Deep Learning

Power of the Depth

Lionel Fillatre

April 2021

Outline of Lecture 5

- Introduction
- Benefits of depth
- Vanishing gradients
- Conclusion

Introduction

Going deeper?

- Why would it be a good idea to stack more layers?
- Are there any theoretical insights for doing his? Empirical ones?

ReLU Deep Networks

- Let us remember that a feed-forward deep neural network has the form :

$$\hat{y} = f(x;\theta) = \left(f_{\theta_L} \circ f_{\theta_{L-1}} \circ \cdots \circ f_{\theta_1}\right)(x)$$

where $\theta = \{\theta_k : k = 1, \cdots, L\}$ denotes the model parameters

- Les us assume that $\theta_k = (W_k, b_k)$ and, for any $h \in \mathbb{R}^{n_{k-1}}$,

$$f_{\theta_k}(h) = \sigma(W_k h + b_k)$$

where, for any $z \in \mathbb{R}^n$,

$$\sigma(z) = \begin{pmatrix} \mathrm{ReLU}(z_1) \\ \vdots \\ \mathrm{ReLU}(z_n) \end{pmatrix} = \begin{pmatrix} \max\{0, z_1\} \\ \vdots \\ \max\{0, z_n\} \end{pmatrix}$$

Approximation with ReLU nets

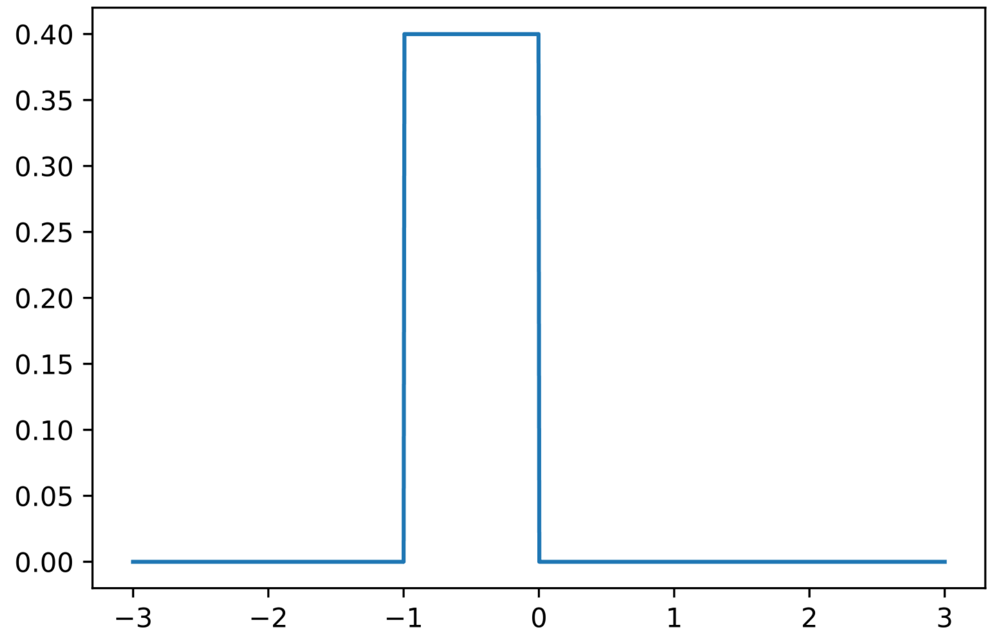
```
import numpy as np
import matplotlib.pyplot as plt

def relu(x):
        return np.maximum(x, 0)

def rect(x, a, b, h, eps=1e-7):
        return h / eps * (
                relu(x - a)
                - relu(x - (a + eps))
                - relu(x - b)
                + relu(x - (b + eps)))

x = np.linspace(-3, 3, 1000)
y = ( rect(x, -1, 0, 0.4))

plt.plot(x, y)
```



Approximation with ReLU nets

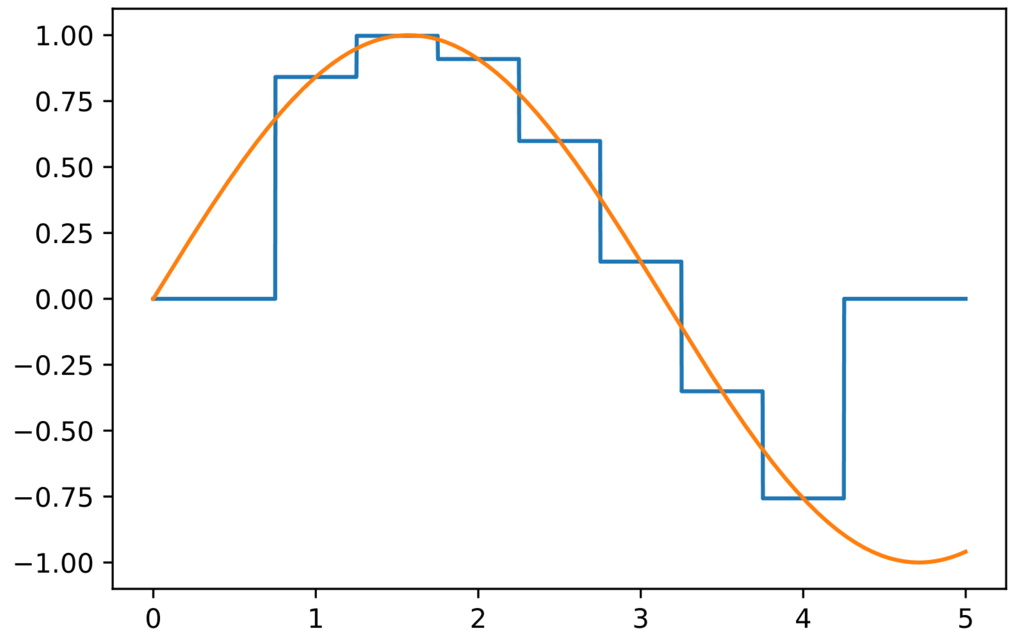
```
import numpy as np
import matplotlib.pyplot as plt


def relu(x):
        return np.maximum(x, 0)

def rect(x, a, b, h, eps=1e-7):
        return h / eps * (
                relu(x - a)
                - relu(x - (a + eps))
                - relu(x - b)
                + relu(x - (b + eps)))


x = np.arange(0,5,0.5) # 10 samples
z = np.arange(0,5,0.001)
sin_approx = np.zeros_like(z)

for i in range(2, x.size-1):
        sin_approx = sin_approx + rect(z,(x[i]+x[i-1])/2,
                (x[i]+x[i+1])/2, np.sin(x[i]), 1e-7)
plt.plot(x, y)
```



Conner Davis, Quora: Is a single layered ReLU network still a universal approximator?

Approximation with ReLU nets

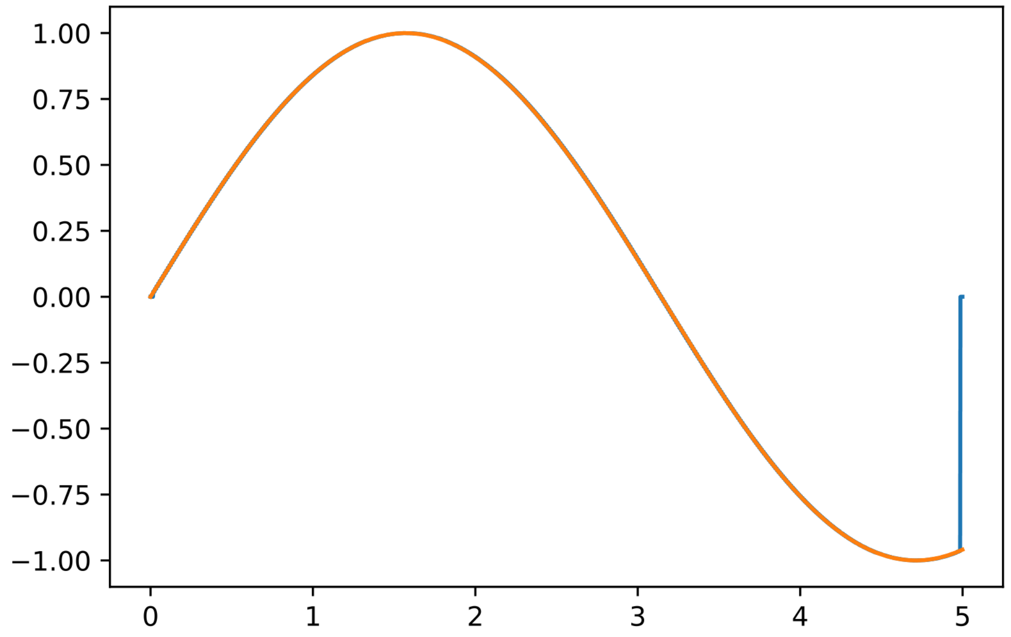
```
import numpy as np
import matplotlib.pyplot as plt

def relu(x):
        return np.maximum(x, 0)

def rect(x, a, b, h, eps=1e-7):
        return h / eps * (
                relu(x - a)
                - relu(x - (a + eps))
                - relu(x - b)
                + relu(x - (b + eps)))

x = np.arange(0,5,0.01) # 500 samples
z = np.arange(0,5,0.001)
sin_approx = np.zeros_like(z)

for i in range(2, x.size-1):
        sin_approx = sin_approx + rect(z,(x[i]+x[i-1])/2,
                (x[i]+x[i+1])/2, np.sin(x[i]), 1e-7)
plt.plot(x, y)
```



Conner Davis, Quora: Is a single layered ReLU network still a universal approximator?

Universal function approximation

Theorem. (Hornik et al, 1991)

- Let σ be a nonconstant, bounded, and monotonically-increasing continuous function.
- For any $f \in C\big([0,1]^d\big)$ and $\varepsilon > 0$, there exists $q \in \mathbb{N}$, real constants $v_i, b_i \in \mathbb{R}$ and $w_i \in \mathbb{R}^d$ such that:

$$\left|\sum_{i=1}^{q} v_i \sigma\big(w_i^T x + b_i\big) - f(x)\right| < \varepsilon$$

- In other words, neural nets are dense in $C\big([0,1]^d\big)$.

- It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth);
- It does not inform about good/bad architectures, nor how they relate to the optimization procedure.
- The universal approximation theorem generalizes to any non-polynomial (possibly unbounded) activation function, including the ReLU (Leshno, 1993).

K. Hornik et al., Approximation Capabilities of Multilayer Feedforward Networks, 1991

Upper-Bound for one-hidden layer network

Theorem (Barron, 1994)

- The mean integrated square error between the estimated network $\hat{f}(x) = \sum_{i=1}^{q} v_i \sigma(w_i^T x + b_i) + v_0$ and the target function f is bounded by

$$O\left(\frac{C_f^2}{q}\right) + O\left(\frac{qn}{N} \log N\right)$$

where N is the number of training points, q is the number of neurons, n is the input dimension, and C_f measures the global smoothness of f.

- Provided enough data, it guarantees that adding more neurons will result in a better approximation.

- For your information,

$$C_f = \int \|\omega\|_1 \tilde{f}(\omega) d\omega$$

where $\tilde{f}(\omega)$ is the Fourier transform of $f(x)$.

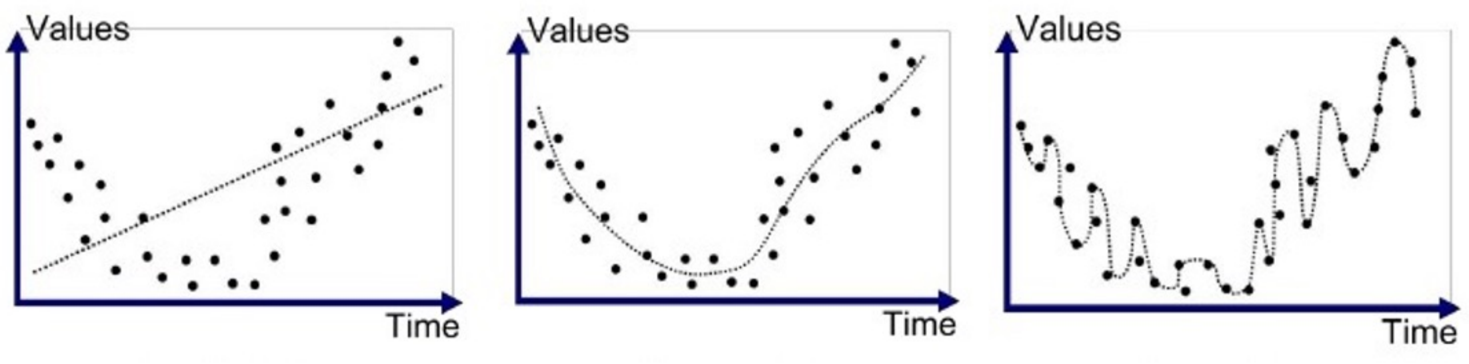
Problem solved?

- Universal function approximation theorems do not tell us:
  - The number q of hidden units is small enough to have the network fit in RAM.
  - There is no constructive way to find an optimal solution
  - The optimal function parameters can be found in finite time by minimizing the Empirical Risk with SGD and the usual random initialization schemes.

- Going deeper?
  - Why would it be a good idea to stack more layers?
  - Are there any theoretical insights for doing his? Empirical ones?

Universal approximation

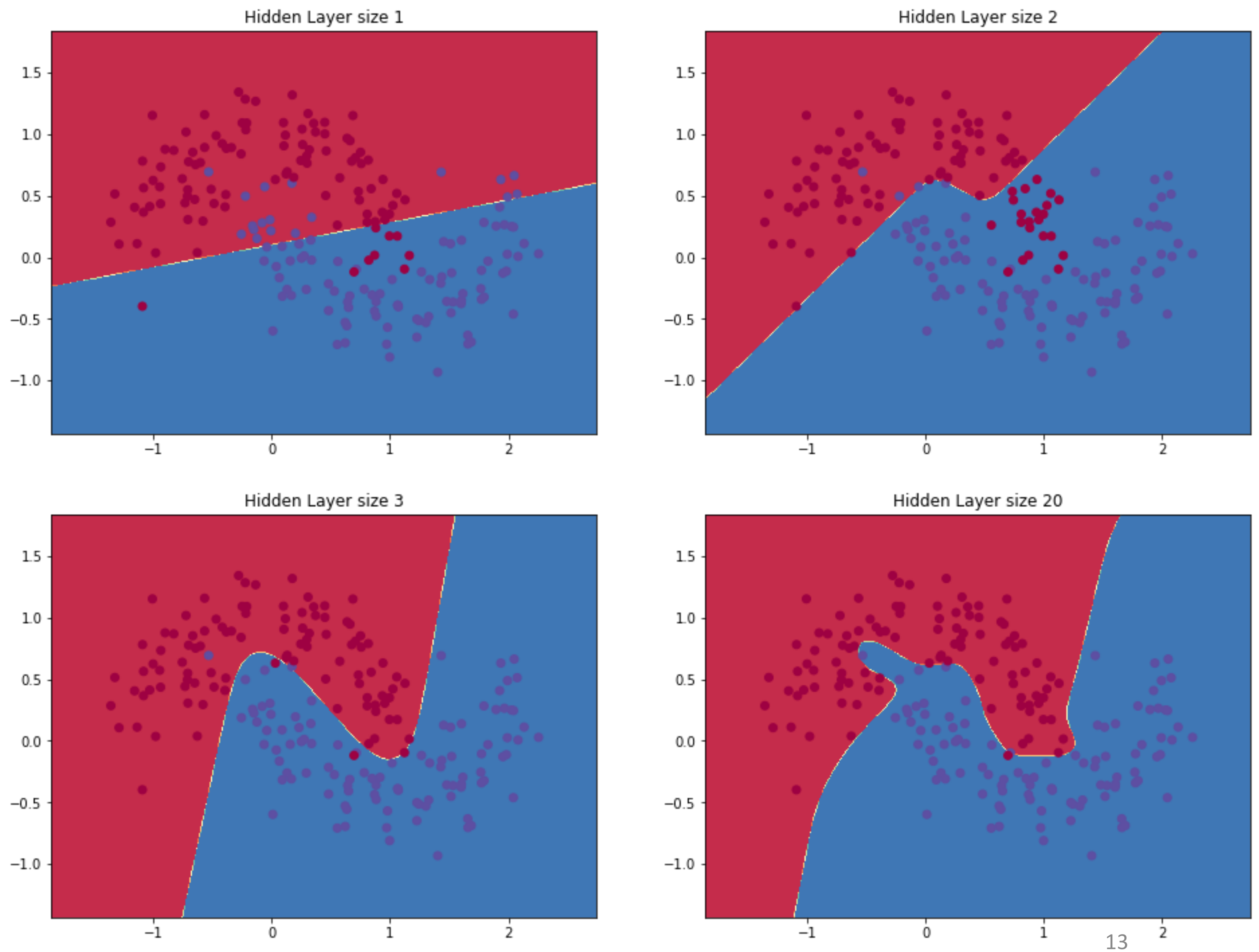
- Even if the MLP is able to represent the function, learning can fail for two different reasons:
  - the optimization algorithm may not be able to find the value of the parameters that correspond to the desired function
  - the training algorithm might chose the wrong function as result of overfitting



Underfitted Good Fit/Robust Overfitted

Overfitting

- Two moons
- 1-Hidden layer NN



Finite-sample expressivity

- As soon as the number of parameters of a networks is greater than N, even simple two-layer neural networks can represent any function of the input sample.

- We say that a neural network $f_\theta(x)$ can represent any function of a sample of size N in n dimensions if for every sample $S \subset (\mathbb{R}^n)^N$ with $|S| = N$ and every function $f: S \to \mathbb{R}$, there exists a setting of the weights θ of $f_\theta(x)$ such that $f_\theta(x) = f(x)$ for every $x \in S$.

Theorem (Zhang, 2016)

- There exists a two-layer neural network with ReLU activations and $2N + n$ weights that can represent any function on a sample of size N in n dimensions.

Corollary

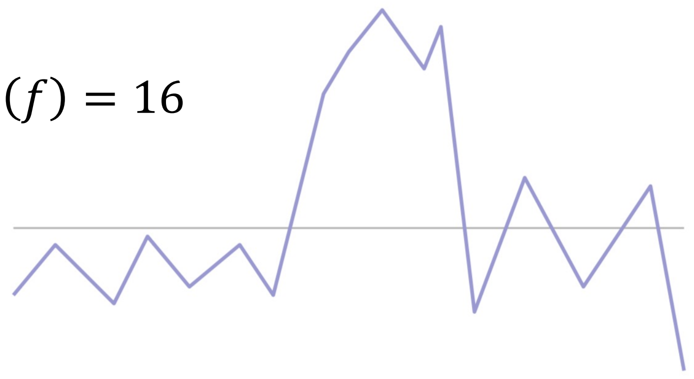
- For every $k \geq 2$, there exists a neural network with ReLU activations of depth k, width $O\left(\frac{N}{k}\right)$ and $O(N + n)$ weights that can represent any function on a sample of size N in n dimensions.

Zhang, C.; Bengio, S.; Hardt, M.; Recht, B. & Vinyals, O. Understanding deep learning requires rethinking generalization, 2016.

Benefits of Depth

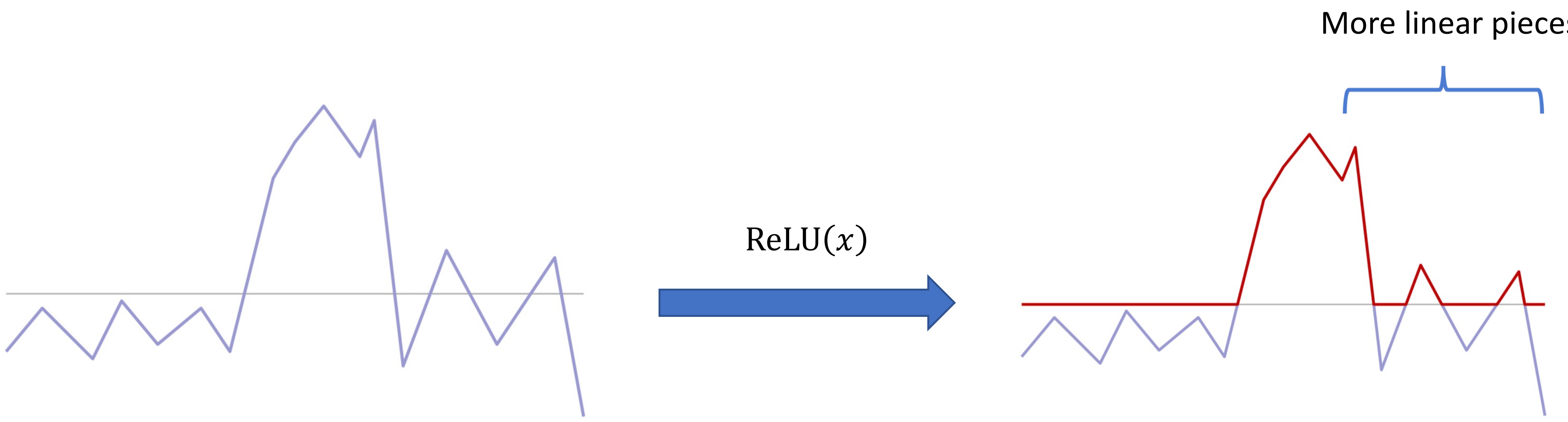
Number of linear pieces

$\kappa(f) = 16$



- Let $\mathcal{F}$ be the set of piecewise linear mappings on $[0,1]$
- Let $\kappa(f)$ be the minimum number of linear pieces needed to represent $f \in \mathcal{F}$.
- Let $\sigma \colon \mathbb{R} \to \mathbb{R}$ be the ReLU function
$$\sigma(x) = \mathrm{ReLU}(x) = \max(0, x)$$
- If we compose σ and $f \in \mathcal{F}$, any linear piece that does not cross 0 remains a single piece or disappears, and one that does cross 0 breaks into two, hence
$$\forall f \in \mathcal{F}, \kappa\big(\sigma(f)\big) \leq 2\,\kappa(f)$$
- We also have
$$\forall (f, g) \in \mathcal{F}^2, \kappa(f + g) \leq \kappa(f) + \kappa(g)$$

Illustration



$\kappa(f) = 16$

$\kappa\big(\sigma(f)\big) = 14$

Bound on the number of linear pieces

- Consider a MLP with ReLU, D layers, a single input unit, and a single output unit.
    - Single unit input layer: $h^0 = x \in \mathbb{R}^{n_0=1}$
    - Hidden layers: $h^d = \left(h_1^d, \dots, h_{n_d}^d\right),\ \forall d = 1, \dots, D$, with $h_i^d = \sigma\left(z_i^{d-1}\right)$ and $z_i^{d-1} = \sum_{j=1}^{n_{d-1}} w_{ij}^{d-1} h_j^{d-1} + b_i^{d-1}$
    - Single unit output layer: $\hat{y} = f(x) = h^D \in \mathbb{R}^{n_D=1}$

- Then, we get

$$\forall i, \ell, \qquad \kappa\left(h_i^d\right) = \kappa\left(\sigma\left(z_i^{d-1}\right)\right) \leq 2\kappa\left(z_i^{d-1}\right) \leq 2 \sum_{j=1}^{n_{d-1}} \kappa\left(h_j^{d-1}\right)$$

- It follows that

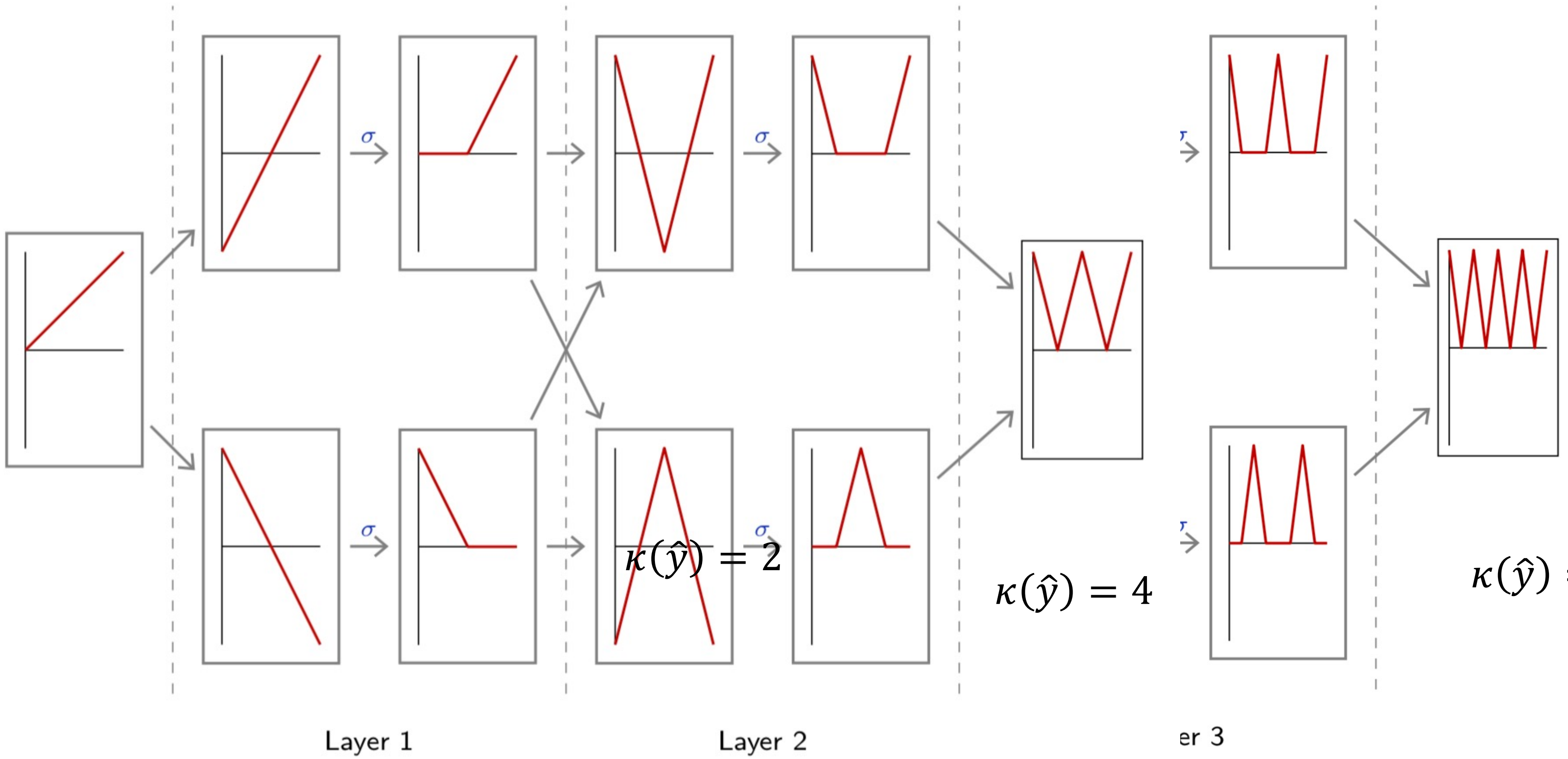
$$\forall d, \qquad \max_i \kappa\left(h_i^d\right) \leq 2 n_{d-1} \max_j \kappa\left(h_j^{d-1}\right)$$

- We get the following bound for any ReLU MLP

$$\kappa(\hat{y}) \ \leq 2^D \prod_{d=1}^{D} n_d$$

Tight bound?

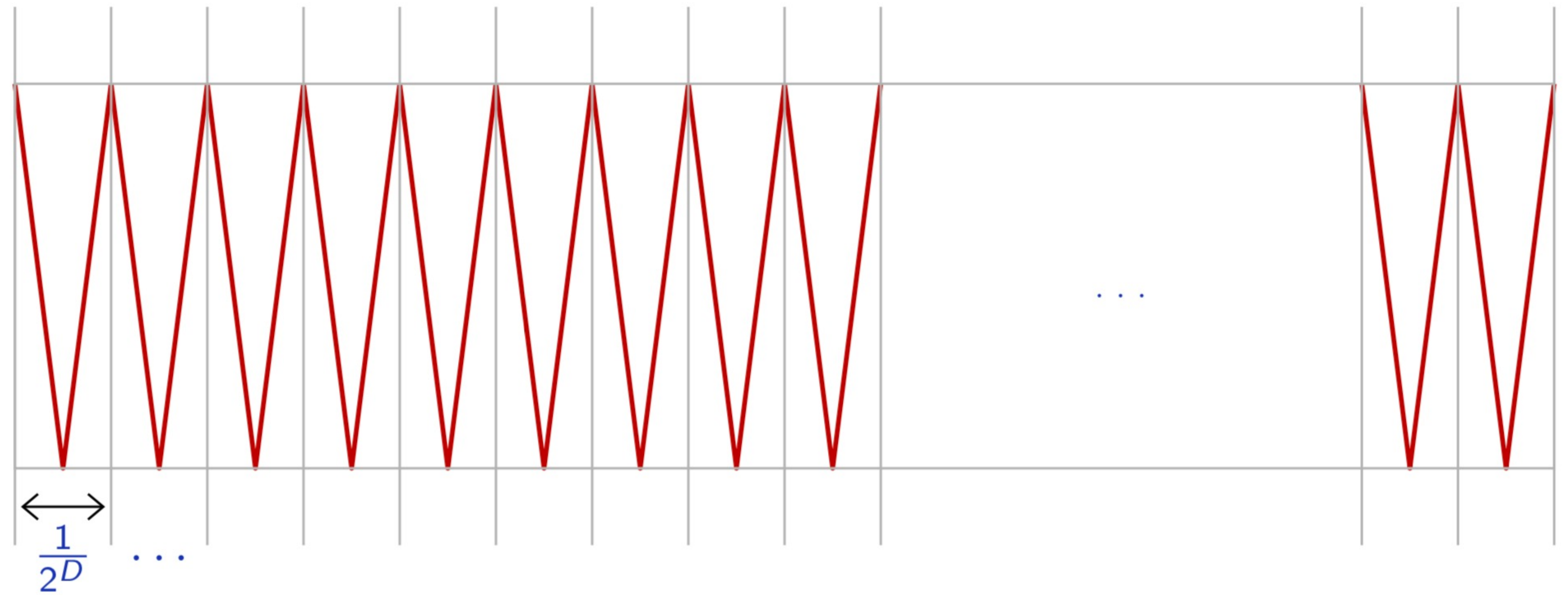
- Although this seems quite a pessimist bound, we can hand-design a network that (almost) reaches it



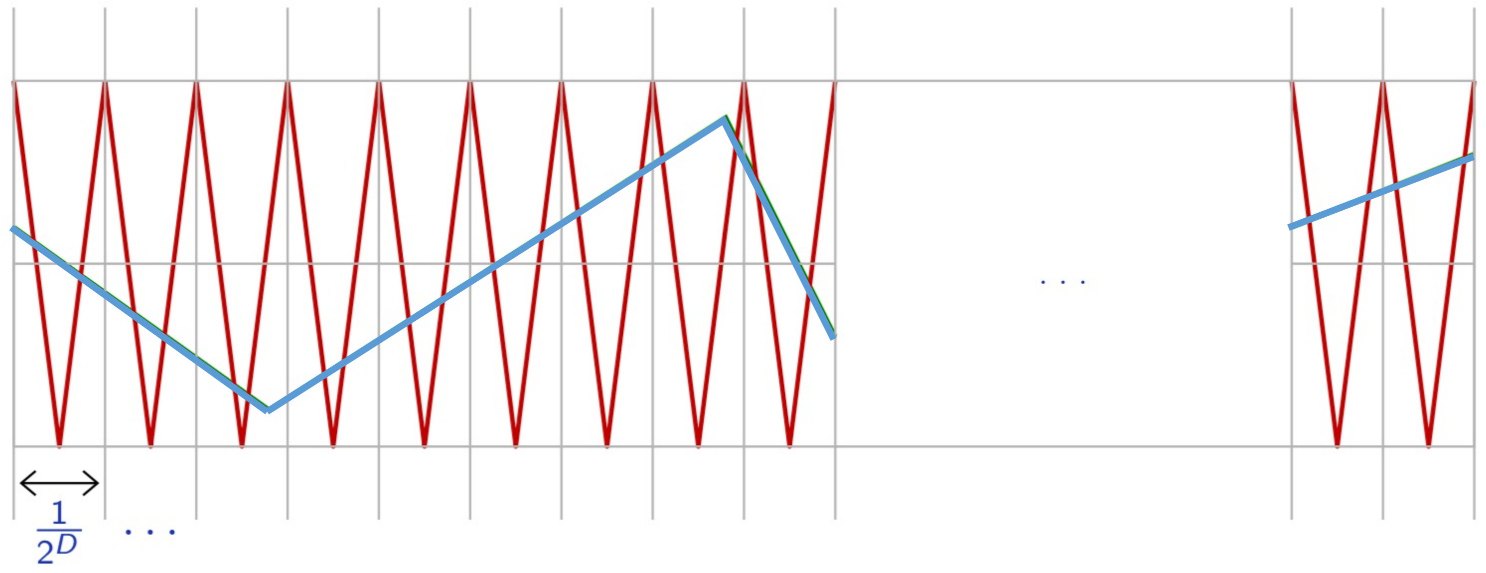
$\kappa(\hat{y}) = 2$ $\kappa(\hat{y}) = 4$ $\kappa(\hat{y}) = 8$

Triangle wave

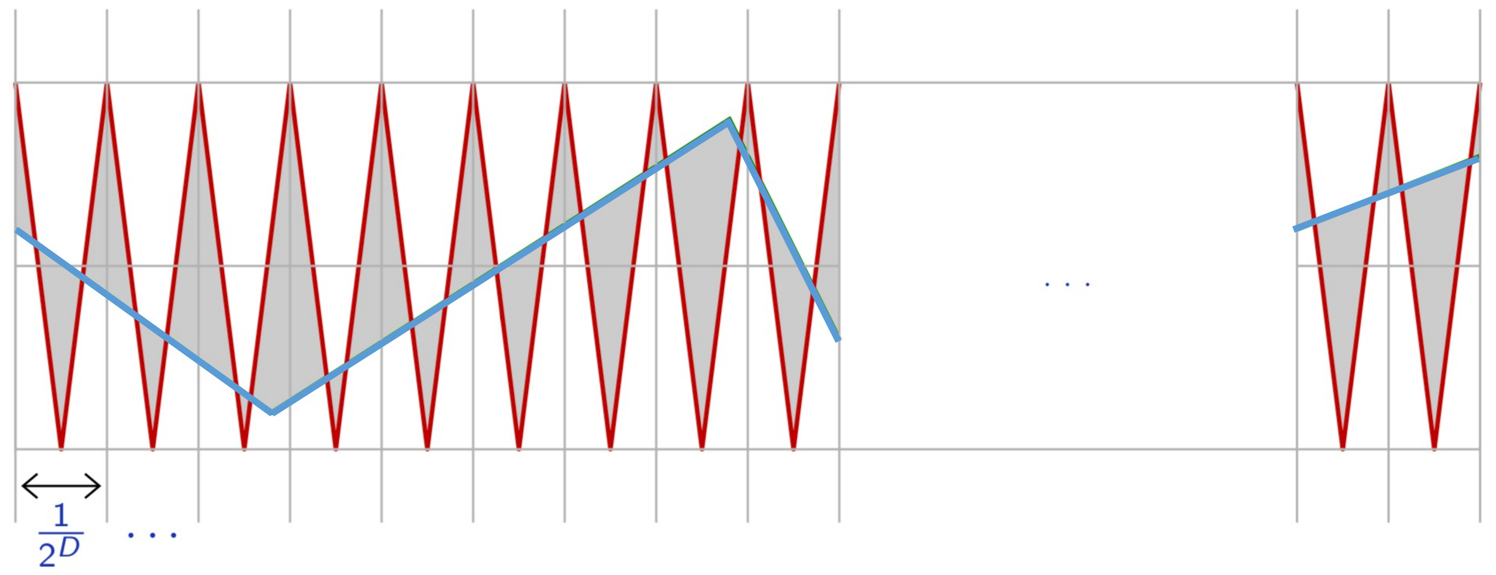
- So for any D, there is a network with D hidden layers and $2D$ hidden units which computes a function $f: [0,1] \to [0,1]$ of period $\frac{1}{2^D}$



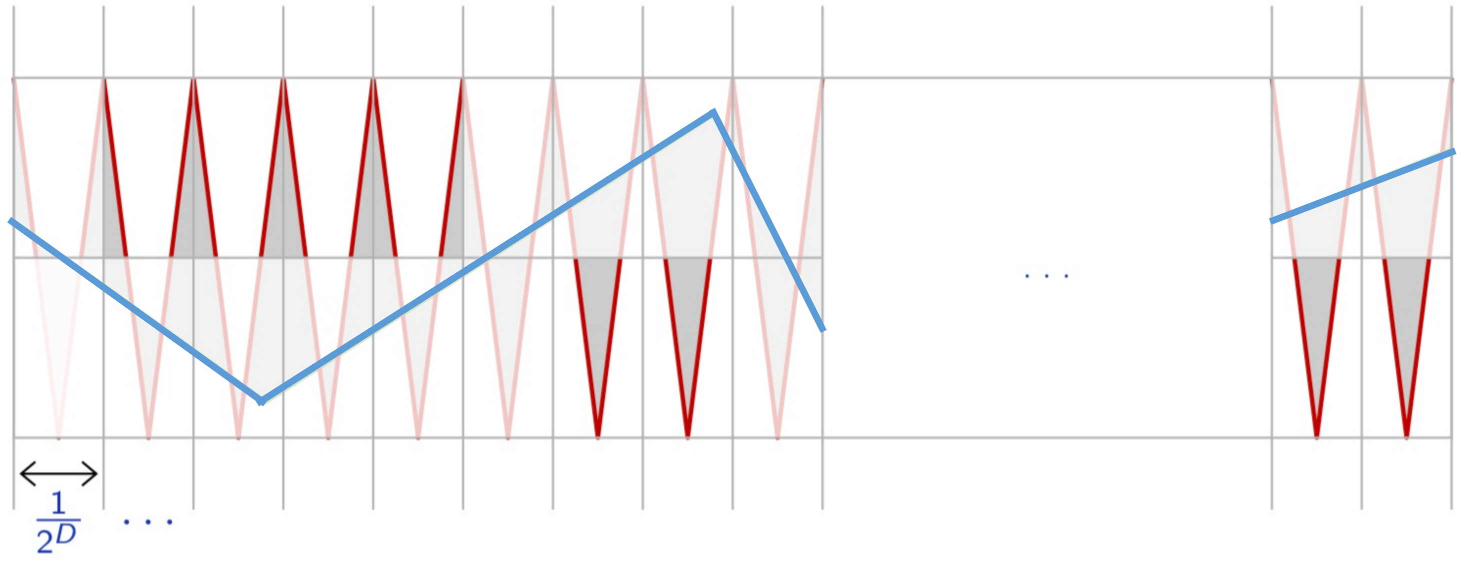
Approximation of the triangle wave



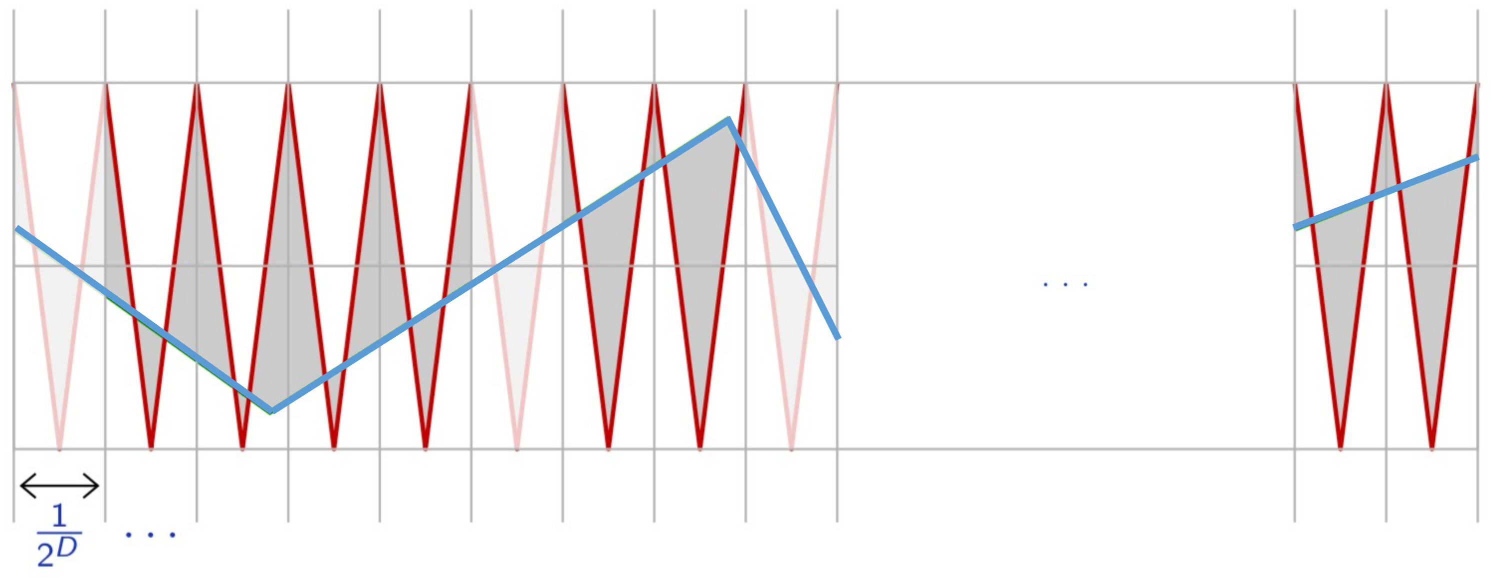
Approximation of the triangle wave



Approximation of the triangle wave



Approximation of the triangle wave



Approximation of the triangle wave $f(x)$

- Given $g \in \mathcal{F}$, it crosses $\frac{1}{2}$ at most $\kappa(g)$ times,
- Which means that, on at least $2^D - \kappa(g)$ segments of length $\frac{1}{2^D}$, it is on one side of $\frac{1}{2}$,
- It follows that

$$\int_0^1 |f(x) - g(x)|\, dx \geq \left(2^D - \kappa(g)\right) \overbrace{\frac{1}{2}\int_0^{\frac{1}{2^D}} \left|f(x) - \frac{1}{2}\right| dx}^{\text{Error on 1 segment for the constant function } c(x) = \frac{1}{2}}$$

$$= \left(2^D - \kappa(g)\right)\frac{1}{2}\cdot\frac{1}{2^D}\cdot\frac{1}{8}$$

$$= \frac{1}{16}\left(1 - \frac{\kappa(g)}{2^D}\right)$$

- We multiply f by 16 to get the final result: $\int_0^1 |f(x) - g(x)|\, dx \geq 1 - \frac{\kappa(g)}{2^D}$

ReLU MLPs with a single input/output

- There exists a network f with D layers, and $2D$ internal units, such that, for any network g with D' layers of sizes $\{n_1, \dots, n_{D'}\}$

$$\int_0^1 |f(x) - g(x)|\, dx \geq 1 - \frac{2^{D'}}{2^D} \prod_{d=1}^{D'} n_d$$

- In particular, with g a single hidden layer network ($D' = 1$)

$$\int_0^1 |f(x) - g(x)|\, dx \geq 1 - \frac{2n_1}{2^D}$$

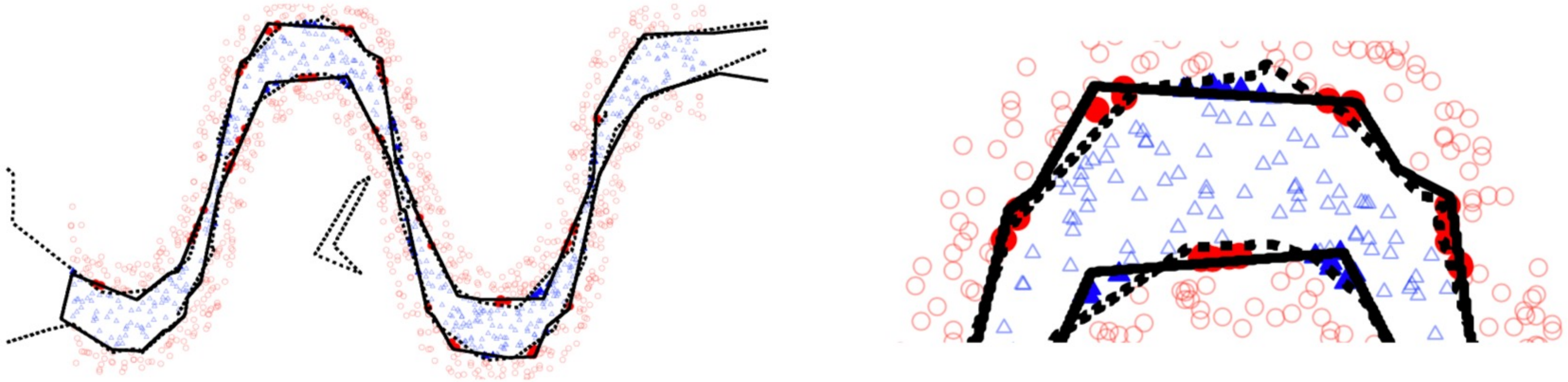
- To approximate f properly, the width n_1 of g's hidden layer has to increase exponentially with f's depth D.
- This is a simplified variant of results by Telgarsky (2015, 2016).

Benefits of depth

- Hence, it can be shown
  - Functions with few oscillations poorly approximate functions with many oscillations.
  - Functions computed by networks with few layers must have few oscillations.
  - Functions computed by networks with many layers can have many oscillations.

M. Telgarsky, Benefits of depth in neural networks, COLT 2016

Deeper is better?

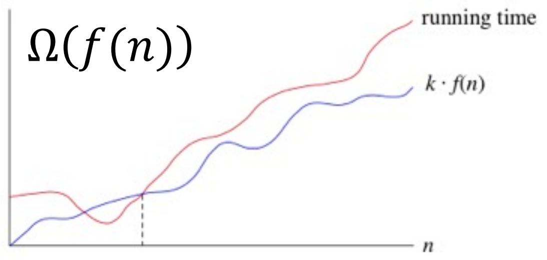


- Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line).
- The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

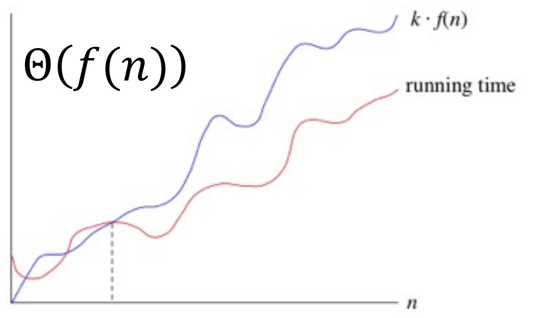
Montufar et al.; On the number of linear regions of deep neural networks; 2014

Depth and Parametric Cost

- **Theorem** (Telgarsky, 2016): There exists functions that can be approximated by a deep ReLU network with $\Theta(n^3)$ layers with $\Theta(1)$ units that cannot be approximated by shallower networks with $\Theta(n)$ layers unless they have $\Omega(2^n)$ units.
- Note: the number of parameters of a deep network is typically quadratic with the number of units.
- This also holds for ReLU convnets with max pooling layers.



- Notation:
    - If a running time is $\Omega(f(n))$, then for large enough n, the running time is at least $k \cdot f\ (n)$ for some constant C
    - If a running time is $\Theta(f(n))$, then for large enough n, the running time is at most $k \cdot f\ (n)$ for some constant C



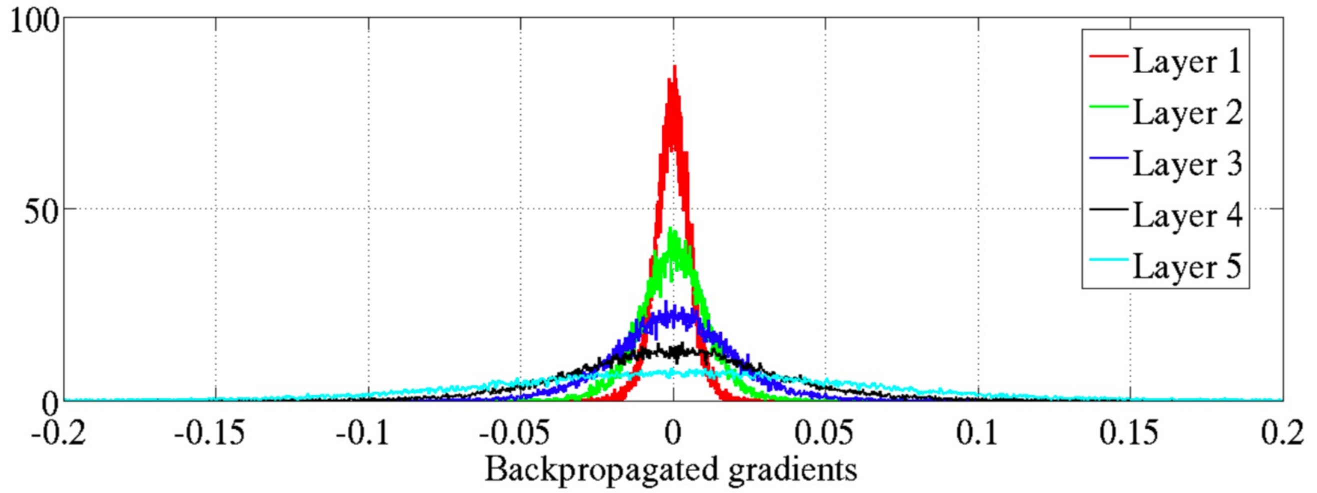
The problem of depth

- Although it was known that deeper is better, for decades training deep neural networks was highly challenging and unstable.
- Besides limited hardware and data there were a few algorithmic flaws that have been fixed/softened in the last decade.
- An important issue is to control the amplitude of the gradient, which is tightly related to controlling activations.
- In particular we have to ensure that
  - The gradient does not « vanish » (Bengio et al., 1994; Hochreiter et al., 2001),
  - The gradient amplitude is homogeneous so that all parts of the network train at the same rate (Glorot and Bengio, 2010),
  - The gradient does not vary too unpredictably when the weights change (Balduzzi et al., 2017).

Vanishing gradients

Vanishing gradients

- Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the vanishing gradient problem.
    - Small gradients slow down, and eventually block, stochastic gradient descent.
    - This results in a limited capacity of learning.



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).
Gradients for layers far from the output vanish to zero.

Glorot and Bengio, Understanding the difficulty of training deep feedforward neural networks; AISTAT 2010

Vanishing gradients

- Consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x, w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma(w_1 x)\right)\right)$$

- Under the hood, this would be evaluated as

$$\begin{gathered} u_1 = w_1 x \\ u_2 = \sigma(u_1) \\ u_3 = w_2 u_2 \\ u_4 = \sigma(u_3) \\ u_5 = w_3 u_4 \\ \hat{y} = \sigma(u_5) \end{gathered}$$

- Its derivative $\frac{d\hat{y}}{dw_1}$ is

$$\frac{d\hat{y}}{dw_1} = \frac{d\hat{y}}{du_5}\frac{du_5}{du_4}\frac{du_4}{du_3}\frac{du_3}{du_2}\frac{du_2}{du_1}\frac{du_1}{dw_1} = \frac{\partial\sigma(u_5)}{\partial u_5} w_3 \frac{\partial\sigma(u_3)}{\partial u_3} w_2 \frac{\partial\sigma(u_1)}{\partial u_1} x$$

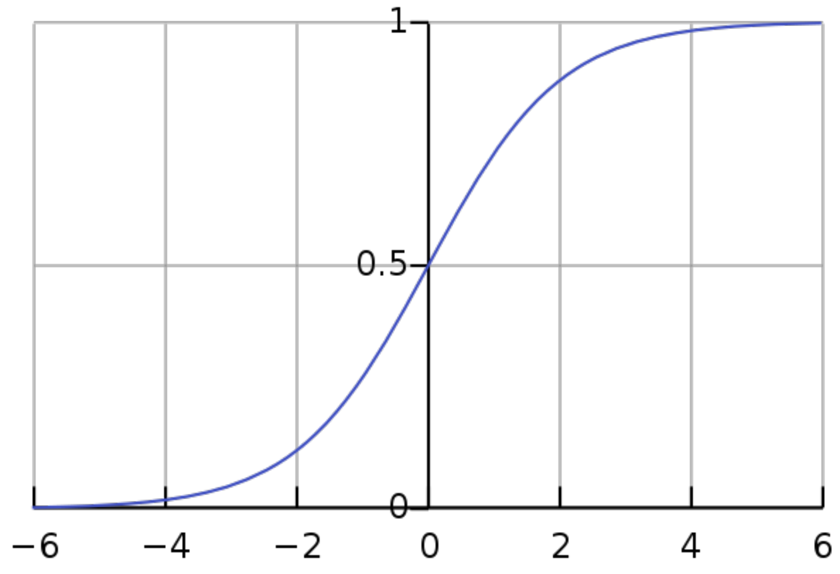
Derivative of the sigmoid

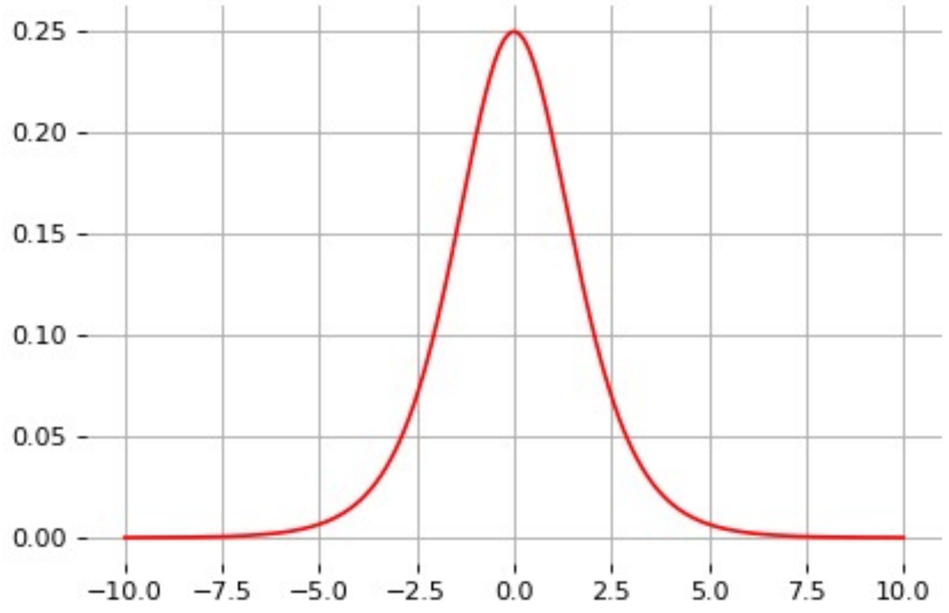
- $\sigma(x) = \frac{1}{1+e^{-x}}$

- $\frac{d\sigma(x)}{dx} = \sigma(x)\big(1 - \sigma(x)\big)$

- Hence, we get

$0 \leq \frac{d\sigma(x)}{dx} \leq \frac{1}{4}$ for all x.





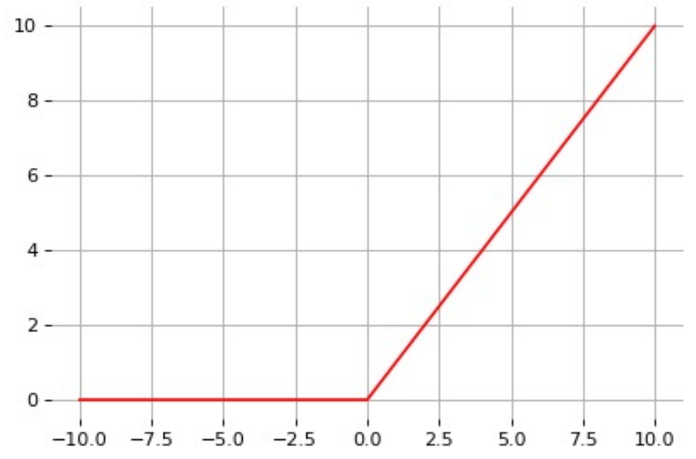
Bound on the derivative

- Assume that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.
- Then,

$$\left|\frac{d\hat{y}}{dw_1}\right| = \left|\frac{\partial\sigma(u_5)}{\partial u_5}\right| |w_3| \left|\frac{\partial\sigma(u_3)}{\partial u_3}\right| |w_2| \left|\frac{\partial\sigma(u_1)}{\partial u_1}\right| |x| \leq \left(\frac{1}{4}\right)^3 |x|$$

- This implies that the gradient $\frac{d\hat{y}}{dw_1}$ exponentially shrinks to zero as the number of layers in the network increases. This is the vanishing gradient problem.
- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note also the importance of a proper initialization scheme.

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

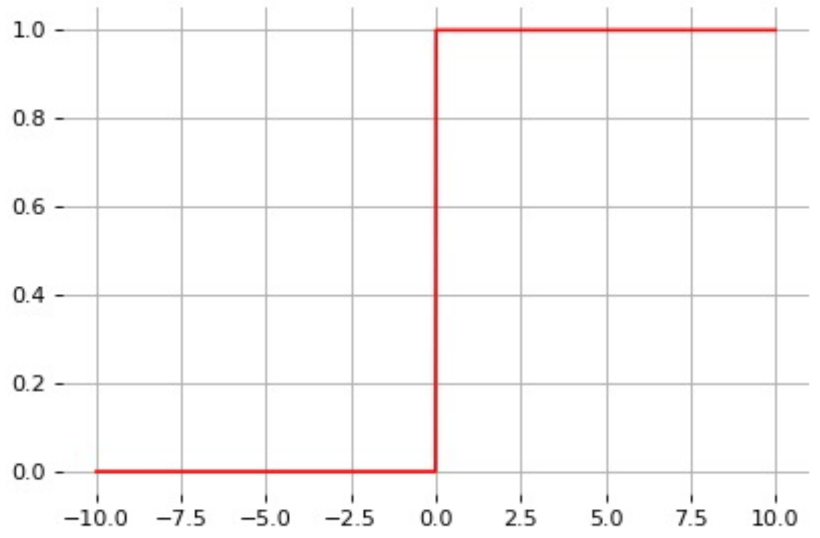


Derivative of the ReLU function

- Note that the derivative of the ReLU function is

$$\frac{d\mathrm{ReLU}(x)}{dx} = \begin{cases} 0 \text{ if } x \leq 0 \\ 1 \text{ if } x > 0 \end{cases}$$

- For $x = 0$, the derivative is undefined. In practice, it is set to zero.



Solving the gradient vanishing problem?

- Assume again that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.

- We get

$$\left|\frac{d\hat{y}}{dw_1}\right| = \left|\frac{\partial \sigma(u_5)}{\partial u_5}\right| |w_3| \left|\frac{\partial \sigma(u_3)}{\partial u_3}\right| |w_2| \left|\frac{\partial \sigma(u_1)}{\partial u_1}\right| |x| \leq |x|$$

- This solves the vanishing gradient problem, even for deep networks! (provided proper initialization)

- Note that:
    - The ReLU unit dies when its input is negative, which might block gradient descent.
    - This is actually a useful property to induce sparsity.
    - This issue can also be solved using leaky ReLUs, defined as

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x)$$

for small $\alpha > 0$

Conclusion

Conclusion

- Neural networks with several layers provide high capability for approximating multivariate functions
- Depth leads to vanishing gradient, which is an important issue
- Activation functions play an important role
- Theoretical results with neural network representation are in progress.