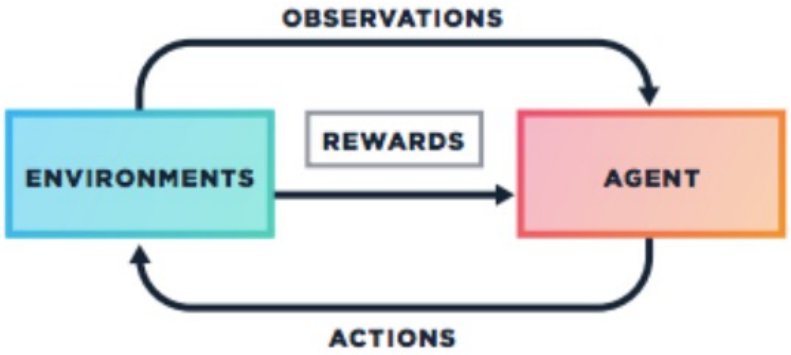
- Deep learning is an old idea that took time to become practically useful
- **Reasons why it works well now** (on some tasks):
  - compute power (GPU)
  - more machines (parallel)
  - tons of data (linear scaling)
  - algorithms (backprop)
  - autodiff
  - modeling flexibility (autodiff)
  - modeling freedom (building blocks)
  - socially acceptable
  - file-drawer effect
  - Standard libraries (PyTorch, TensorFlow, …)
- So “new electricity” or “fancy logistic regression”?
- **Suggestion**: just make the best of it!

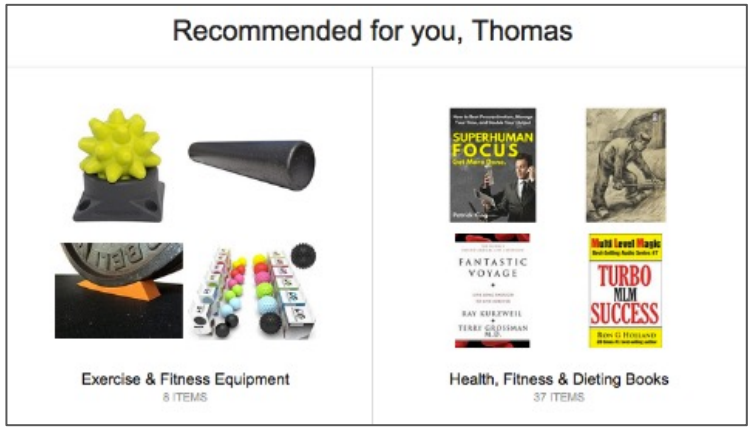
Discussion and course summary

Ending on a positive note

- Thankfully, lots of research on how to learn properly in these challenging settings
- But more generally, and as we've noted, *learning is not all about prediction*
- In fact, certain branches of machine learning are *all about decisions*
- **Key example**:
  reinforcement learning (way outside of our scope)







Our course in retrospect

- **Part I:** *supervised binary classification*
  - Introduction
  - Methods
  - SVM in depth
- **Part II:** *aspects of learning*
  - Statistical: generalization
  - Modeling: model selection, validation
  - Optimization: gradient descent
  - Practical aspects of learning
- **Part III:** *more supervised learning*
  - Regression
  - Bagging and boosting
  - Generative models
  - Deep learning

types of learning

input:
feature vectors
images
text
speech
behavior
graphs
time-series
…

output:
binary
multiclass
multilabel
structured
scalars
intervals
distributions
…

supervision:
supervised
unsupervised
semi-supervised
weakly supervised
zero-shot
…

approach:
discriminative
generative
hybrid
…

environment:
batch learning
online learning
reinforcement learning
active learning
metric learning
multi-task learning
learning to teach
meta-learning
…

types of learning

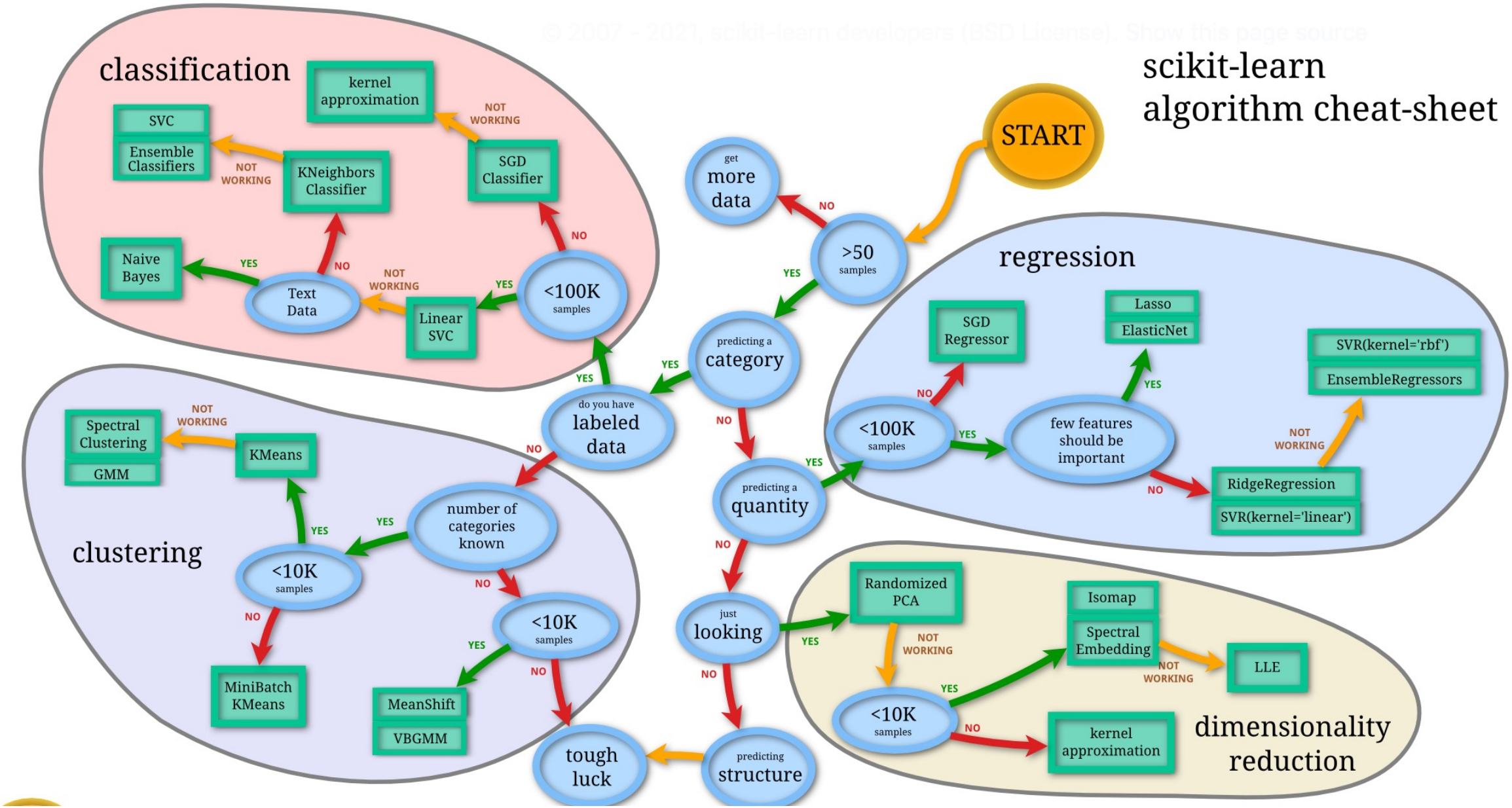
input:
feature vectors
images
text
speech
behavior
graphs
time-series
…

output:
binary
multiclass
multilabel
structured
scalars
intervals
distributions
…

supervision:
supervised
unsupervised
semi-supervised
weakly supervised
zero-shot
…

approach:
discriminative
generative
hybrid
…

environment:
batch learning
online learning
reinforcement learning
active learning
metric learning
multi-task learning
learning to teach
meta-learning
…

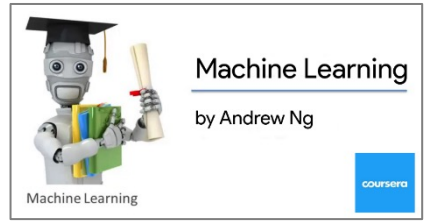


The scikit-learn cheat cheat:
https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

Related and advanced courses

- **קורסי בסיס בפקולטה:**
  - מבוא לבינה מלאכותית
  - מבוא לאופטימיזציה
  - מבוא לסטטיסטיקה
  - ...
- **קורסים מתקדמים בפקולטה:**
  - חיזוי והסקה סטטיסטית
  - למידה עמוקה ושימושיה
  - למידה עמוקה על מאיצים חישוביים
  - מבוא לעיבוד שפות טבעיות
  - מערכות לומדות והתנהגות אנושית
  - מבוא לטרנספורמרים
  - ...
- **קורסים מחוץ לפקולטה:**
  - תכנון ולמידה מחיזוקים (RL)
  - עיבוד וניתוח מידע (unsupervised)
  - מבוא להסקה סיבתית
  - מערכות סוכנים חכמים
  - …

Recap


=


+


- **Recall**: one of our goals for the course was to give you an "edge" in machine learning
- Made up, roughly correct statistic:
  - ML is 90% predictive, discriminative, supervised learning
  - 10% everything else (e.g., things we saw today)
- **The edge lies in the 10% –**
  but succeeding there requires a fundamental understanding of the core 90%
- This means *theory* and *practice* (=in-depth, not-necessarily-coding-wise practice)
- **Key take-away**: remember that

  machine learning = **handling uncertainty** using **modeling** + **statistics** + **optimization**

  (and that they are tightly and inseparably interconnected)
- Now you are ready!

Good luck!