
WE NEED TO GO
DEEPER

CIS 522: Lecture 14

Depth and width
03/03/20

Penn Engineering

Are you here?

Yes

No

Puppy

Feedback / Logistics

-

Deep learning theory

Penn Engineering

Field of deep learning is very ad hoc

- New insights driven by trial and error + intuition
- Has been described as “alchemy”
- Often don’t understand *why things work*

Need both of the following:

- Science - run experiments on deep networks themselves
- Math - prove theorems about deep networks

Neural networks as functions

- Neural networks are functions from inputs to outputs
- World gives you a set of points (x,y)
- No closed-form function f(x) = y (e.g. f(image) = label)
- Find network N approximating unknown f - that is, N(x) ~ y

What functions can a network compute?

Different network architectures can compute different functions

Good network architecture = good at learning whatever function one is interested in

Universal Approximation

Penn Engineering

Universal Approximation Theorem

"A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasibly large and may fail to learn and generalize correctly."

– Ian Goodfellow

Universal Approximation Theorem

Let $\varphi : \mathbb{R} \to \mathbb{R}$ be a nonconstant, bounded, and continuous function. Let I_m denote the *m*-dimensional unit hypercube $[0,1]^m$. The space of real-valued continuous functions on I_m is denoted by $C(I_m)$. Then, given any $\varepsilon > 0$ and any function $f \in C(I_m)$, there exist an integer N, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $w_i \in \mathbb{R}^m$ for $i = 1, \ldots, N$, such that we may define:

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

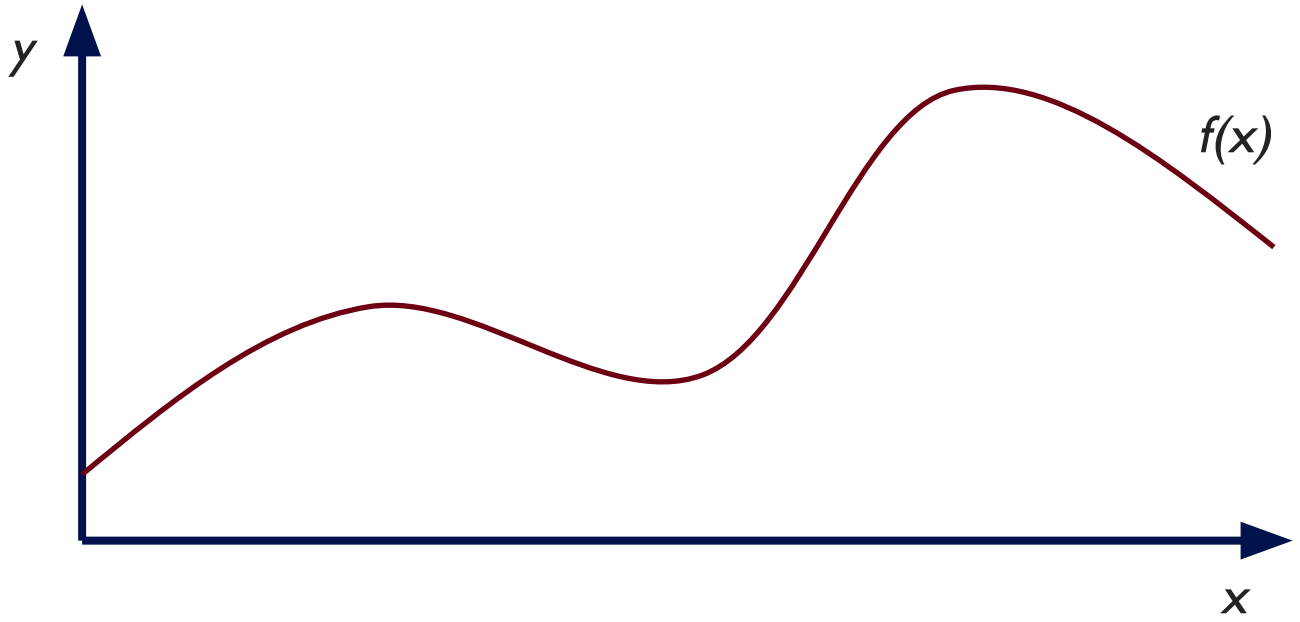
as an approximate realization of the function f; that is,

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. In other words, functions of the form $F(x)$ are dense in $C(I_m)$.

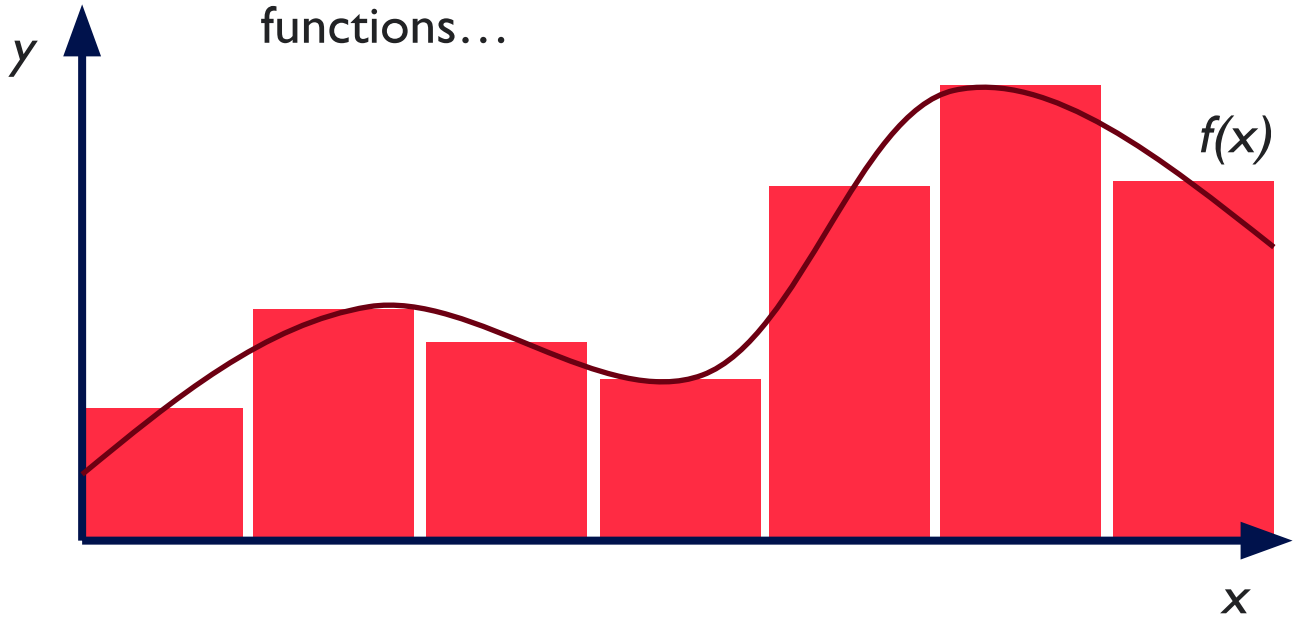
Cybenko's Construction

Suppose we want to approximate f(x)



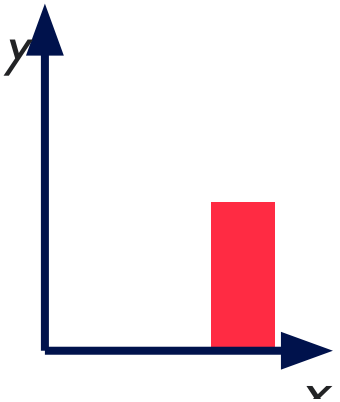
Cybenko's Construction

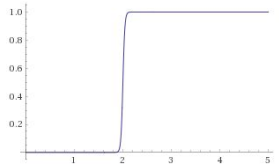
Let's approximate it as a piecewise function consisting of "indicator" functions…

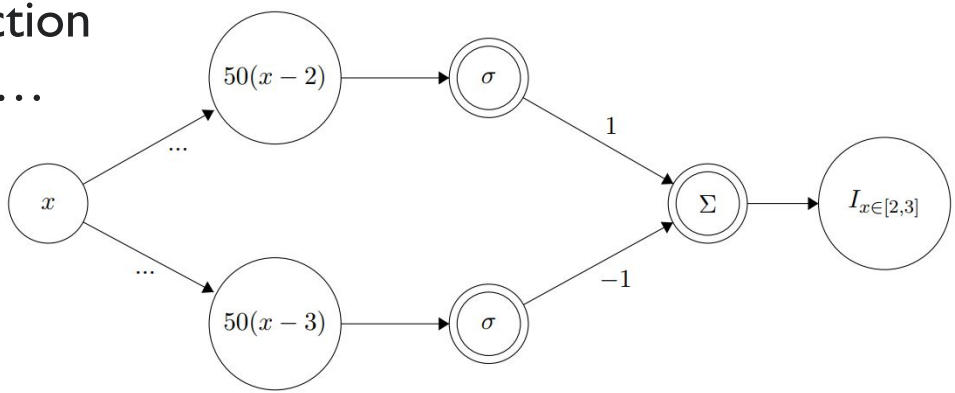


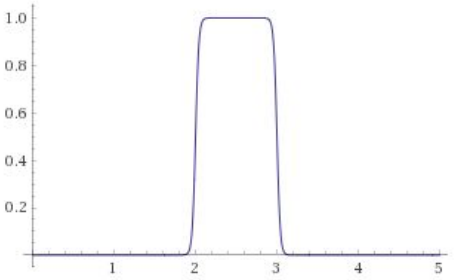
Cybenko's Construction

We can create a network that approximates this indicator function I[2, 3] like so…

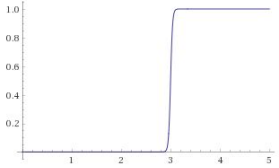






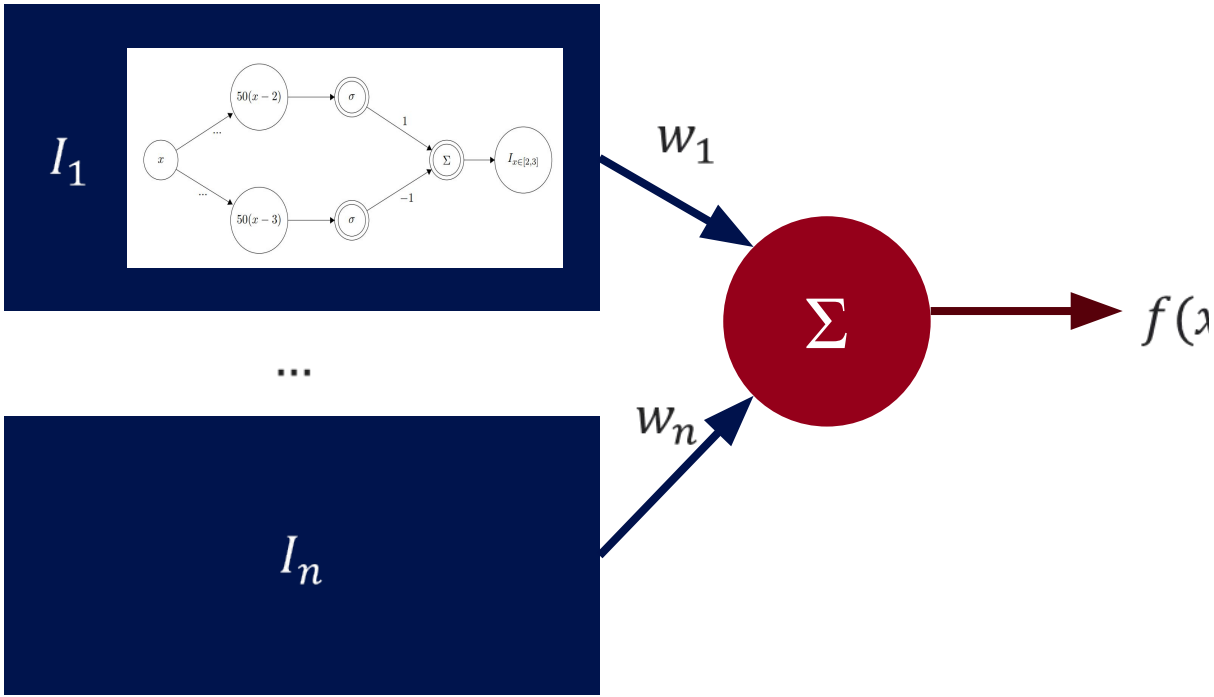


$$y = \frac{1}{1+e^{-50(x-2)}} - \frac{1}{1+e^{-50(x-3)}}$$

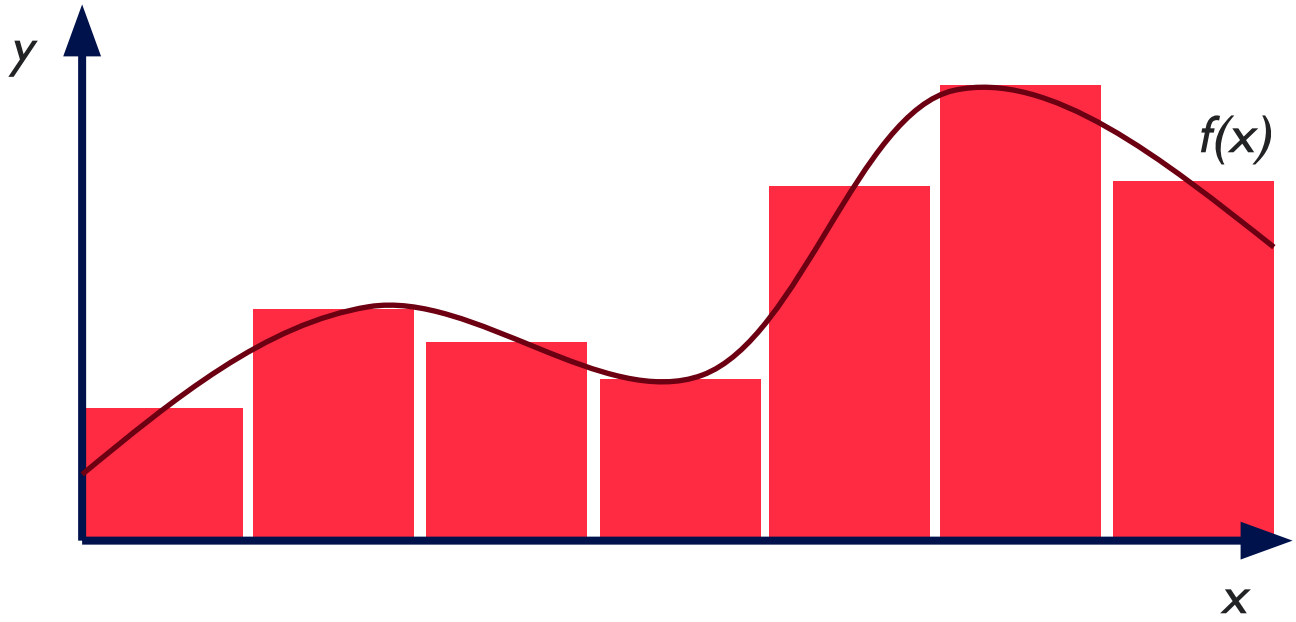


Cybenko's Construction

We can use these as building blocks for our universal approximator …



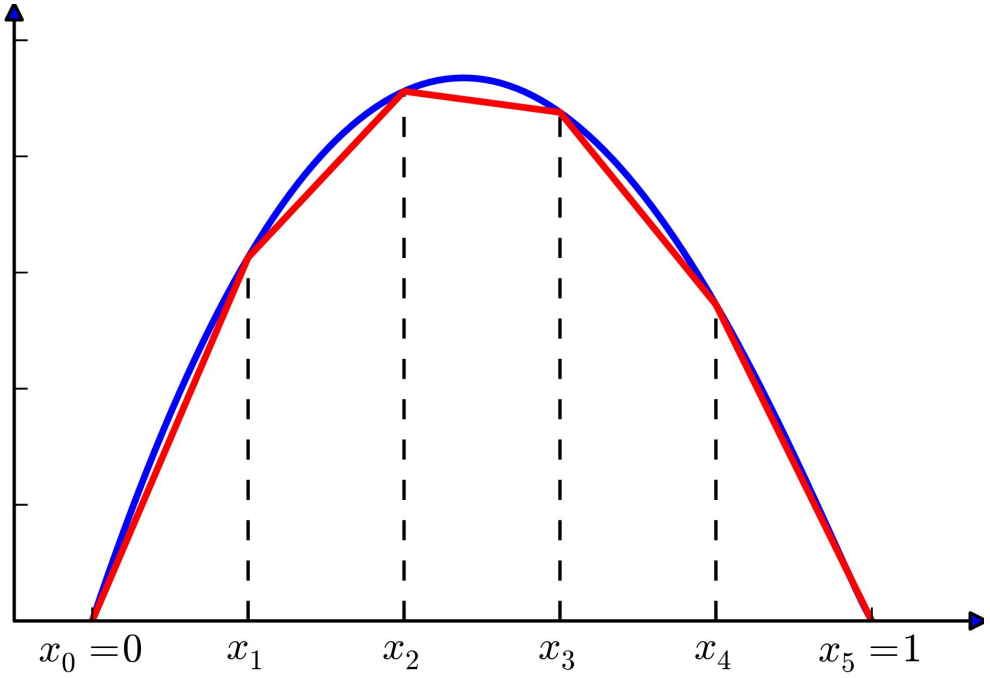
Cybenko's Construction



Expressivity

Penn Engineering

Universal approximation by ReLUs

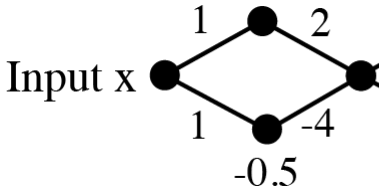


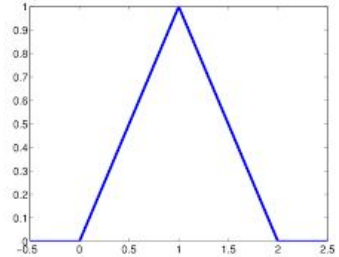
line segments for a shallow network?

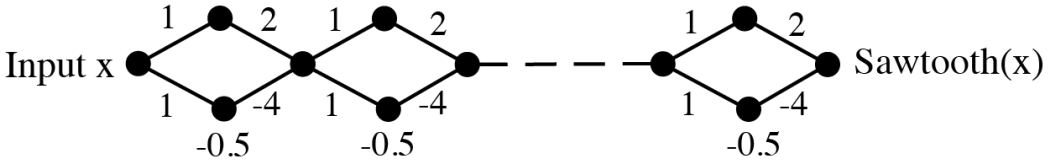
- Each neuron adds one “kink” in the function
- One layer: all the neurons add up
- # line segments = # neurons + 1

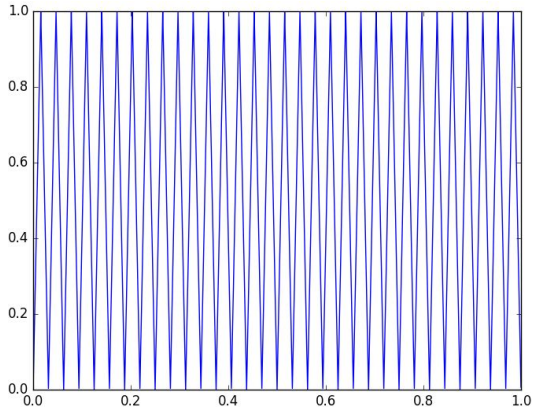
line segments for a deep network?

- Can grow exponentially in depth!









Conclusion

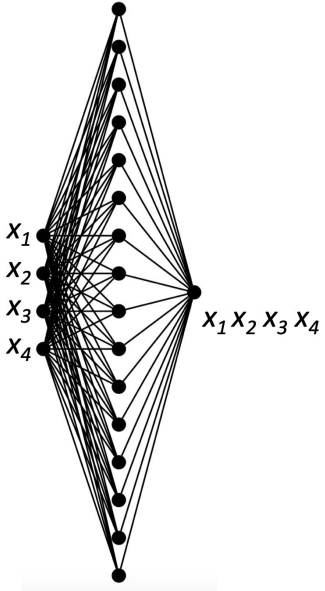
Sawtooth function with 2^d "teeth" can be written using a width-2, depth-d network.

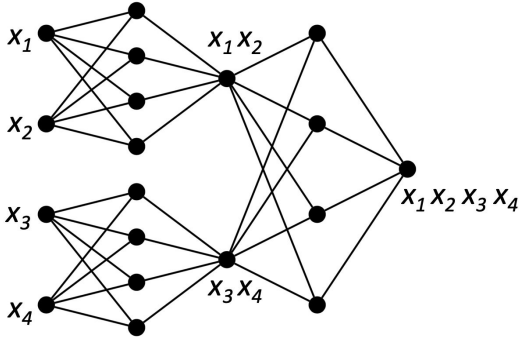
Since it has ~2^d line segments, would need ~2^d neurons if we were using a depth-one network.

Other expressivity gaps b/w deep and shallow

Some functions take exponentially more neurons to write down with a one-layer network than a deep one

2^n neurons to multiply n inputs with one layer





O(n) neurons to multiply n inputs with log(n) layers

Inductive biases

Penn Engineering

Distillation

Experiment by Ba & Caruana:

- 2 networks, deep one and shallow one
- Train the deep net on a classification task - **succeeds**
- Train the shallow net on the task - **fails**
- Train the shallow net on the output of the deep one - **succeeds!**

Distillation

How can this work?

- Shallow net sees the full output vector of deep net predictions - not just 1-hot
- Also can be trained on more examples than the original dataset

Distillation

Takeaways

- Deep net better at learning these tasks
- But shallow net can still *express* the function - just has troubles learning it
- Maybe the functions we are learning are ones that *both deep and shallow nets can express*, but deeper networks find *easier to learn*

ConvNets

- Weight-sharing in ConvNets reduces expressive power
- Could replace convolutional layers by (larger) fully-connected layers - if knew the right weights
- Would be able to express same function

Convolutional

$$\begin{pmatrix} k1 & k2 \\ k3 & k4 \end{pmatrix} \begin{pmatrix} x1 & x2 & x3 \\ x4 & x5 & x6 \\ x7 & x8 & x9 \end{pmatrix}$$

Fully-connected

$$\begin{pmatrix} k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 & 0 \\ 0 & k1 & k2 & 0 & k3 & k4 & 0 & 0 & 0 \\ 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 & 0 \\ 0 & 0 & 0 & 0 & k1 & k2 & 0 & k3 & k4 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \\ x4 \\ x5 \\ x6 \\ x7 \\ x8 \\ x9 \end{pmatrix}$$

ConvNets

- ConvNets good because weight-sharing biases the net to learn functions that are *translation-invariant*
- It's functions that are *learned easily*, not the functions that are *expressible*

ResNets

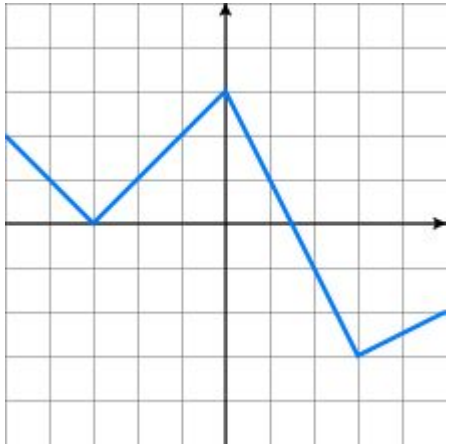
- Adding skip connections between layers doesn't increase expressivity - same as (slightly larger) fully connected nets
- Could just increase the size of the layers to keep information from lower layers
- But can make learning easier

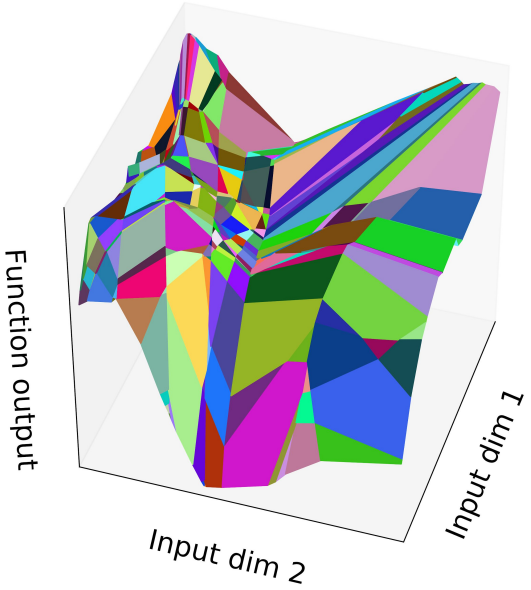
Learnability

Penn Engineering

Linear regions

ReLU networks are *piecewise linear* functions





How many linear regions?

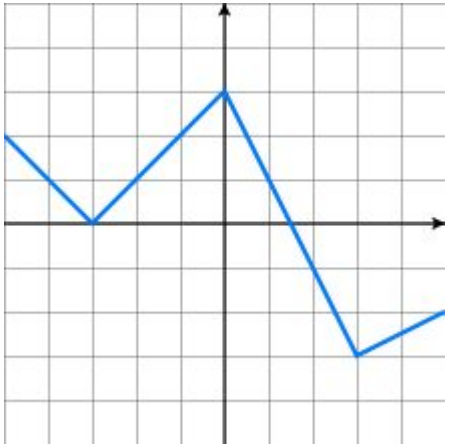
Each ReLU has two options: OFF or ON

Each pattern of OFF/ON defines a linear region

But not every pattern exists

Max # regions: $2^{\#\text{ neurons}}$

Min # regions: just one!



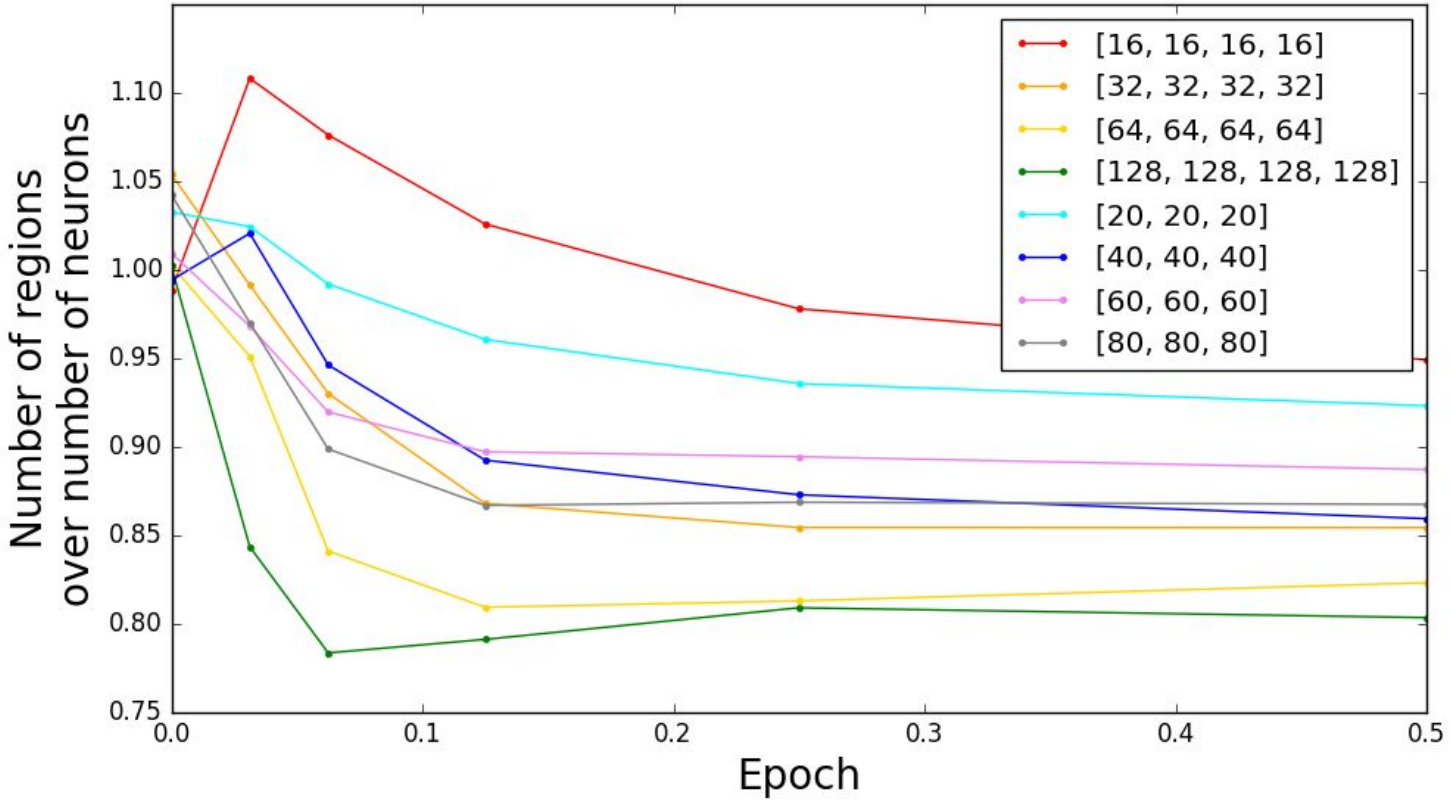
How many linear regions in practice?

Theorem.

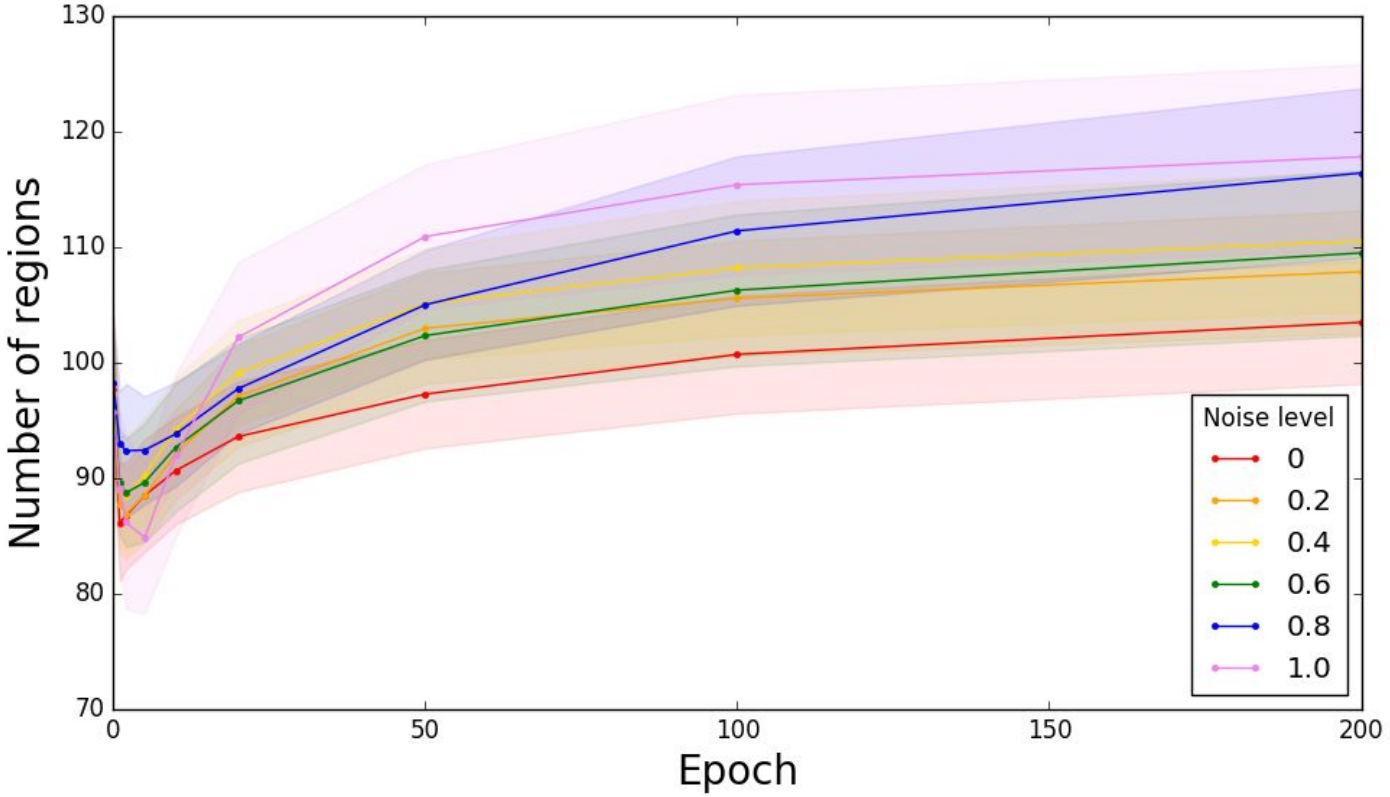
The number of linear regions defined a ReLU network with one input dimension is ~ # neurons in the network for typical settings of the weights and biases.

- And it's much smaller than $2^{\#\text{ neurons}}$
- And it doesn't depend on the depth!

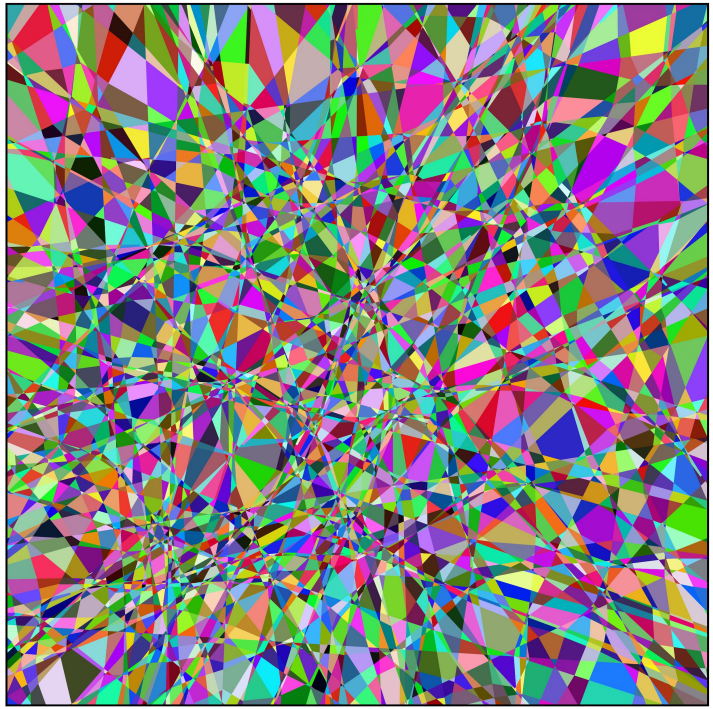
How many linear regions in practice?



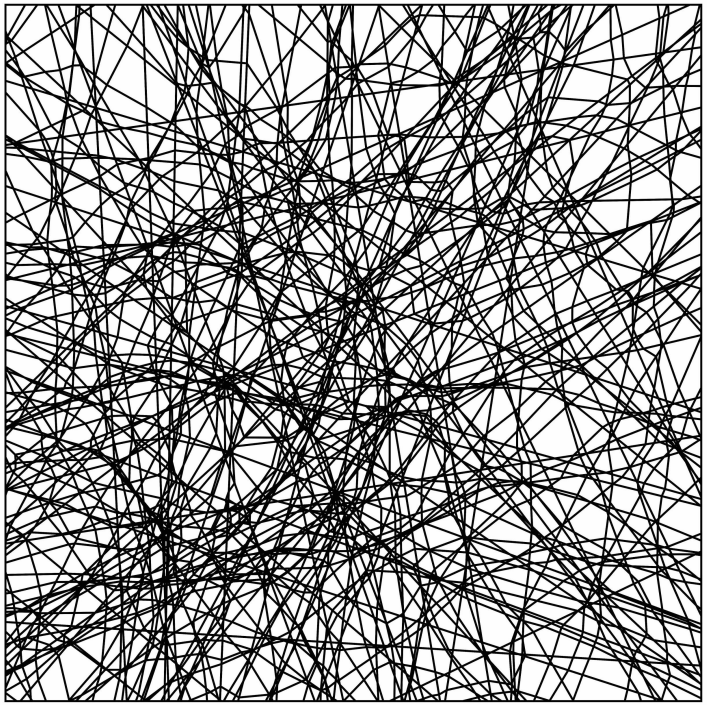
How many linear regions in practice?



Boundaries of linear regions



Boundaries of linear regions



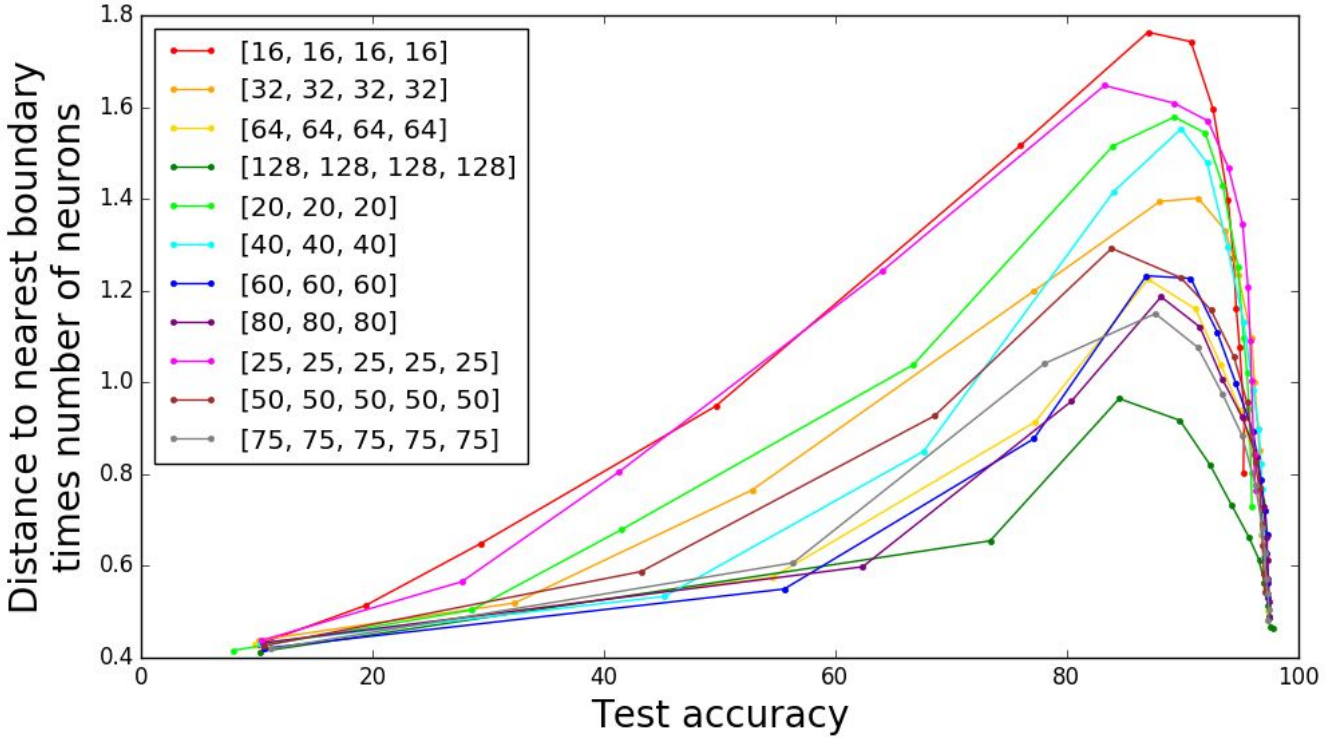
Distance to the nearest different linear region

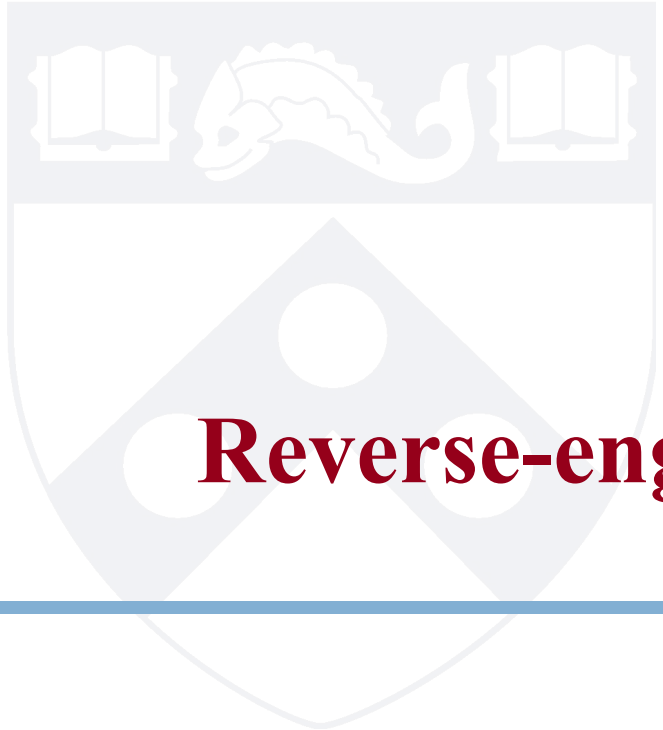
Theorem.

For any input x, the expected distance to the nearest different region is ~1/# neurons.

- Within a ball of radius 1/# neurons around x, the network is just a linear function
- Small adversarial perturbations are actually hard

Distance to the nearest region boundary





Reverse-engineering deep networks

Penn Engineering

Reverse-engineering deep nets

- Have a network - unknown but can ask queries
- “What is output N(x) for this x?”
- Want the structure and weights of the network
- Many deep networks have public output, private structure/weights
- Possibility of recovering confidential data if network known

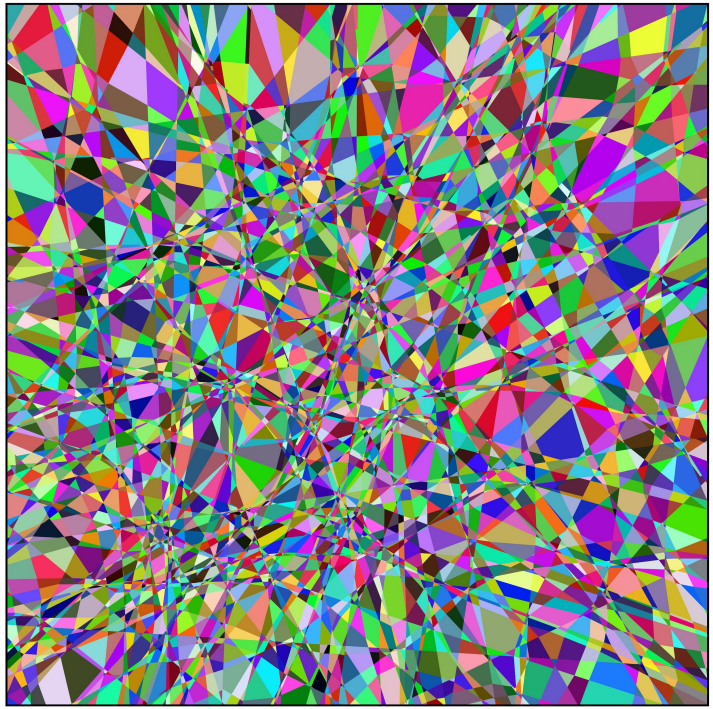
How good could this be?

- Could we recover the exact structure and weights of a ReLU network?

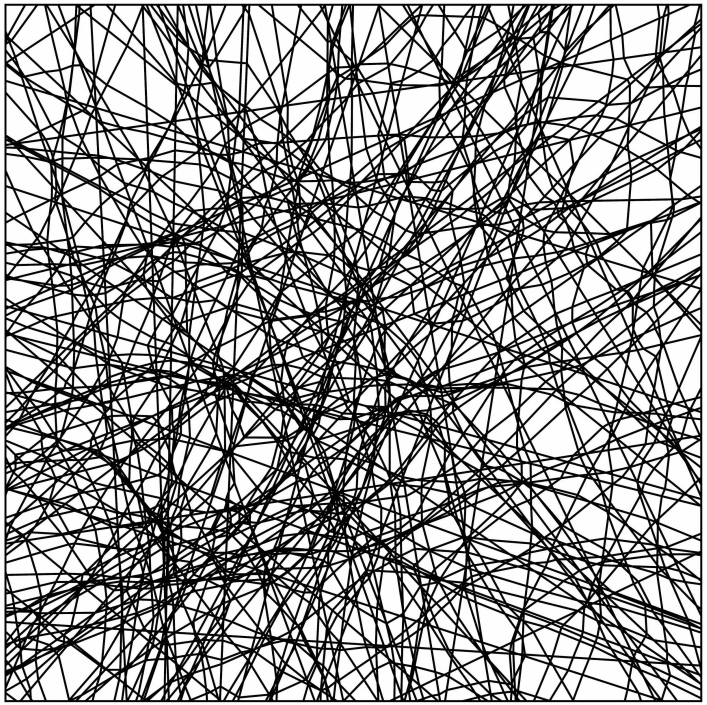
How good could this be?

- Could we recover the exact structure and weights of a ReLU network?
- Two ReLU networks can be indistinguishable!
  - For example, swapping neurons
  - And scaling any neuron by a positive constant:
    - ReLU(cwx + cb) = c ReLU (wx + b)
    - so (w'/c) ReLU(cwx + cb) = w' ReLU(wx + b)

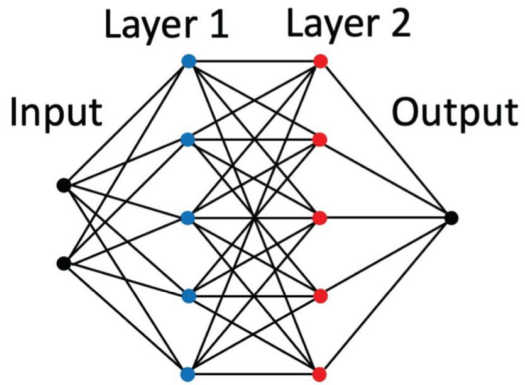
Boundaries of linear regions

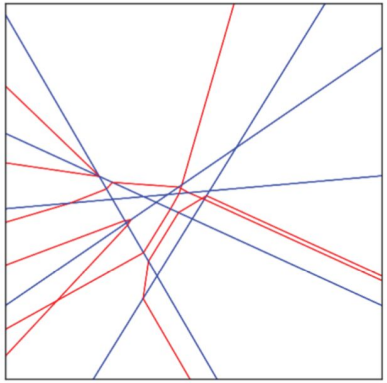


Boundaries of linear regions



Boundaries of linear regions





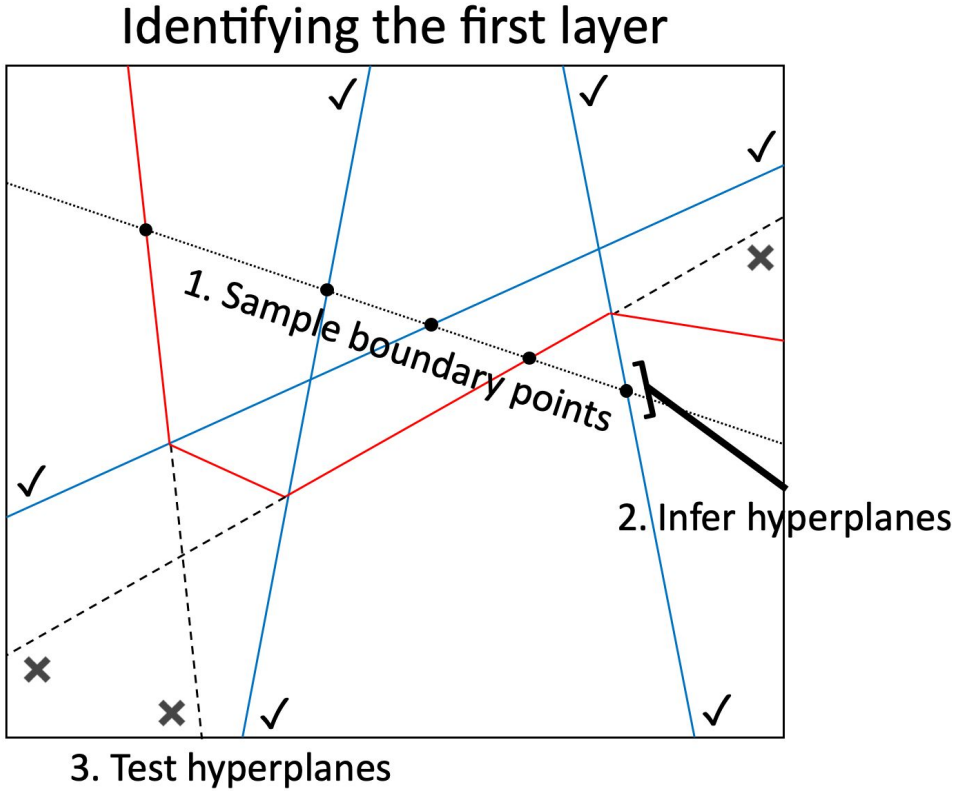
- Component of boundary for each neuron in network
- Occurs where ReLU(neuron) = 0
- Neurons in layer 1 = lines (hyperplanes in higher dim)
- Deeper neurons = bent lines (bent hyperplanes)

If you knew the boundaries

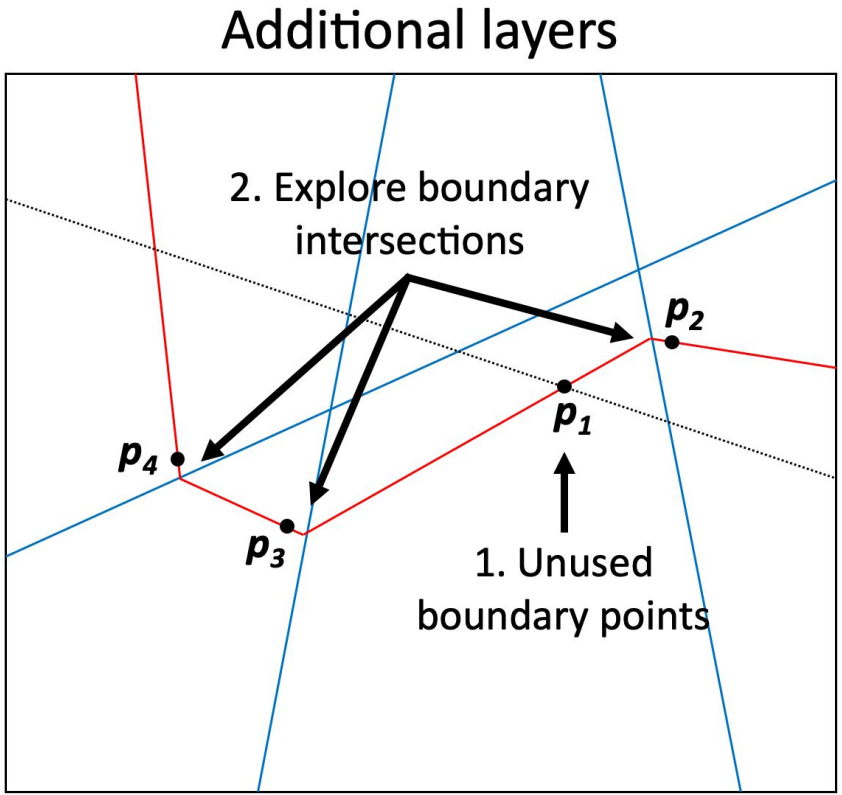
Theorem.

Suppose that for a given neural network, (i) the set of boundaries between linear regions is known, and (ii) the boundaries corresponding to every two adjacent neurons intersect. Then, it is possible to recover the complete structure and weights of the network, up to permutation and scaling (except for a measure-zero set of networks).

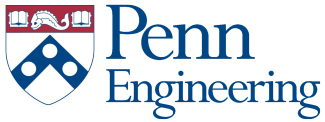
If you can just query the network



If you can just query the network



Infinitely wide networks



What would happen in an infinitely wide net?

- No real need to learn “features” - features already there by chance
- Goal: pick out the right features to use
- Most weights wouldn’t change

A way to (sort of) train an infinitely wide net

Neural tangent kernel:

$$\Delta\mathcal{N}(x) \;=\; -\lambda\,\langle K_{\mathcal{N}}(x,\cdot),\nabla\mathcal{L}(\cdot)\rangle \;=\; -\frac{\lambda}{|\mathcal{B}|}\sum_{j=1}^{|\mathcal{B}|}K_{\mathcal{N}}(x,x_j)\frac{\partial\mathcal{L}}{\partial\mathcal{N}}(x_j,y_j). \tag{1}$$

Here $\mathcal{B}=\{(x_1,y_1),\ldots,(x_{|\mathcal{B}|},y_{|\mathcal{B}|})\}$ is the current batch, the inner product is the empirical ℓ_2 inner product over $\mathcal{B}$, and $K_{\mathcal{N}}$ is the *neural tangent kernel* (NTK):

$$K_{\mathcal{N}}(x,x') \;=\; \sum_{p=1}^{P}\frac{\partial\mathcal{N}}{\partial\theta_p}(x)\frac{\partial\mathcal{N}}{\partial\theta_p}(x').$$

A way to (sort of) train an infinitely wide net

Neural tangent kernel:

$$\Delta\mathcal{N}(x) \;=\; -\lambda\,\langle K_{\mathcal{N}}(x,\cdot),\nabla\mathcal{L}(\cdot)\rangle \;=\; -\frac{\lambda}{|\mathcal{B}|}\sum_{j=1}^{|\mathcal{B}|}K_{\mathcal{N}}(x,x_j)\frac{\partial\mathcal{L}}{\partial\mathcal{N}}(x_j,y_j). \tag{1}$$

Here $\mathcal{B}=\{(x_1,y_1),\ldots,(x_{|\mathcal{B}|},y_{|\mathcal{B}|})\}$ is the current batch, the inner product is the empirical ℓ_2 inner product over $\mathcal{B}$, and $K_{\mathcal{N}}$ is the *neural tangent kernel* (NTK):

$$K_{\mathcal{N}}(x,x') \;=\; \sum_{p=1}^{P}\frac{\partial\mathcal{N}}{\partial\theta_p}(x)\frac{\partial\mathcal{N}}{\partial\theta_p}(x').$$

Key point: infinite-width limit, K converges to a Gaussian process - can fit without gradient descent

A way to (sort of) train an infinitely wide net

Algorithm: fit the “neural tangent kernel” to the data by computing a covariance matrix (takes quadratic time to compute)

Performs pretty well, but not as well as (finite-width) neural nets trained by gradient descent

Takeaways

Penn Engineering

Takeaways

- 1-layer networks can approximate any function
- Deeper networks can approximate some functions using fewer parameters
- But learning those good deep networks may be hard
- Deep networks may be biased towards learning "good" functions
  - Will discuss more in next lecture on generalization!