GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Deep Learning

Lecture 5: Multi-layer perceptron

Alexander Ecker
Institut für Informatik, Uni Göttingen

alxecker

https://alexanderecker.wordpress.com

– Credit: some of the slides based on Ian Goodfellow's slides –

Agenda for this week

Motivation for deep learning

Perceptron

Multi-layer perceptron

Stochastic gradient descent

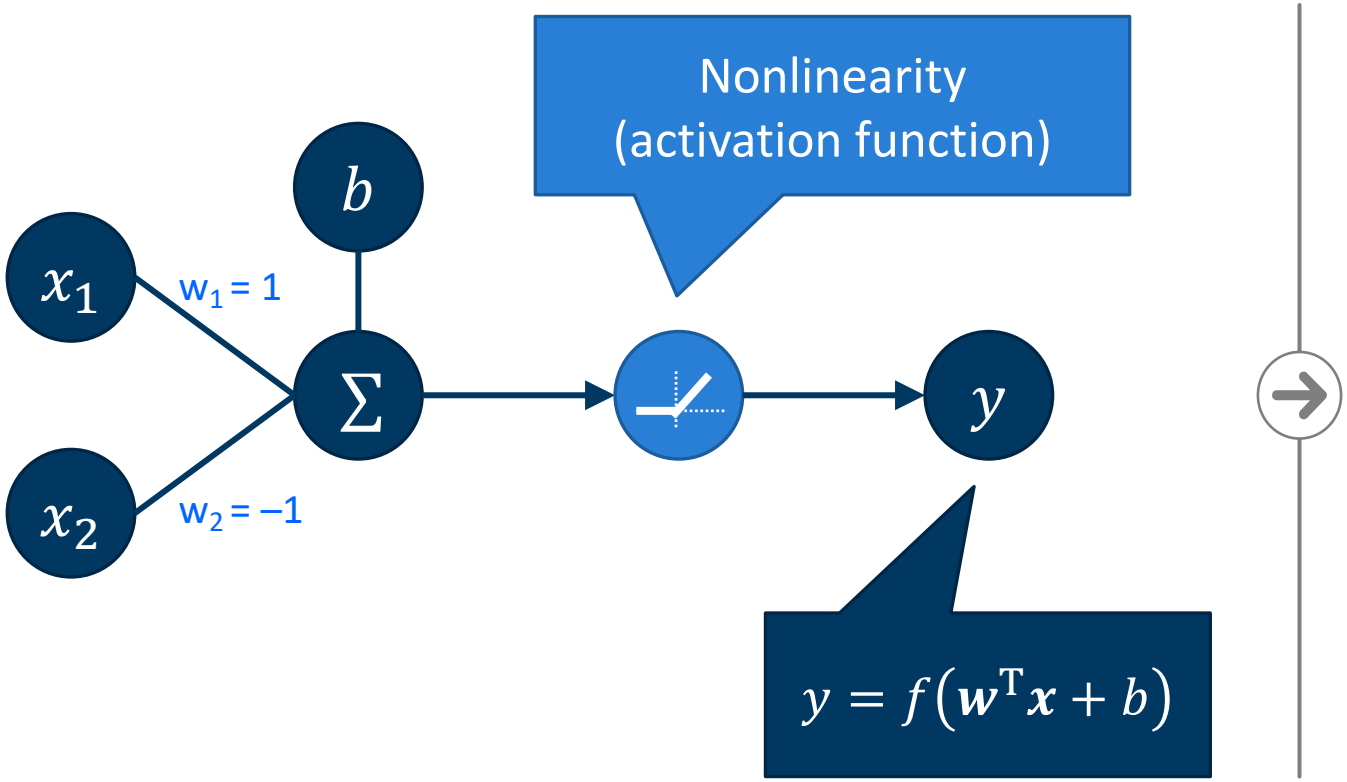
Regularization of deep nets

Architecture choices

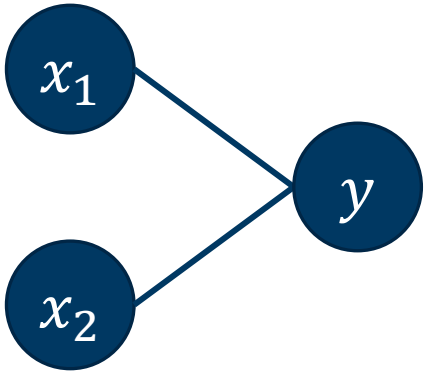
Deep learning frameworks

Architecture choices

Perceptron / MLP hidden units



Simplified representation



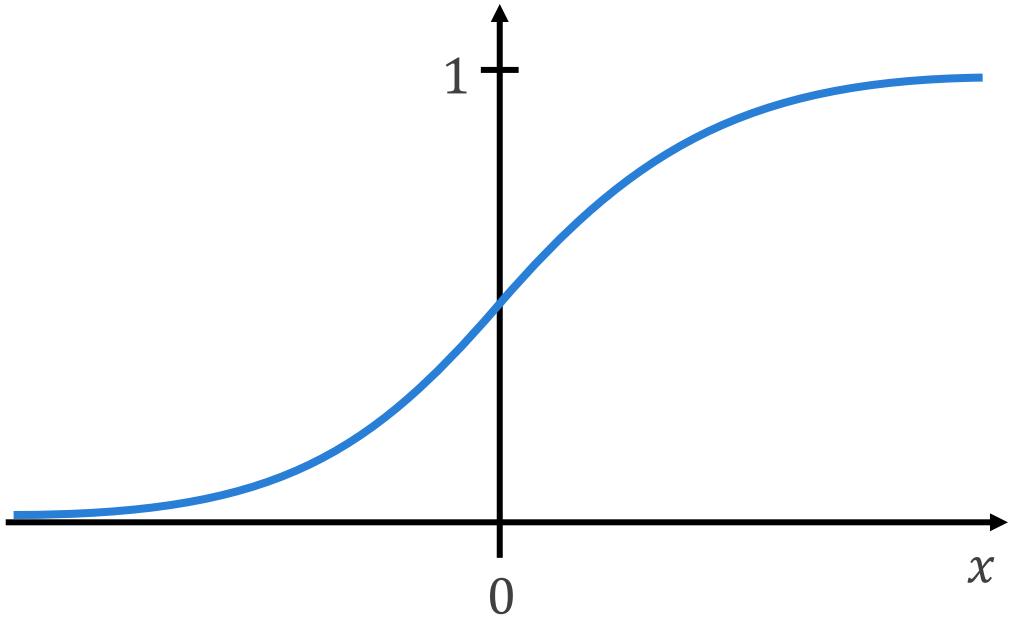
Linear combination and nonlinear activation function are implicit in this diagram

Sigmoids

Activation functions

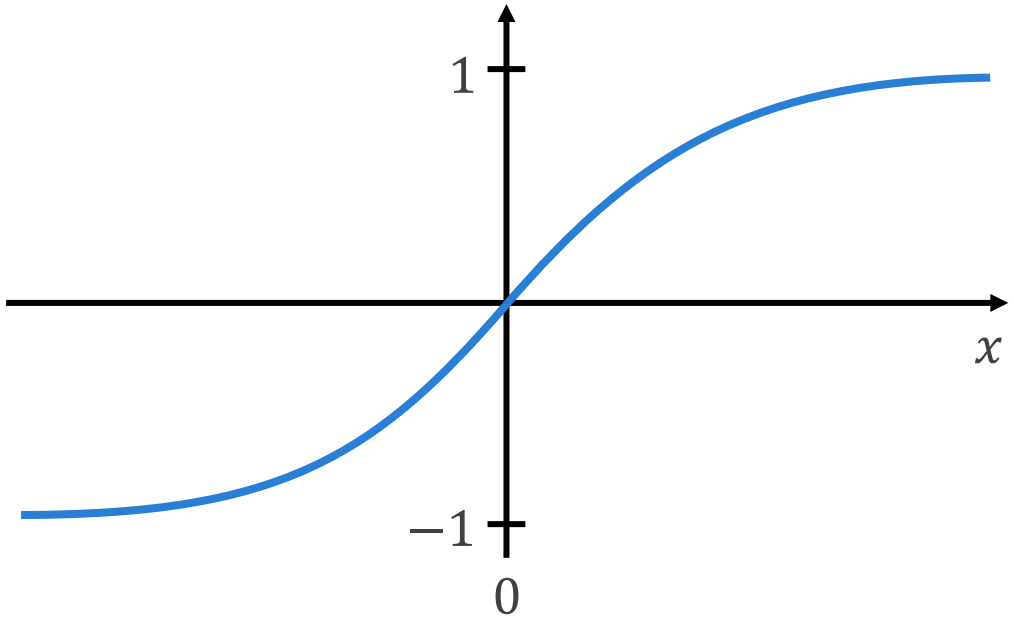
Logistic function

$$f(x) = \frac{1}{1 + \exp(-x)}$$

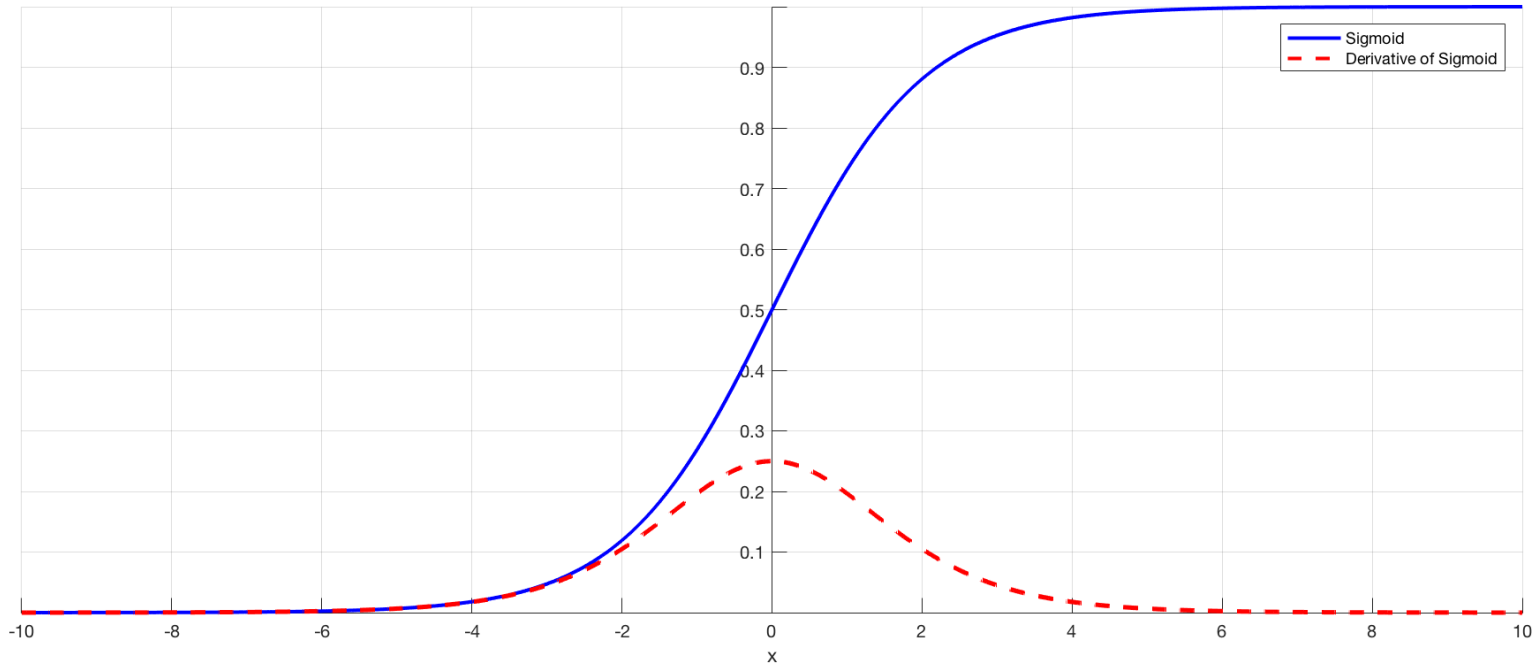


Hyperbolic tangent (tanh)

$$f(x) = \tanh(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1}$$

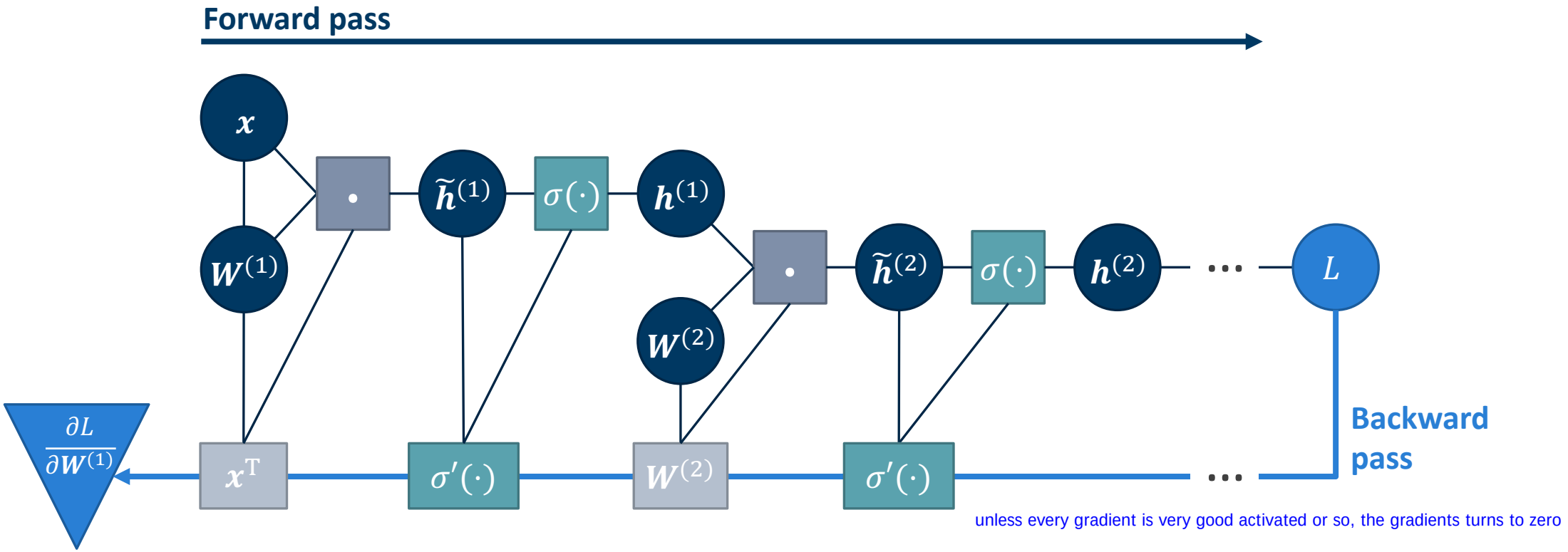


Sigmoid activation functions: vanishing gradients



Gradient close to zero for large positive or negative inputs

Stacking multiple layers with sigmoid activations



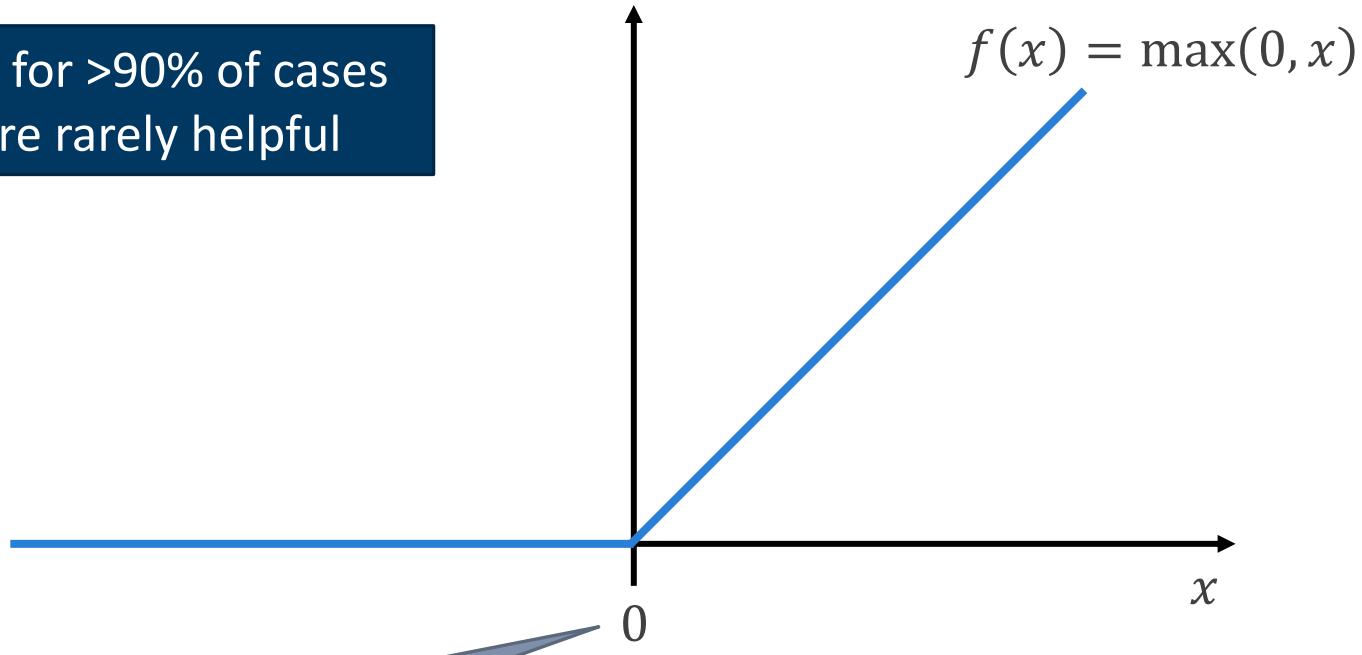
For deep networks: overall gradient very likely to be close to zero

Rectified linear unit (ReLU)

another activation function than sigmoid

very simple

Standard choice for >90% of cases
Alternatives are rarely helpful



No gradient at zero
→ not an issue

the chance to get zero zero is very small, so it is no big issue

Jarrett, Kavukcuoglu, Ranzato, LeCun, *ICCV 2009*
Nair & Hinton, *ICML* 2010

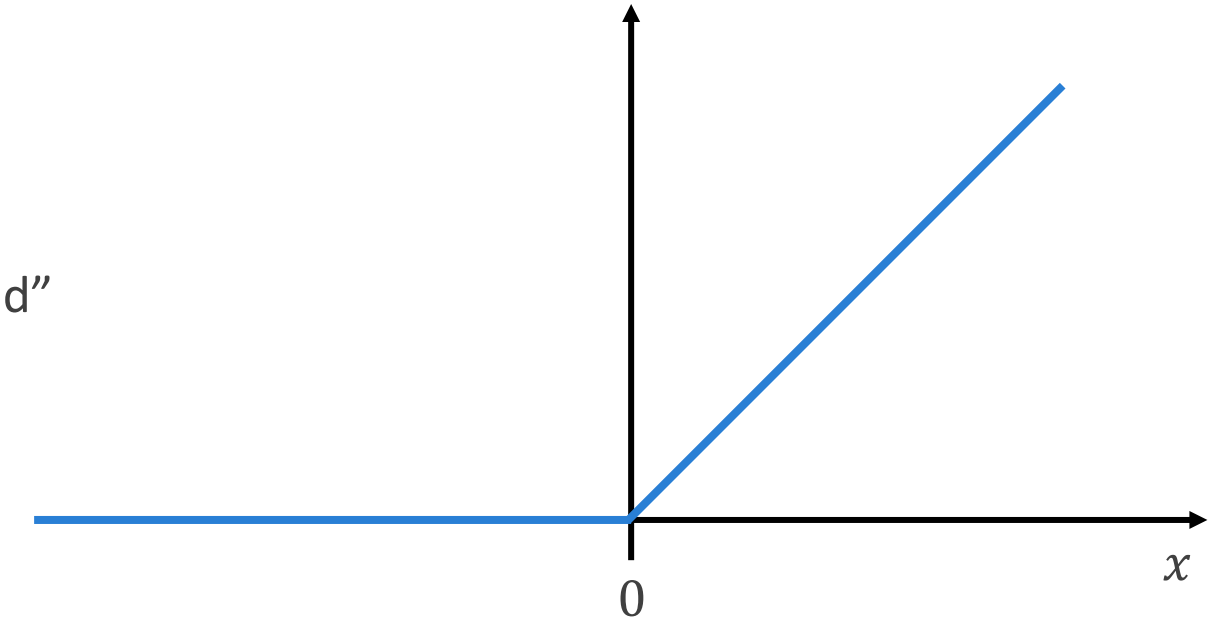
ReLU fixes vanishing gradient issue (somewhat)

Deep ReLU networks are locally linear
→ gradients well behaved

“Dead ReLU” problem:
pre-ReLU input zero for all network inputs → no gradients & unit is “dead”

Advantage:
- fixed and do not miss up with other gradients
- gradient is actuall zero, if it is not activated
- very well behaviour when it is active, is linear

Disadvantage:
- if everything is never be activated, because all the inputs are all zero, but in practise it does not real matter



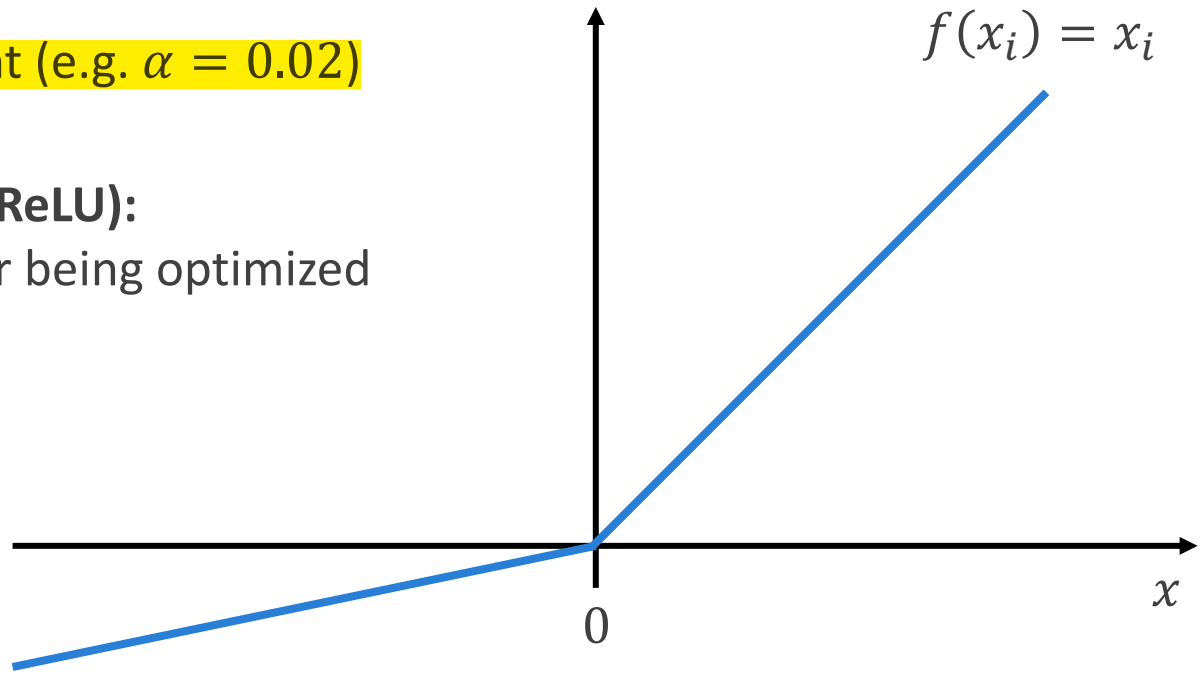
Leaky ReLU / Parametric ReLU

Leaky ReLU:

$\alpha_i = \alpha < 1$ constant (e.g. $\alpha = 0.02$)

Parametric ReLU (PReLU):

α_i model parameter being optimized



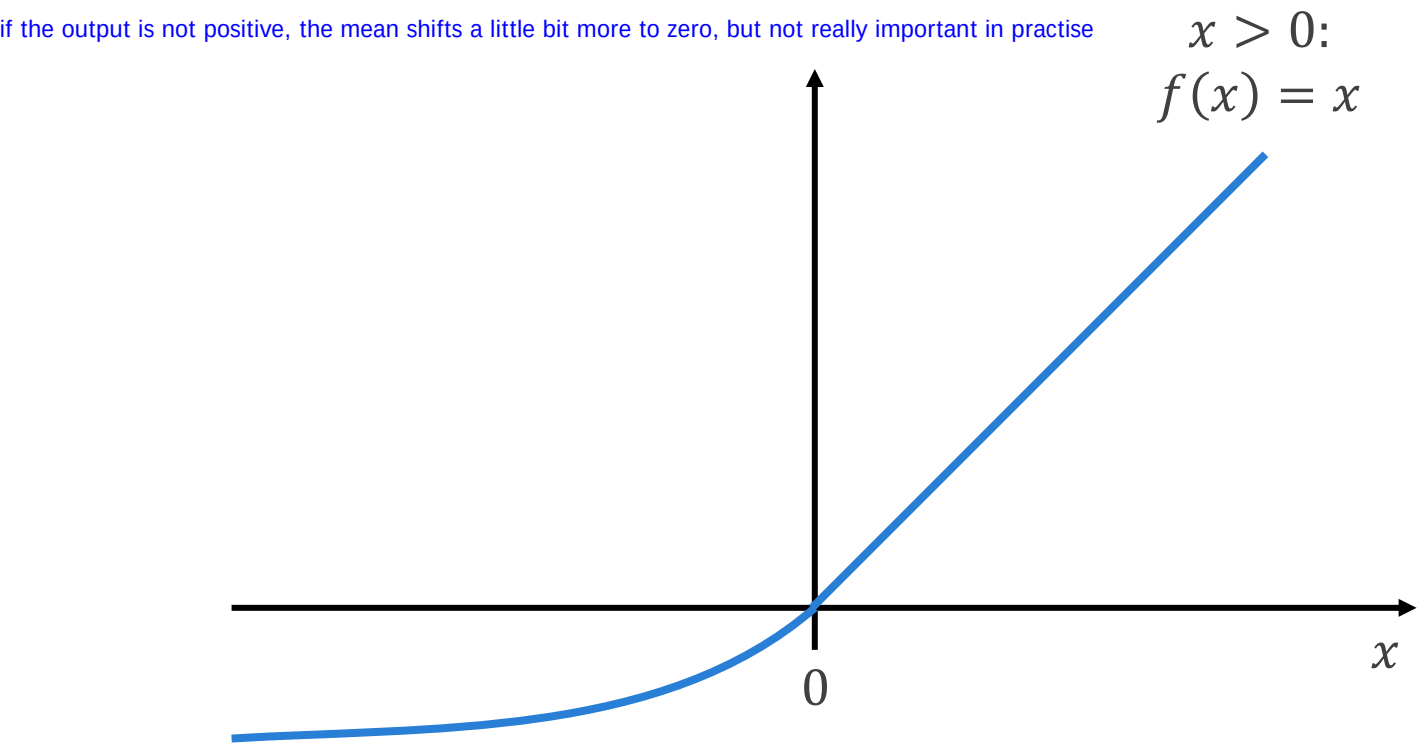
still have a slope for negative inputs :)

Fixes “dead ReLU” problem

Exponential linear unit (eLU)

a littele bit like the one before, but not linear

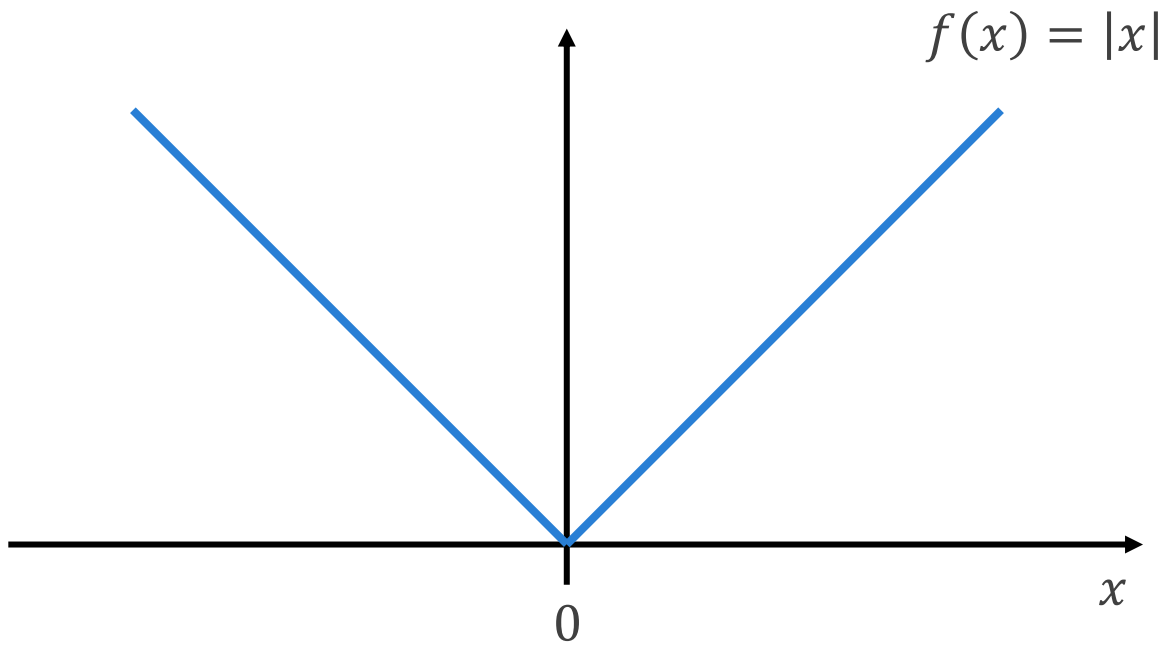
Advantage: if the output is not positive, the mean shifts a little bit more to zero, but not really important in practise



Clevert, Unterthiner, Hochreiter, *ICLR* 2016

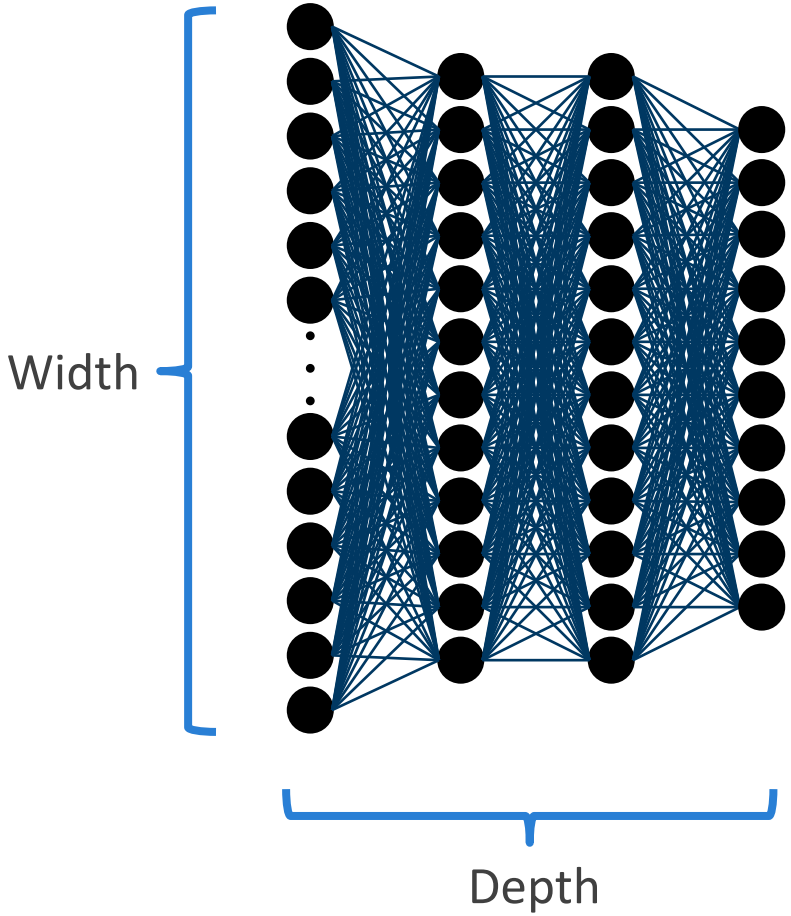
Absolute value activation

Theoretically nice, but not used in practise



Not really used in practice as it can be built from two ReLU units with inverted weights

Architecture basics



limitations: bounded inputs are needed

Universal Approximator Theorem for one-hidden-layer neural networks

$\phi : \mathbb{R} \to \mathbb{R}$ continuous, bounded, non-constant activation function ReLu is not bounded

it does not have to be 0 and 1, we can change/ shift it

continuous!

$f: [0,1]^M \to \mathbb{R}$ arbitrary function on unit hypercube

$F(\boldsymbol{x}) = \sum_{i=1}^{N} v_i \, \phi\left(\boldsymbol{w}_i^{\mathrm{T}} \boldsymbol{x} + b_i\right)$ one-hidden-layer neural network of width *N*

Theorem

For any function $f \in [0,1]^M \to \mathbb{R}$ and any $\epsilon > 0$
there exist
a finite integer *N*, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $\boldsymbol{w}_i$
such that
$|F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon$

Universal Approximator Theorem in words

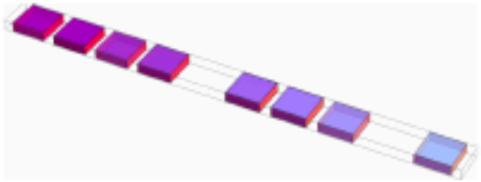
One hidden layer is enough to approximate any function $f \in [0,1]^N \rightarrow \mathbb{R}$ to an arbitrary degree of accuracy

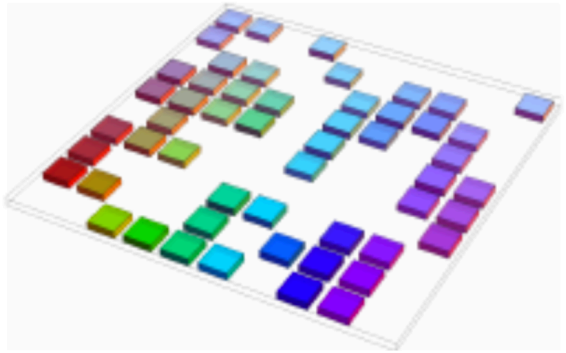
how well is it generalised?

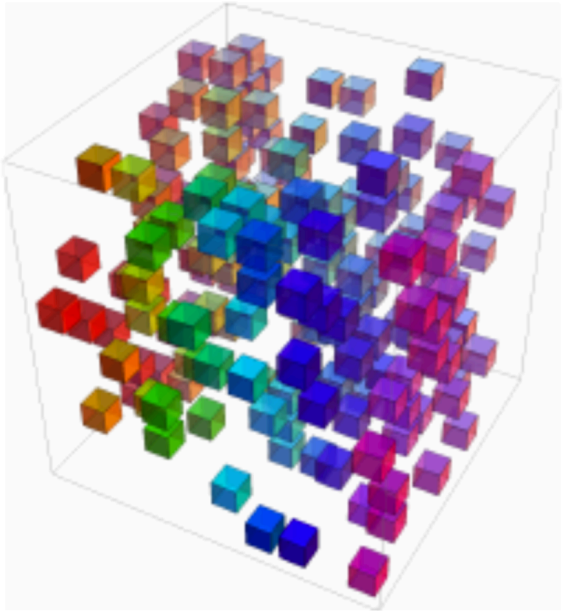
So why deeper?

- Able to *represent* ≠ able to *learn*
- Shallow net may need exponentially more width
- Shallow net may overfit more

Curse of Dimensionality







...

1D → $N = 10$

2D → $N = 100$

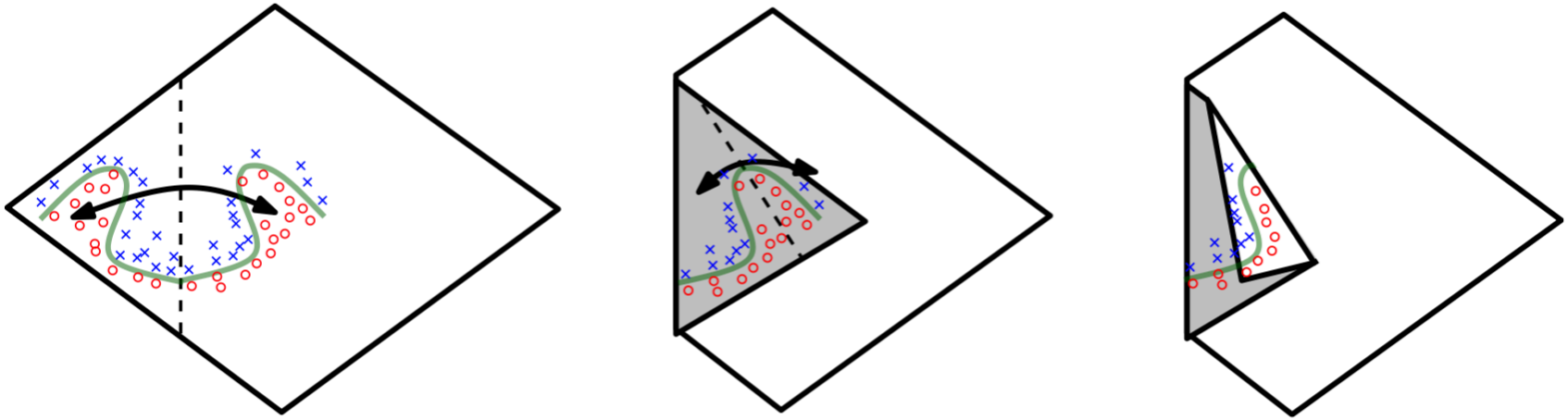
3D → $N = 1000$

k-D → $N = 10^k$

Exponential representation: advantage of depth

ReLu can be used as an activation function for all

Absolute value activation function "folds" the space

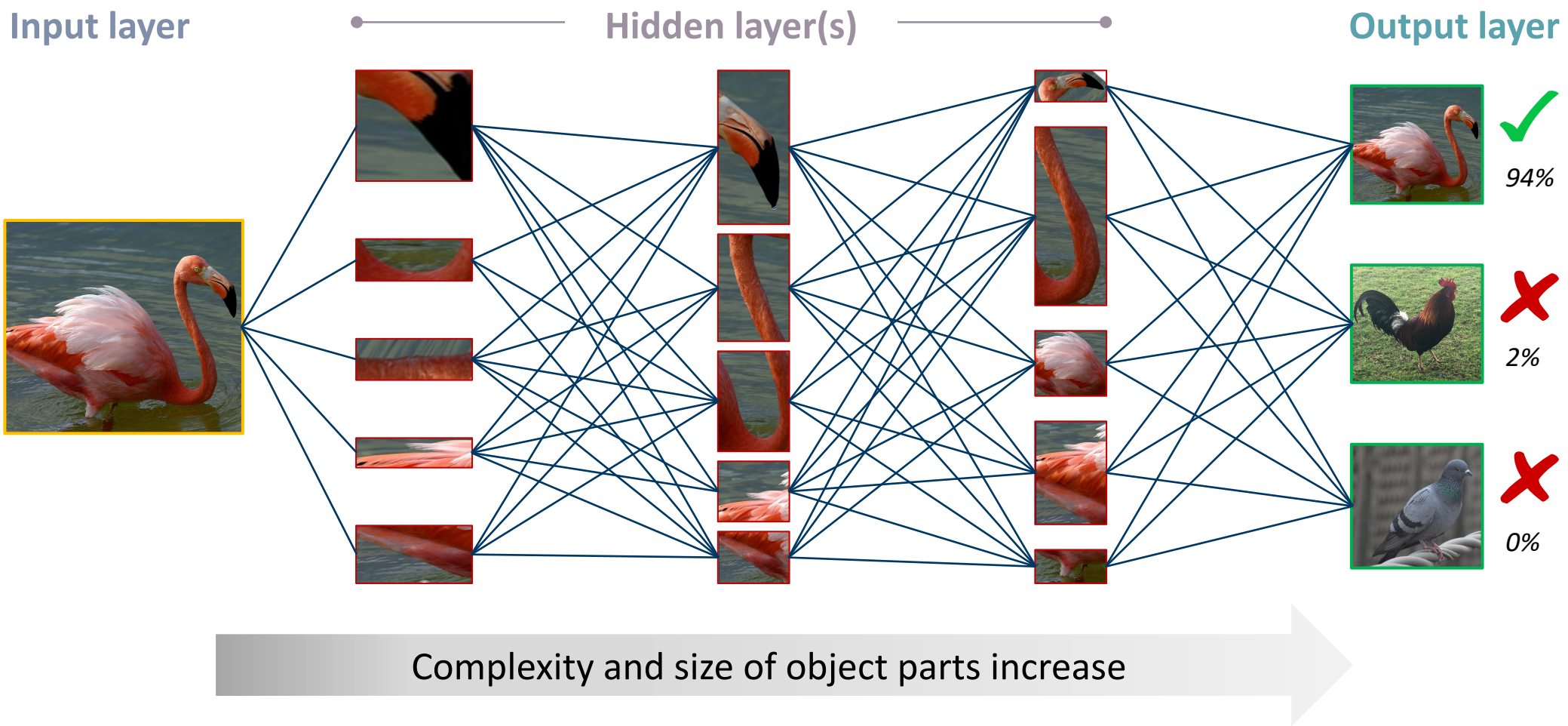


Number of linear regions carved out by **deep rectifier network** with d inputs, depth l, n hidden units per hidden layer:

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right)$$

Montufar, *Neural Computation* 2014; Goodfellow 2016

Neural networks can detect increasingly complex features throughout their layers



Universal Approximator Theorem for width-bounded ReLU networks

we need to determine the area under the function

these are theoretical results!

$f: \mathbb{R}^M \to \mathbb{R}$	arbitrary function Lebesque-integrable function
$\phi(x) = \max(0, x)$	ReLU activation function
$F(\boldsymbol{x})$	Fully-connected ReLU network of width $M + 4$

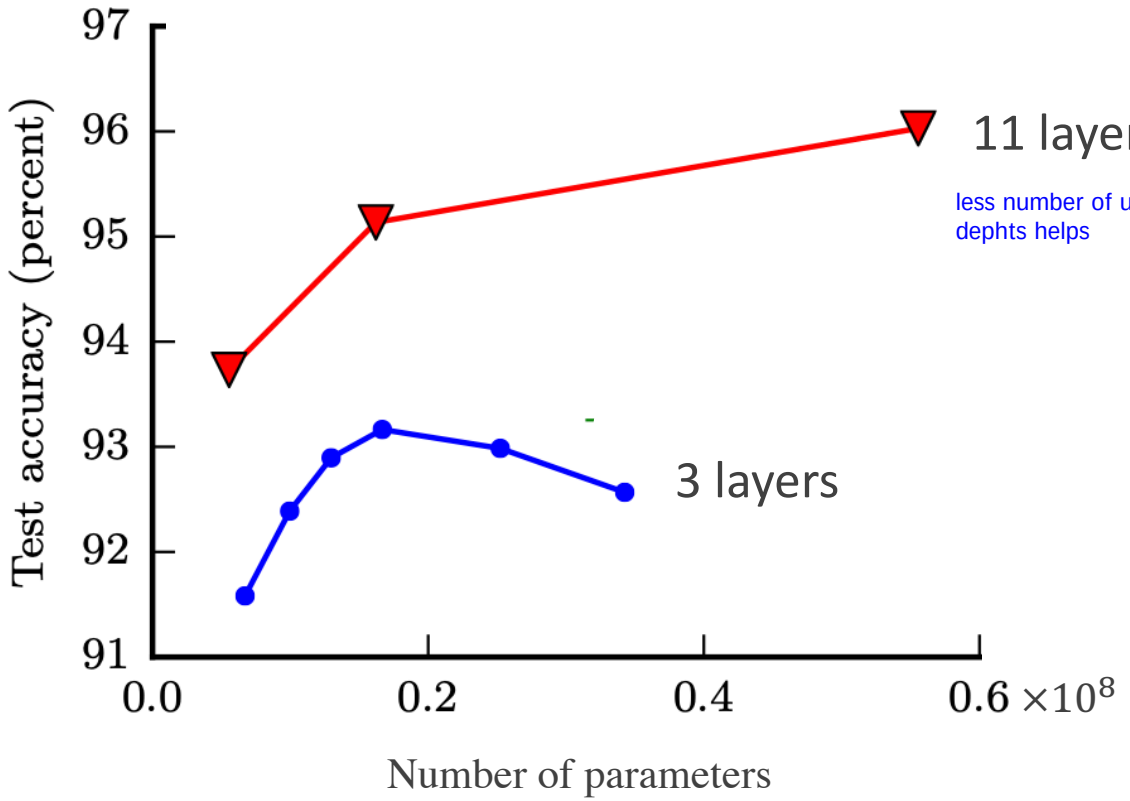
Theorem

For any Lebesque-integrable function $f : \mathbb{R}^M \to \mathbb{R}$ and any $\epsilon > 0$
there exist
a fully-connected ReLU network of finite width $M + 4$
such that
$|F(\boldsymbol{x}) - f(\boldsymbol{x})| < \epsilon$

Large shallow models overfit more

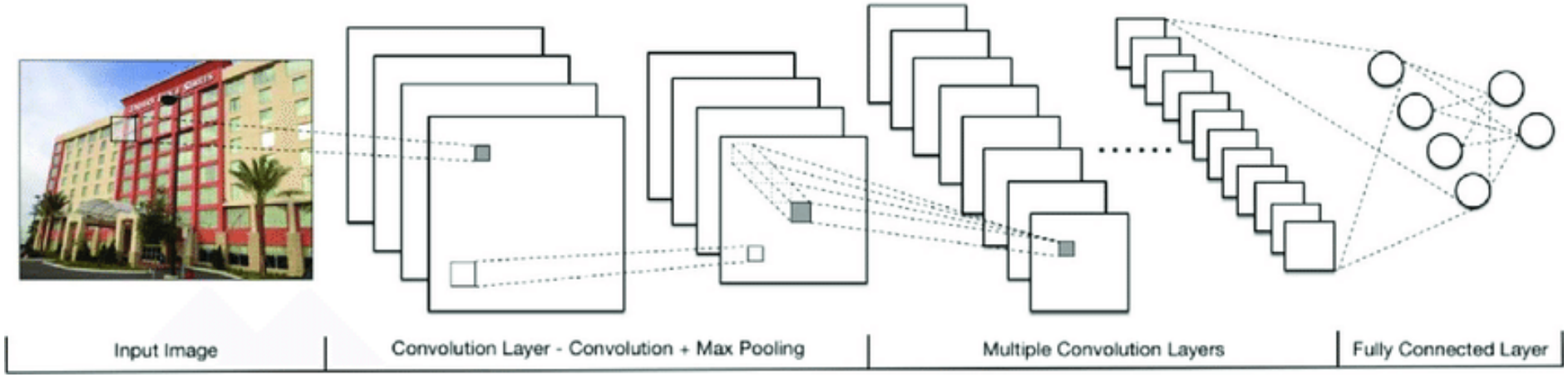
Problem:
House number transcription

Goodfellow et al., *ICLR* 2014



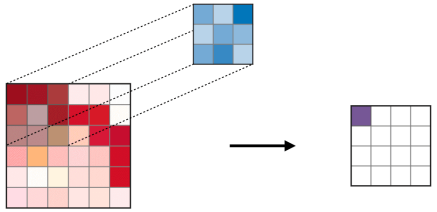
less number of units in each layer, but increasing number of units helps -> dephts helps

Neural network architectures: convolutional nets



Instead mapping every unit, map units locoally - same location
Shares also the wheights for the same location
-> same weights for a location-> reduction number of parameters

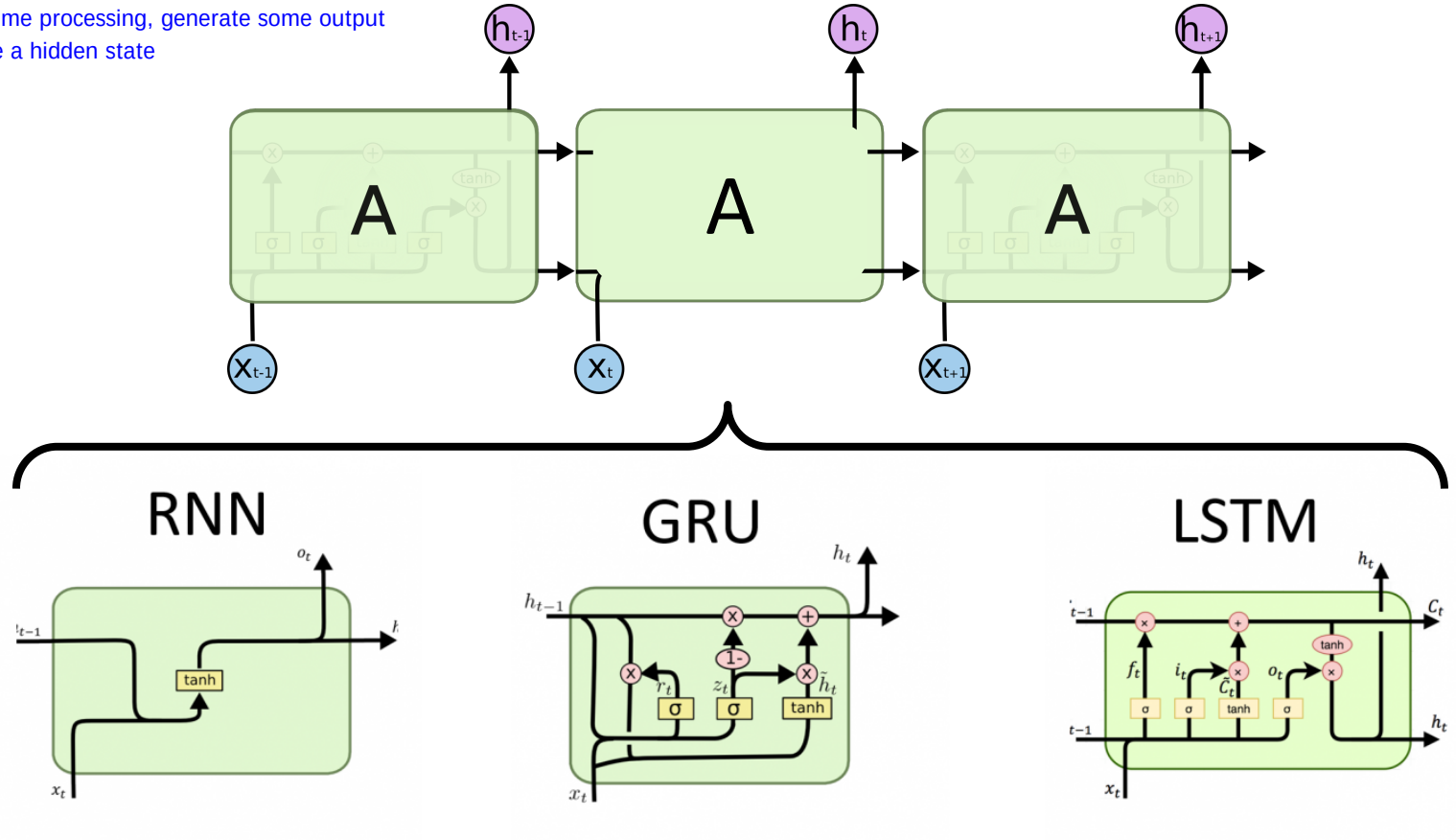
Convolution:



Used for image/video processing

Neural network architectures: recurrent nets

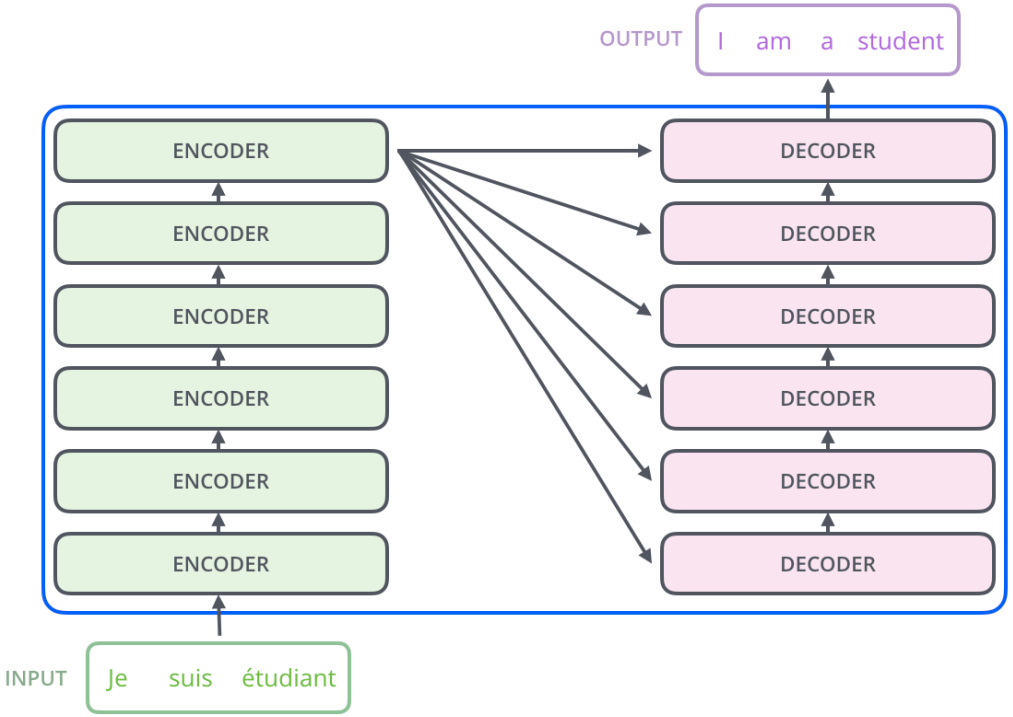
Time depending: do some processing, generate some output
the output can also be a hidden state

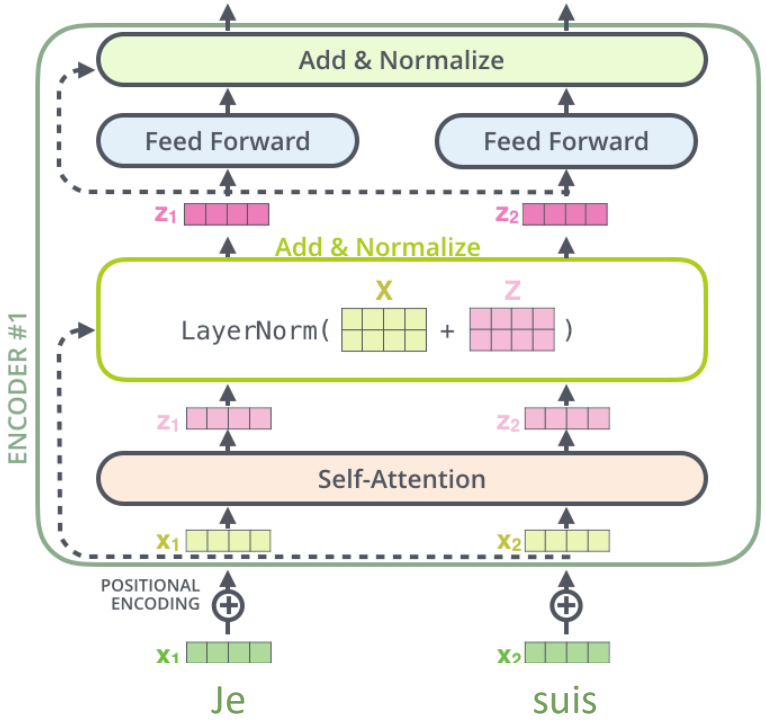


Traditionally used for sequence modeling (e.g. natural language processing)

Neural network architectures: Transformers

Works with attention mechanism





Pretty much completely replaced RNNs for natural language processing since early 2019

A word of caution: No Free Lunch Theorem

Let

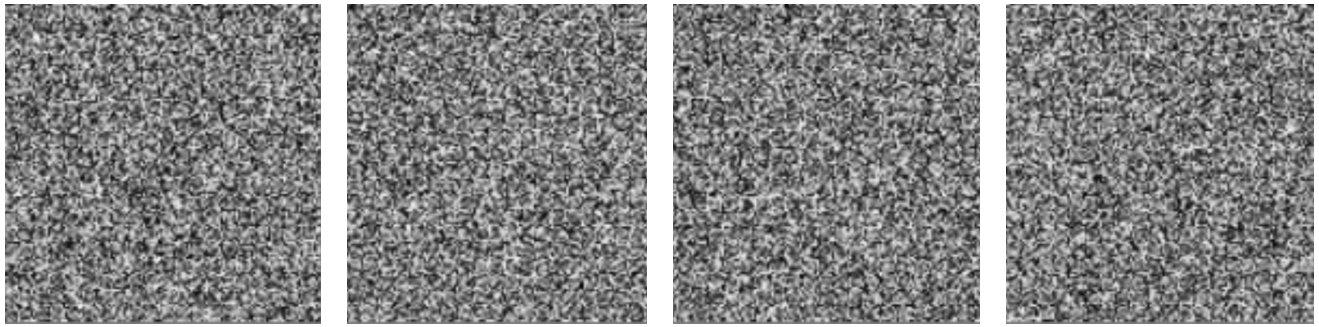
***d:* training set**	
***f:* target function**	true input-output relationship
***h:* hypothesis**	the algorithm's guess for *f* made in response to *d*
***L:* generalization error**	off-training-set loss associated with *f* and *h*

No Free Lunch Theorem

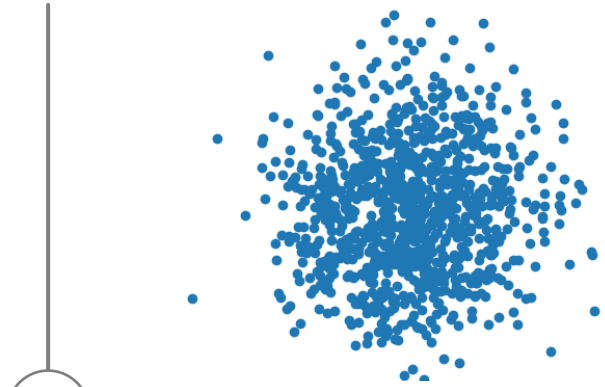
Averaged over all possible functions *f,* all algorithms are equivalent in terms of E[*L*|*d*).

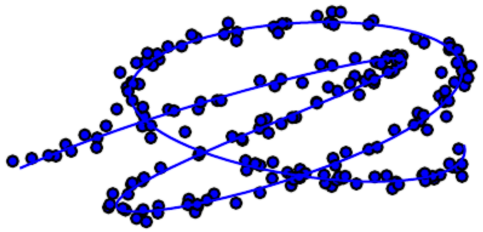
Not all functions are equally likely to be true, though. The world has lots of structure.

Data manifolds



Random sampling from white noise will
pretty much never generate a natural image





The same holds for language → random collection of words don't form meaningful sentences

When to use deep learning?

Problems amenable to deep learning are:

- low-noise
- highly nonlinear
- unstructured data

Examples

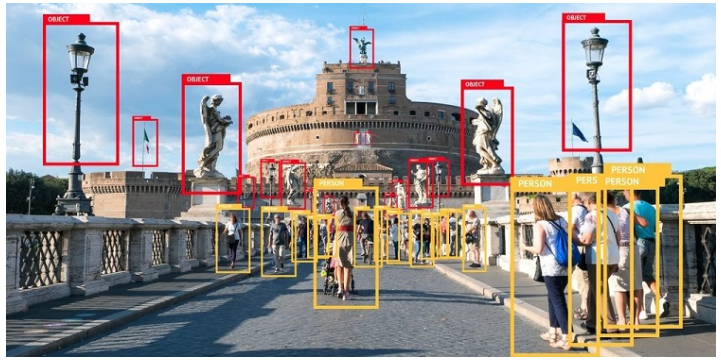
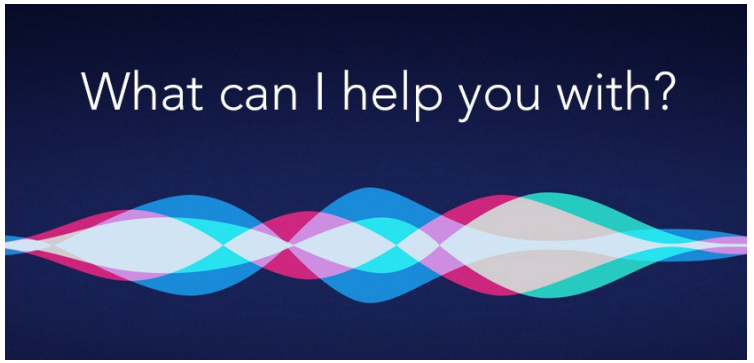


Image recognition

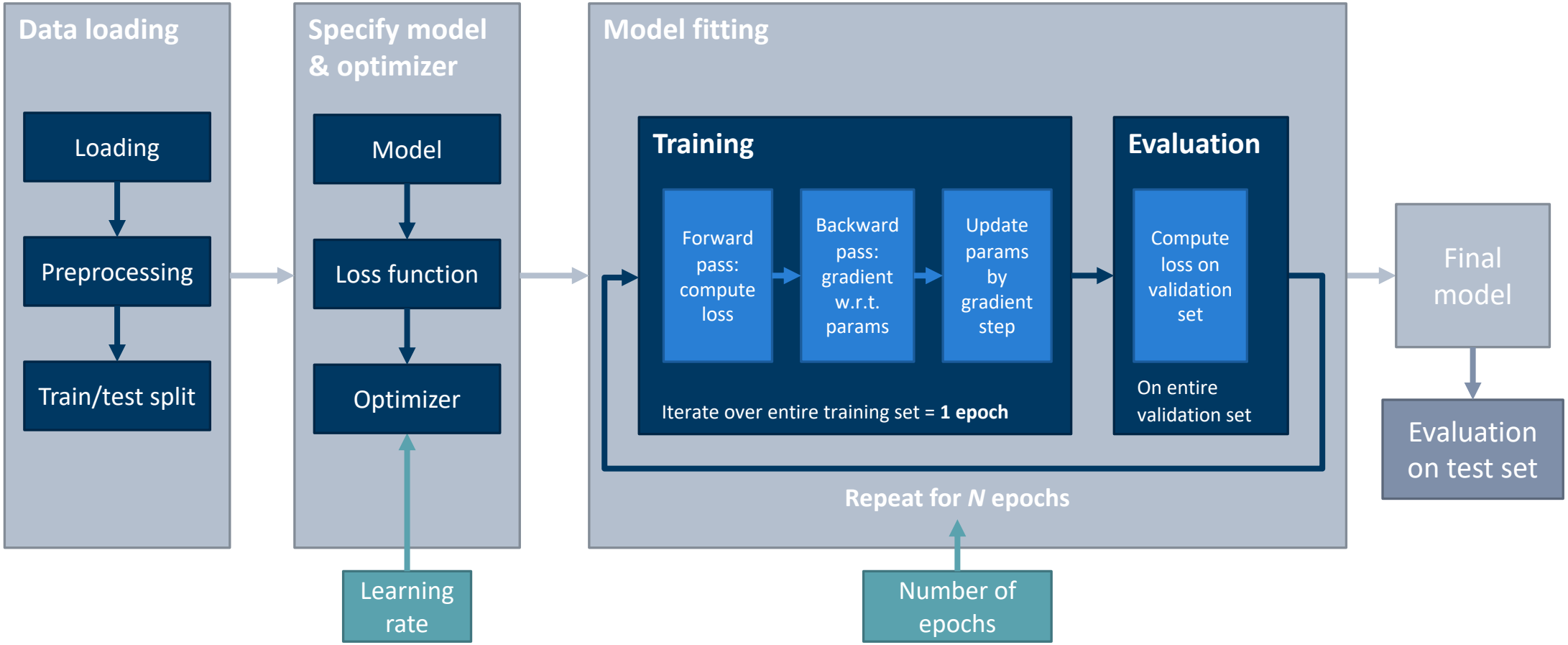


Natural language processing

...

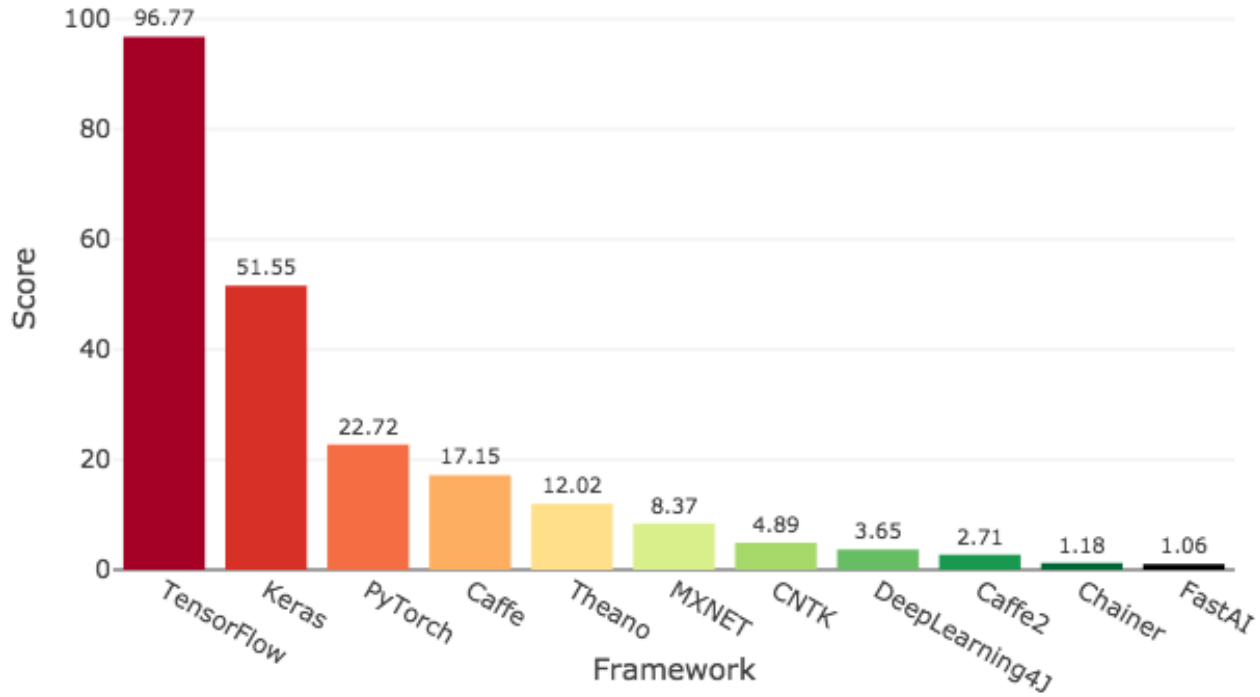
Deep learning frameworks

The process of fitting a machine learning model

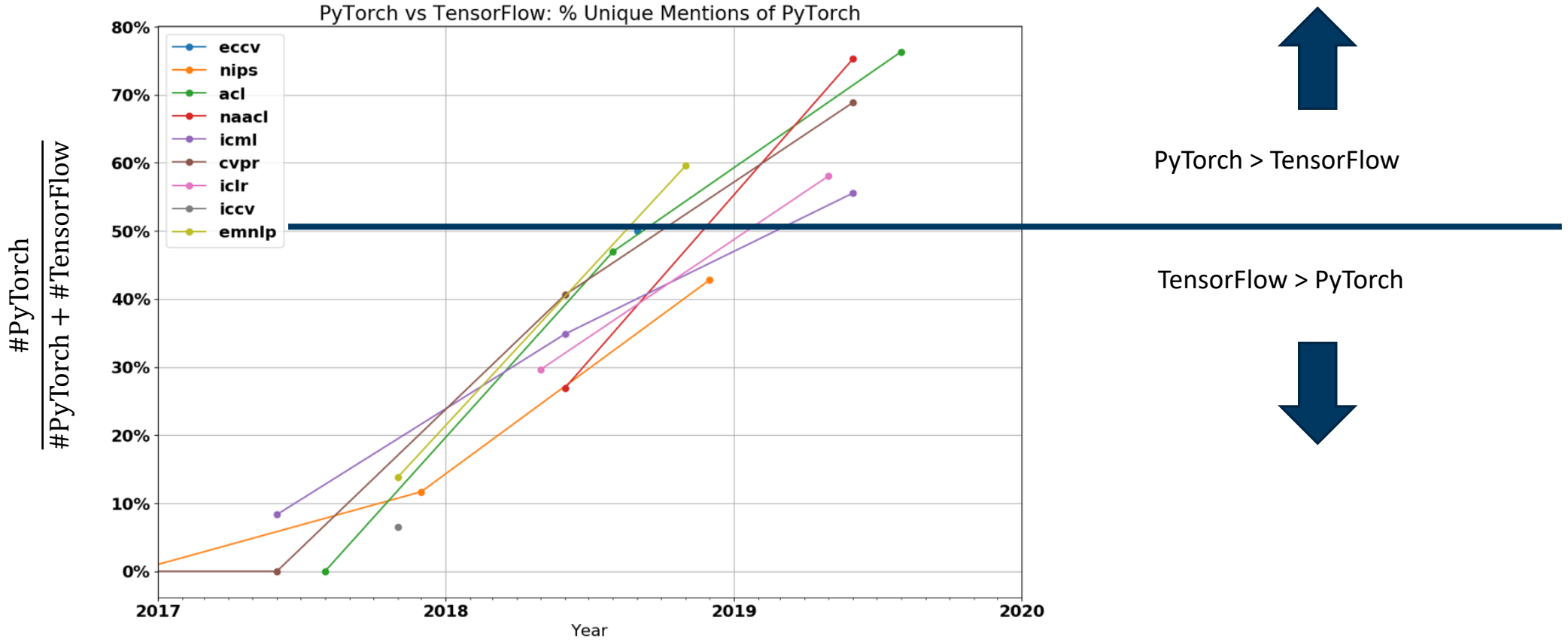


2018: TensorFlow most widely used framework

Deep Learning Framework Power Scores 2018 → Historical data

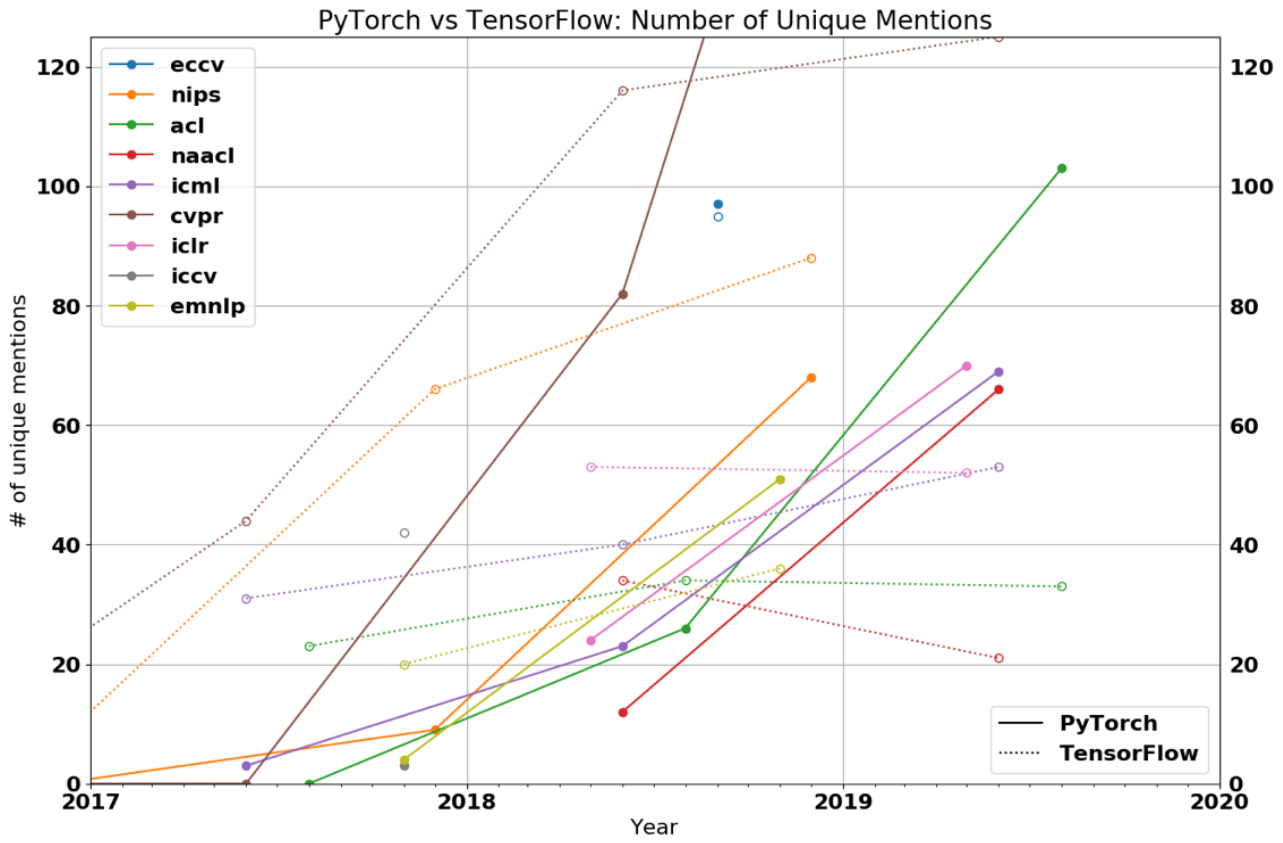


PyTorch surpassed TensorFlow as the most popular framework in 2019



Source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

PyTorch surpassed TensorFlow as the most popular framework in 2019



PyTorch

TensorFlow

Source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

What are these frameworks about?

1. **Automatic differentiation (Autodiff)**
2. **Running computations on different hardware backends (e.g. GPUs)**
3. „Layer“ modules
4. Data loading etc.

What are these frameworks about?

1 **Automatic differentiation (Autodiff)**

Each operation „knows“ its gradient w.r.t. its inputs

You can compose arbitrary chains of operations

Gradient gets computed automatically via chain rule

Backpropagation is a special case of “reverse mode” automatic differentiation

Automatic differentiation

PyTorch code

```
x = torch.ones(2, 2,
        requires_grad=True)

y = x + 2


z = y * y * 3


o = z.mean()

o.backward()
x.grad
```

Output

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)

tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)

tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)

tensor(27., grad_fn=<MeanBackward0>)

tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Math

$$x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$y = x + 2$$

$$z_{ij} = 3y_{ij}^2$$

$$o = \frac{1}{4}\sum_{i,j} z_{ij} = \frac{3}{4}\sum_{i,j}(x_{ij}+2)^2$$

$$\frac{\partial o}{\partial x_{11}} = \frac{3}{2}(x_{11}+2)$$

$$\Rightarrow \left.\frac{\partial o}{\partial x_{11}}\right|_{x_{11}=1} = 4.5$$

What are these frameworks about?

2 **Support for computations on GPU**

GPUs are highly parallel processing units

Originally developed for 3d video gaming

Now mostly used for deep learning and crypto currency mining

Performing computations on GPU with PyTorch

Much of the current success of Deep Learning is also attributed to recent advances in hardware development, especially GPUs

GPUs (graphics processing unit) are optimized for manipulating graphics, which includes linear algebra tasks like, e.g. matrix multiplications

- Contrary to CPUs, GPUs are designed do perform these calulcation in parallel, which makes them a perfect fit for Deep Learning, given that they have enough memory

Companies like NVIDIA today even design GPUs intended for Deep Learning

Machine Learning libraries are often designed s.t. if a GPU is available, it can be easily used for computations without have to care too much about internals

Performing computations on GPU with PyTorch

In PyTorch, you can simply move tensors and models to the GPU and back

- If both model and data reside on the GPU, computations will be performed there
- If only one of both is on the GPU, PyTorch will throw an exception

```
# check if a GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

data = data.to(device)      # move data tensor to device
model = model.to(device)    # move model to device

output = model(data)        # perform computations on GPU if available

output = output.cpu()       # move model to cpu if not already there

output = output.numpy()     # convert tensor to numpy array
```

What are these frameworks about?

3 **Layer modules**

Very fast and easy to build custom deep models

Allow “chaining” different layers together

High-level abstractions for many common concepts and layer types

PyTorch example: nn.Sequential

Models can be easily composed using Pytorch's nn.Sequential() module:

```
model = torch.nn.Sequential(
    torch.nn.Linear(1, 10),
    torch.nn.Dropout(0.5),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1),
)

output = model(input)
```

What are these frameworks about?

4 **Data loading and handling**

Data loaders are common abstractions to handle typical workflows

Many tasks regarding training data is recurring, e.g. loading of data, iterating over it, shuffling data, apply augmentation, …

PyTorch helps with providing well designed interfaces and helper classes to handle such tasks

Deep Learning with PyTorch

https://pytorch.org/deep-learning-with-pytorch

Happy holidays!

Questions!