# Feature Learning

Instructor: Hui Guan

Slides adapted from: https://github.com/jermwatt/machine_learning_refined

# Outline

- What is Feature Learning ←
- Universal approximator ← tree, NN, polynomial
- Naïve Cross-Validation
- Cross-Validation via Boosting
- Cross-Validation via Regularization
- Testing data
- Which Universal Approximator works the Best
- Bagging ←
- When Feature Learning Fails ←

- We saw how supervised and unsupervised learners alike can be extended to perform nonlinear learning via the use of nonlinear functions (or feature transformations) that we engineered ourselves by visually examining data.
- Here we detail the fundamental tools and principles of *feature learning* (or *automatic feature engineering*) that allow us to **automate this task and *learn* proper features from the data itself**.

More specifically, expressing a general nonlinear model for regression and two-class classification as a weighted sum of B nonlinear functions of our input as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_B\left(\mathbf{x}\right) w_B$$

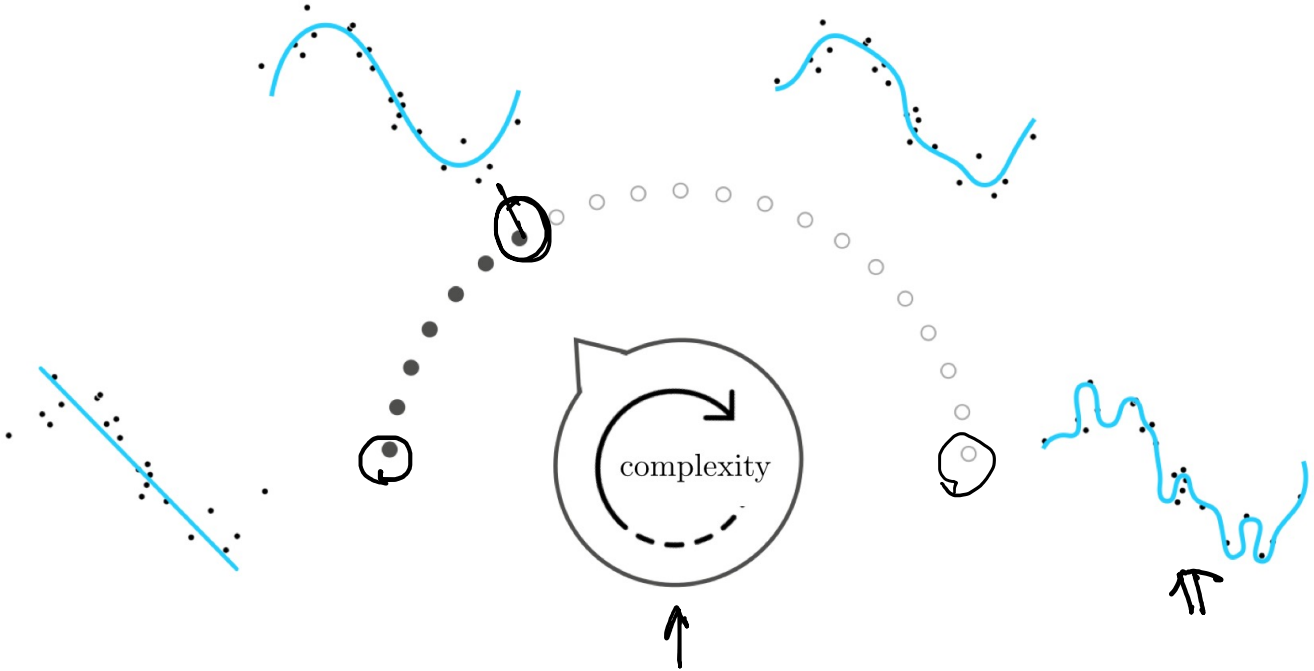
we will discuss how to:

- choose the form of the nonlinear transformations $f_1$ through $f_B$
- the number B of them employed
- and, how the parameters in $\Theta$ (including $w_0$ through $w_B$) are tuned, ***automatically*** and for ***any dataset***

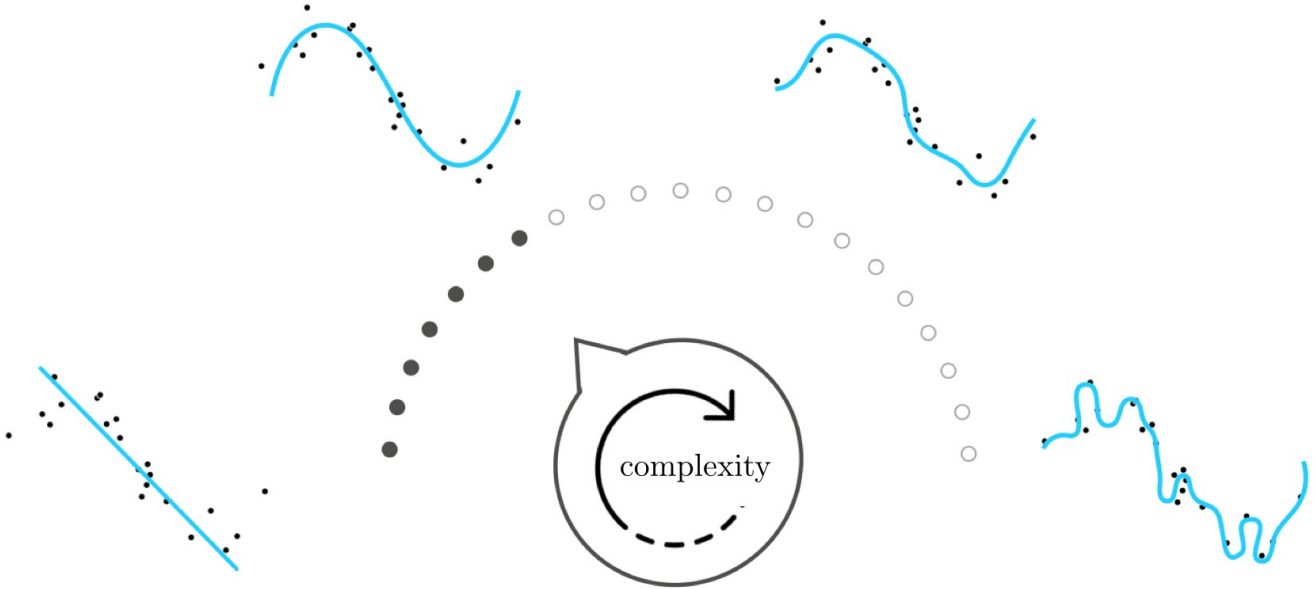
# The complexity dial metaphor of feature learning

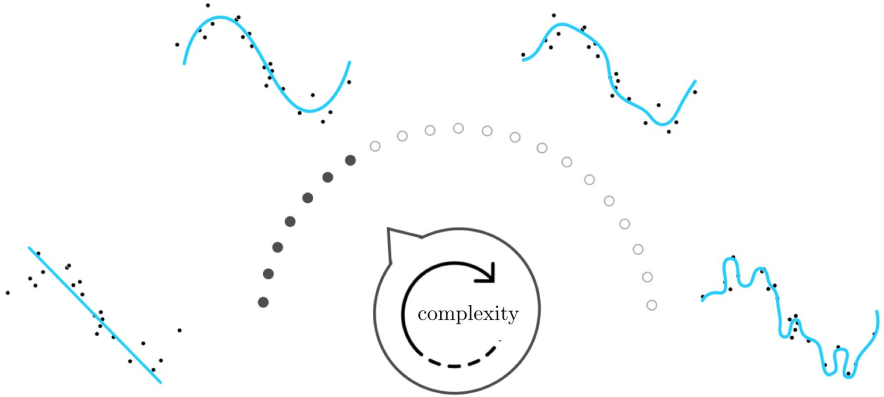
We can think about feature learning metaphorically as

1) the *construction* of and 2) the *automatic setting* of a *complexity dial*, like the one shown below for a simple nonlinear regression dataset.



This 'complexity dial' conceptualization visually depicts the challenge of feature learning at a high level as a dial that must be built and automatically tuned to determine the **appropriate amount of model complexity** needed to represent the data.






- Setting this complexity dial all the way to the left corresponds to choosing a simple form for the nonlinear model that results in a representation of low complexity (e.g., a linear model)
- If turned too far to the right the resulting model will be too complex (or too 'wiggly') with respect to the training data.
- When set 'just right' the resulting model represents the data - as the underlying phenomenon generating it - very well.

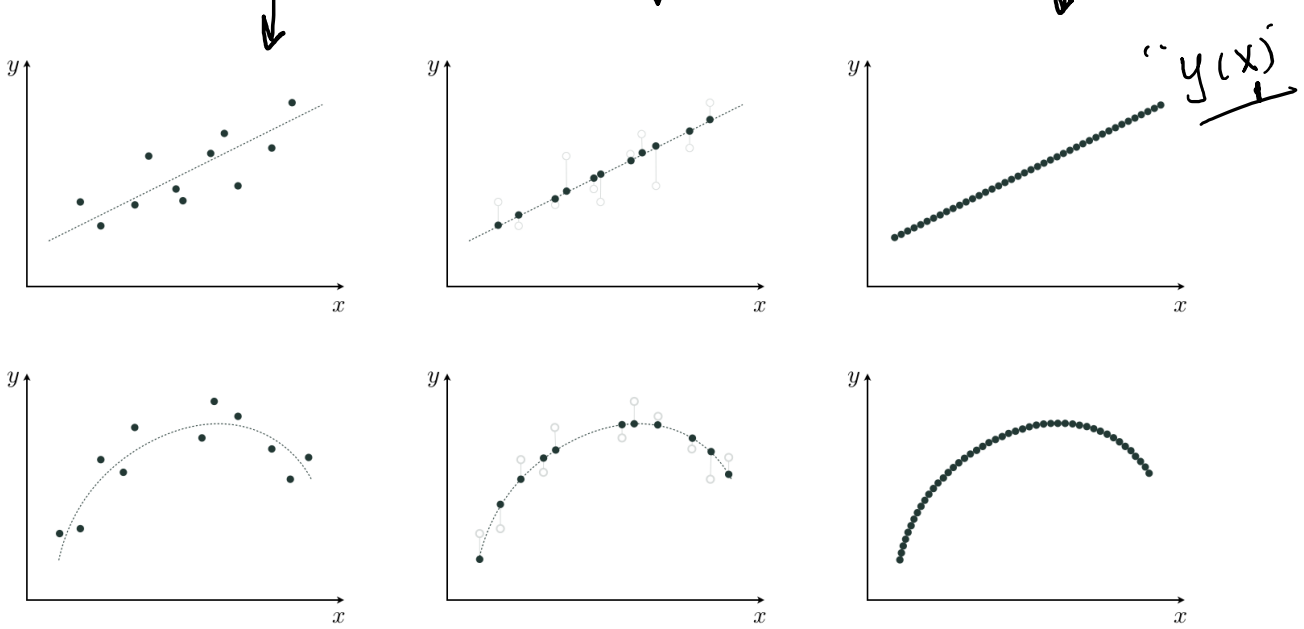
# 11.2 Universal approximators

# Perfect data

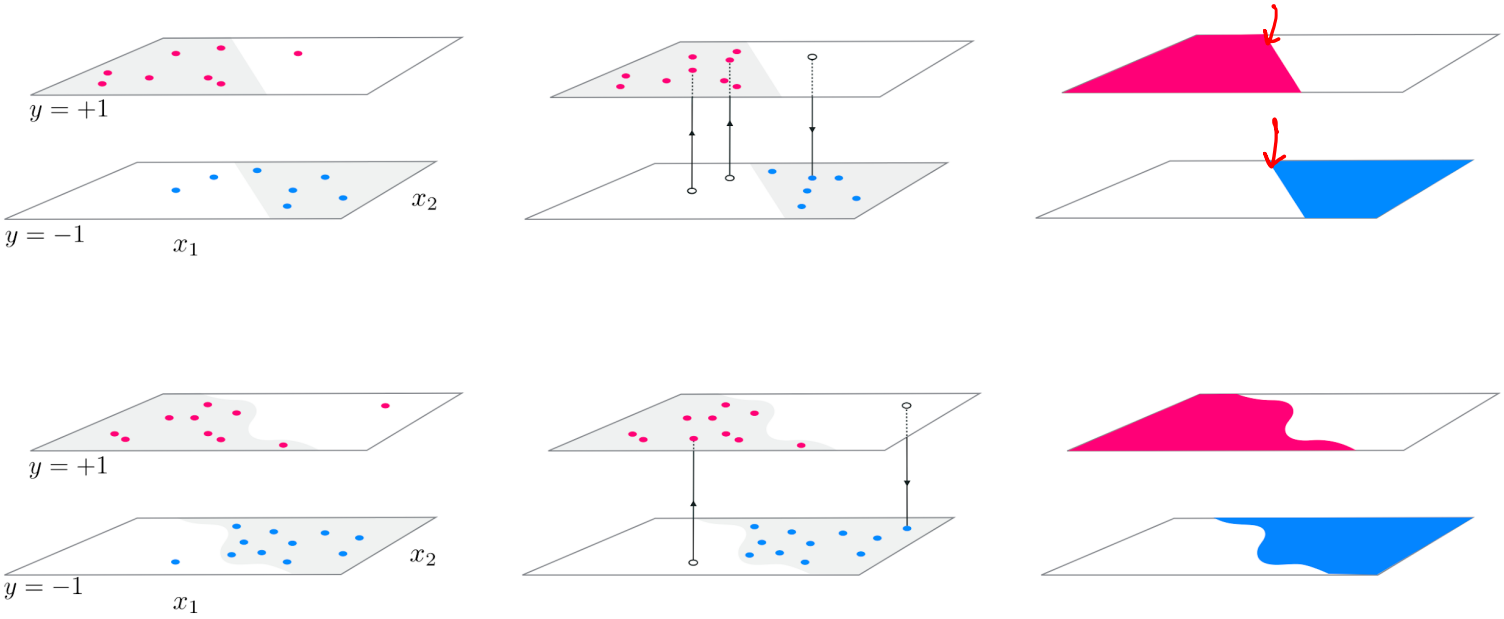
A *__perfect__* dataset for regression or two-class classification has two important (albeit, unrealistic) characteristics:

- It is completely noiseless
- It is infinitely large

When we have unfettered access to perfect data, perfect features can be *learned* automatically by combining elements from a set of basic feature transformations, known as *universal approximators*, the most popular of which include *fixed-shape approximators*, *neural networks*, and *trees*.




(top left) A realistic linear regression dataset.
(top middle) The noiseless version.
(top right) The perfect version of the same data.
(bottom left) A realistic nonlinear regression dataset.
(bottom middle) The noiseless version.
(bottom right) The perfect version, that is noiseless and infinitely large.




(top left) A realistic linear two-class classification dataset.
(top middle) The noiseless version.
(top right) The perfect version of the same data.
(bottom left) A realistic nonlinear two-class classification dataset.
(bottom middle) The noiseless version.
(bottom right) The perfect version, that is noiseless and infinitely large.

$$y(x) \approx f_1(x) + f_2(x) \cdots + f_B(x)$$

- A perfect regression or two-class classification dataset is a continuous (or piece-wise continuous) function with unknown equation.
- Because of this we will refer to our perfect data using the function notation $y(\mathbf{x})$, meaning that the data pair defined at input $\mathbf{x}$ can be written as either $(\mathbf{x}, y(\mathbf{x}))$ or likewise $(\mathbf{x}, y)$
- It is important to bear in mind that the function notation $y(\mathbf{x})$ does not imply that we have knowledge of a closed form *formula* relating the input/output pairs of a perfect dataset, we do not!

# The spanning set analogy for universal approximation

## Linear combinations of vectors

$f_i \in \mathbb{R}^N$

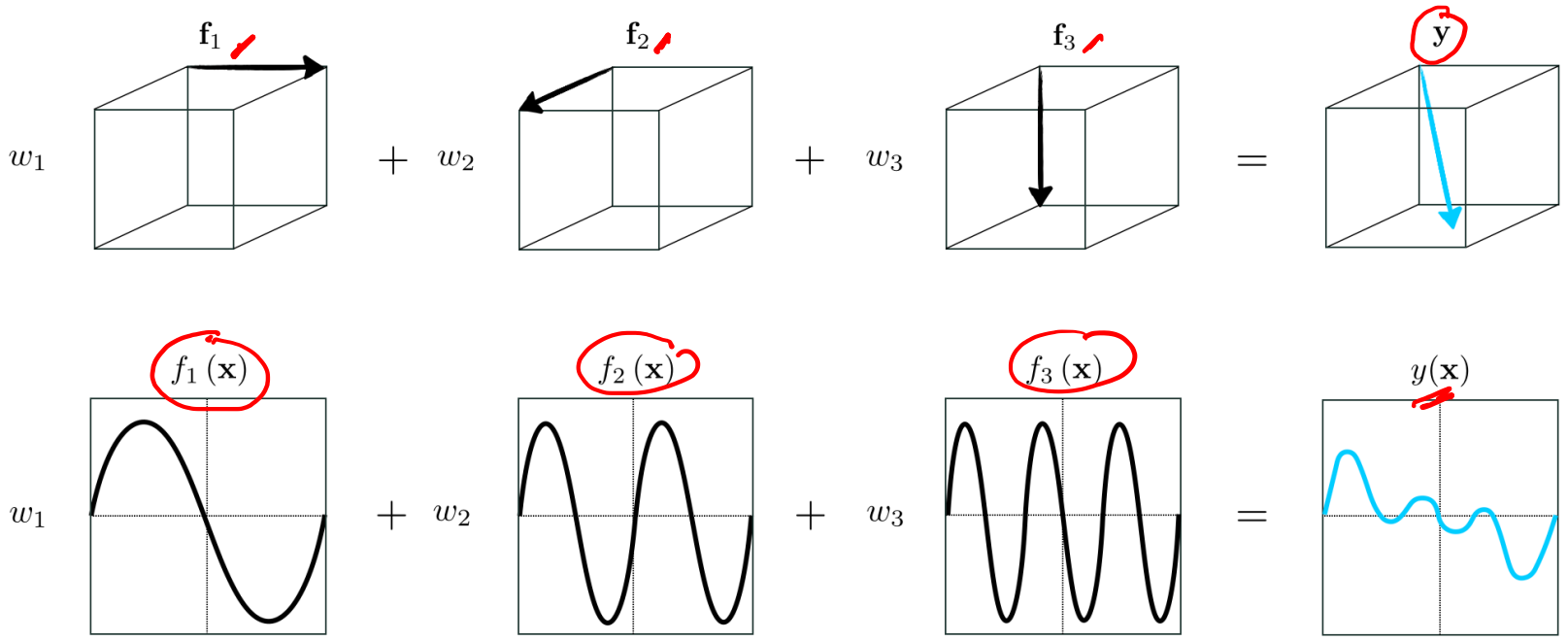
- Assume we have a set of B vectors $\{\mathbf{f}_1, \mathbf{f}_2, \ldots, \mathbf{f}_B\}$, each having dimension N.
- We call this a *spanning* set of vectors.
- Given a particular set of weights $w_1$ through $w_B$, the linear combination below defines a new N dimensional vector $\mathbf{y}$

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B = \mathbf{y} \in \mathbb{R}^N$$

Linear combinations of functions

Similarly, given a spanning set of B nonlinear functions $\{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_B(\mathbf{x})\}$ (where the input is $\mathbf{x}$) is N dimensional and output is scalar) and a corresponding set of weights, the linear combination below defines a new function $y(\mathbf{x})$

$$w_0 + f_1(\mathbf{x})\, w_1 + f_2(\mathbf{x})\, w_2 + \cdots + f_B(\mathbf{x})\, w_B = y(\mathbf{x})$$




A particular linear combination of three vectors (top) and three functions (bottom)

# Capacity of spanning sets

$f_1 \cdots f_B \in \mathbb{R}^N$

① $B$ : $B < N$

② $B \geq N$ : $f_B = f_1 + \cdots f$

## Vector approximation problem

- Computing the vector $\mathbf{y}$ in $\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B \approx \mathbf{y}$, for a *given* set of weights $w_1$ through $w_B$, is a trivial affair.
- The inverse problem, i.e., finding the weights given $\mathbf{y}$, is slightly more challenging.
- Stated algebraically, we want to find the weights $w_1$ through $w_B$ such that the following holds as well as possible.

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B \approx \mathbf{y}$$

How well the *vector approximation* $\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B \approx \mathbf{y}$ holds depends on three crucial and interrelated factors:

1. The diversity (i.e., linear independence) of the spanning vectors

2. The number B of them used (in general the larger we make B the better)

3. How well we tune the weights $w_1$ through $w_B$ via minimization of an appropriate cost

- Factors (1) and (2) determine a spanning set's *rank* or *capacity*, that is a measure for the range of vectors $\mathbf{y}$ we can possibly represent with such a spanning set.
- A spanning set with a *low capacity*, that is one consisting of a non-diverse and/or a small number of spanning vectors can approximate only a tiny fraction of those present in the entire vector space.
- A spanning set with a *high capacity* can represent a broader swath of the space.

Function approximation problem

We can try to find the weights $w_1$ through $w_B$ , for a given $y(\mathbf{x})$ , such that the following holds as well as possible.

$$w_0 + f_1(\mathbf{x})\, w_1 + f_2(\mathbf{x})\, w_2 + \cdots + f_B(\mathbf{x})\, w_B \approx y(\mathbf{x})$$

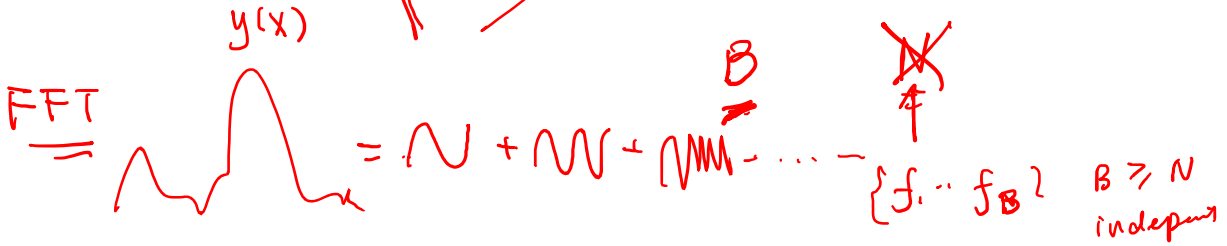
As with the vector case, how well this *function approximation* holds depends on three crucial and interrelated factors:

1. The diversity of the spanning functions
2. The number B of them functions used

} capacity

3. How well we tune the weights $w_0$ through $w_B$ (as well as any parameters internal to our nonlinear functions) via minimization of an appropriate cost.

} optimization



- In analogy to the vector case, factors (1) and (2) determine the *capacity* of a spanning set of functions.
- A *low capacity* spanning set that uses a non-diverse and/or small array of nonlinear functions is only capable of representing a small range of nonlinear functions.
- A spanning set with a *high capacity* can represent a wider swath of functions.

# Universal approximation

- Consider the vector approximation problem

$$\mathbf{f}_1 w_1 + \mathbf{f}_2 w_2 + \cdots + \mathbf{f}_B w_B \approx \mathbf{y}$$

- If $B \geq N$ and at least N of the vectors are linearly independent, then our spanning set has maximal capacity and we can therefore approximate *every* N dimensional vector $\mathbf{y}$ to *any* given precision.
- **Such a set of spanning vectors can approximate (or in this case perfectly represent) every vector *universally*.**

- Now consider the function approximation problem

$$w_0 + f_1(\mathbf{x})\, w_1 + f_2(\mathbf{x})\, w_2 + \cdots + f_B(\mathbf{x})\, w_B \approx y(\mathbf{x})$$

- If we choose the right kind of spanning functions, then our spanning set has maximal capacity and we can therefore approximate *__every__* function $y(\mathbf{x})$ to *__any__* given precision.
- **Such a set of spanning functions, of which there are infinitely many varieties, can approximate every function *universally*, and is thus often referred to as a *universal approximator*.**

Note: one difference between the vector and the function regime of universal approximation is that, with the latter, **we may need infinitely many spanning functions to be able to approximate a given function to an arbitrary precision** (whereas with the former it is always sufficient to set B greater than or equal to N).

# Popular universal approximators

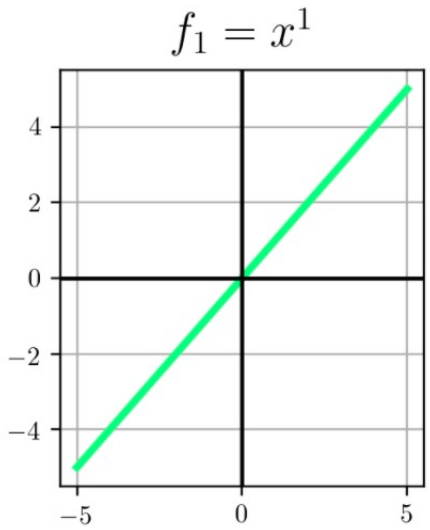
- In theory there are infinitely many universal approximators.
- However for the purposes of organization, convention, as well as a variety of technical matters universal approximators used in machine learning are often lumped into three main categories referred to as *fixed-shape approximators*, *neural networks*, and *trees*.
- Each of these popular families has its own unique practical strengths and weaknesses as a universal approximator, a wide range of technical details to explore, and conventions of usage, which we explore in Chapters 12-14.

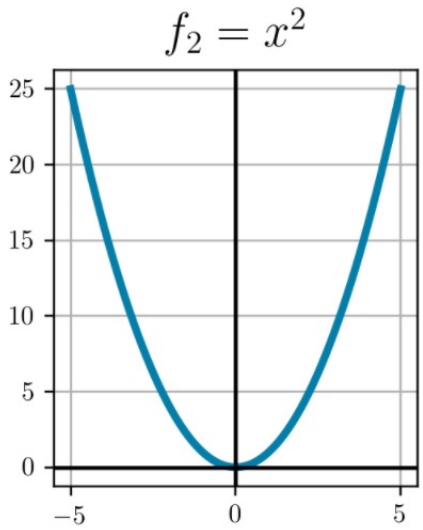
# Example: The fixed-shape family of universal approximators

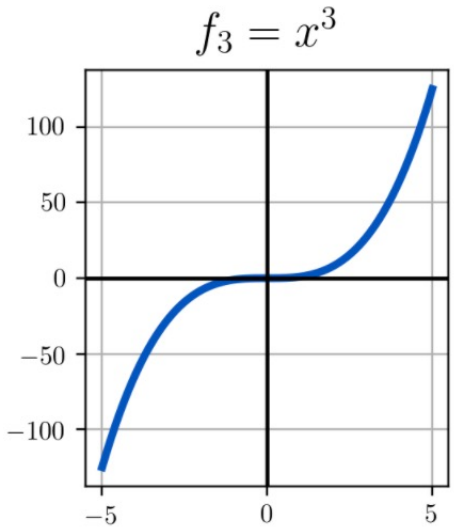
- The family of *fixed-shape* functions **consists of groups of nonlinear functions with no internal parameters, giving each a "fixed" shape**.
- Polynomials are a popular sub-family:

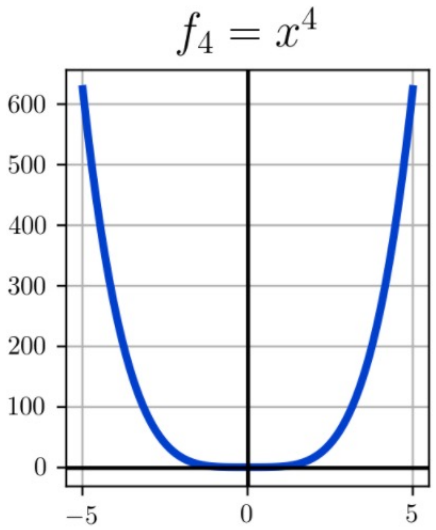
$$f_1(x) = x, \quad f_2(x) = x^2, \quad f_3(x) = x^3, \quad f_D(x) = x^D$$

- A combination of the first D units from this sub-family is often referred to as a *degree D* polynomial.
- Polynomials are *naturally ordered* by their degree.


$f_1 = x^1$
4
2
0
−2
−4
−5
0
5


$f_2 = x^2$
25
20
15
10
5
0
−5
0
5


$f_3 = x^3$
100
50
0
−50
−100
−5
0
5

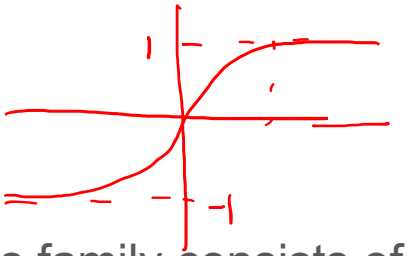

$f_4 = x^4$
600
500
400
300
200
100
0
−5
0
5

With two inputs $x_1$ and $x_2$, a general degree D polynomial unit takes the analogous form below where p and q are non-negative integers and $p + q \leq D$

$$f_b(x_1, x_2) = x_1^p x_2^q$$

Classically, **a degree D polynomial is a linear combination of all such units**. This definition directly generalizes to $N > 2$ dimensional input.

# Example: The neural network family of universal approximators



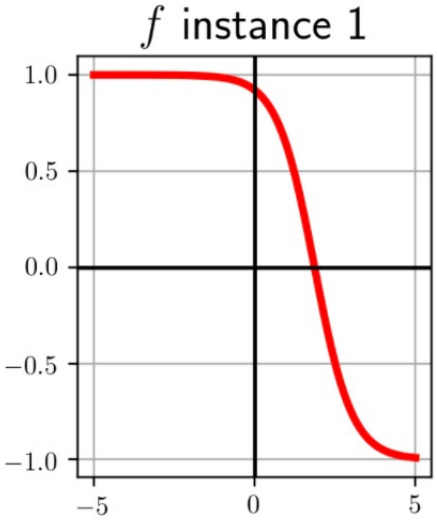

The neural networks family consists of *__parameterized__* functions, allowing them to take on a variety of different shapes (unlike the fixed-shape family).

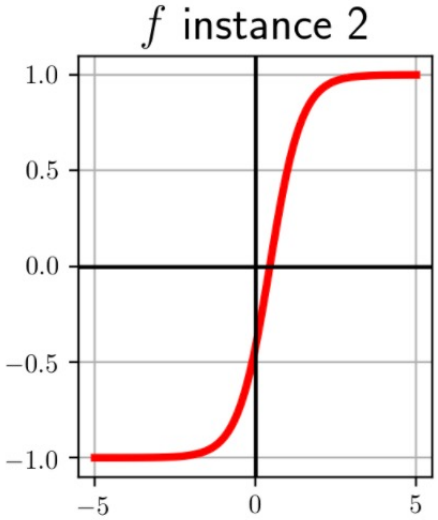
The simplest kind of neural networks universal approximator is a parameterized elementary function

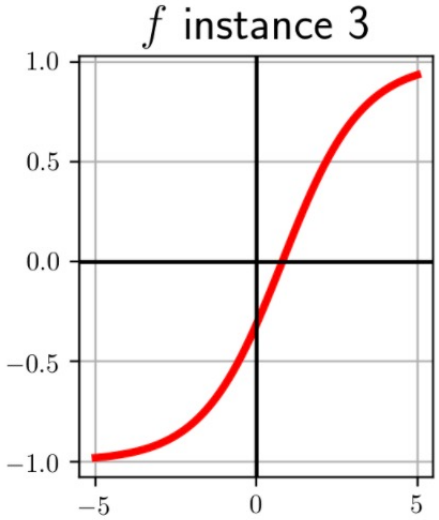

"Relu"

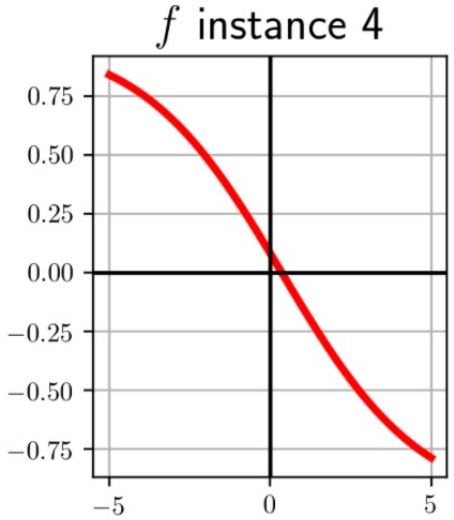
$$f_1(x) = \tanh\left(w_{1,0} + w_{1,1}x\right), \quad f_2(x) = \tanh\left(w_{2,0} + w_{2,1}x\right), \quad \text{etc.}$$

Notice the internal parameters $w_{b,0}$ and $w_{b,1}$ of a generic neural network function $f_b(x) = \tanh\left(w_{b,0} + w_{b,1}x\right)$ allow it to take on a variety of shapes, as illustrated below (where these parameters are set randomly).


$f$ instance 1
1.0
0.5
0.0
−0.5
−1.0
−5
0
5


$f$ instance 2
1.0
0.5
0.0
−0.5
−1.0
−5
0
5


$f$ instance 3
1.0
0.5
0.0
−0.5
−1.0
−5
0
5


$f$ instance 4
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−5
0
5

cell

To construct neural network features taking in higher dimensional input we take a linear combination of the input and pass the result through the nonlinear function.

$$f_b\left(\mathbf{x}\right) = \tanh\left(w_{b,0} + w_{b,1}x_1 + \cdots + w_{b,\,N}x_N\right)$$

As with the lower dimensional example, each function above can take on a variety of different shapes.

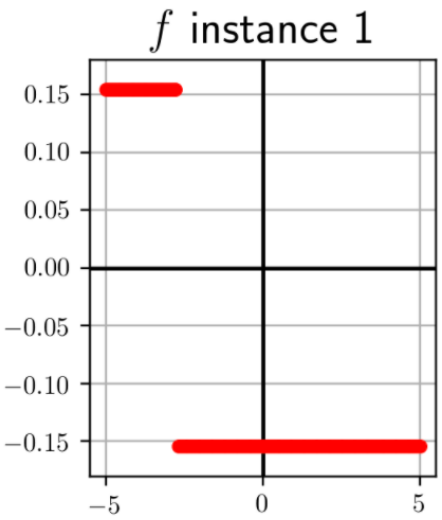
# Example: The trees family of universal approximators

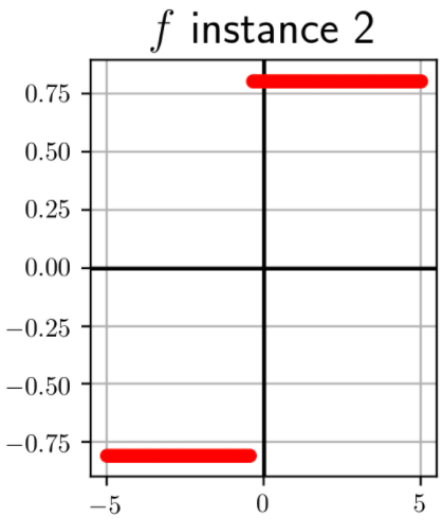
- The simplest sort of tree unit consists of discrete step functions or, as they are more commonly referred to, *stumps* whose break lies along a single dimension of the input space.
- A stump with one dimensional input $x$ can be written as follows where $s$ is called a *split point* at which the stump changes values, and $v_1$ and $v_2$ are values taken by the two sides of the stump respectively, which we refer to as *leaves* of the stump.

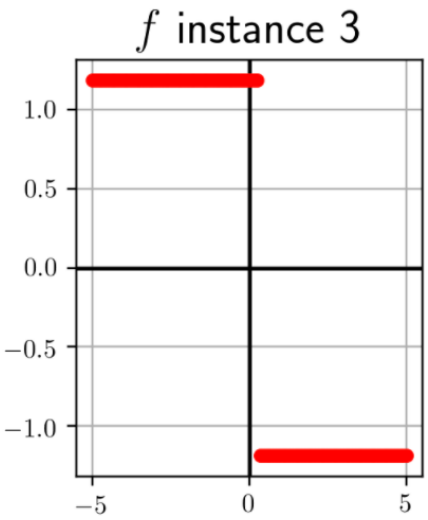
$$f(x) = \begin{cases} v_1 & x < s \\ v_2 & x > s \end{cases}$$

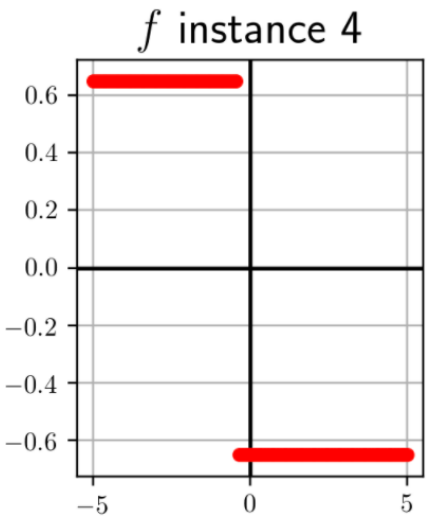
parameter

- A tree-based universal approximator is a set of such stumps (see figure below).

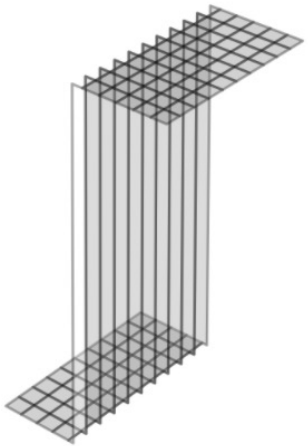

$f$ instance 1
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−5
0
5


$f$ instance 2
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−5
0
5

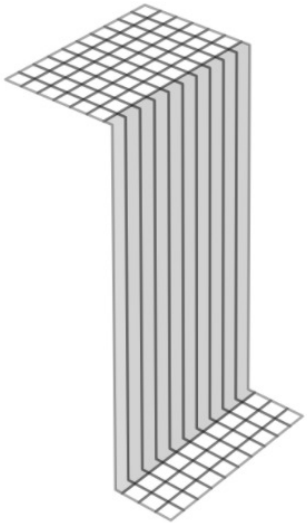

$f$ instance 3
1.0
0.5
0.0
−0.5
−1.0
−5
0
5


$f$ instance 4
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−5
0
5

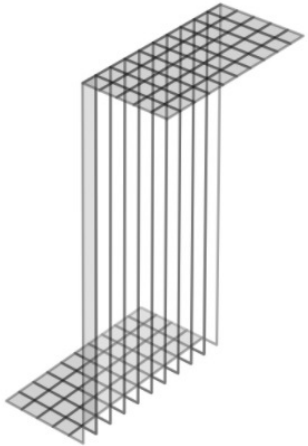
$f$ instance 1



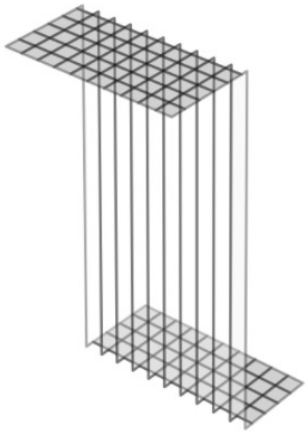
$f$ instance 2

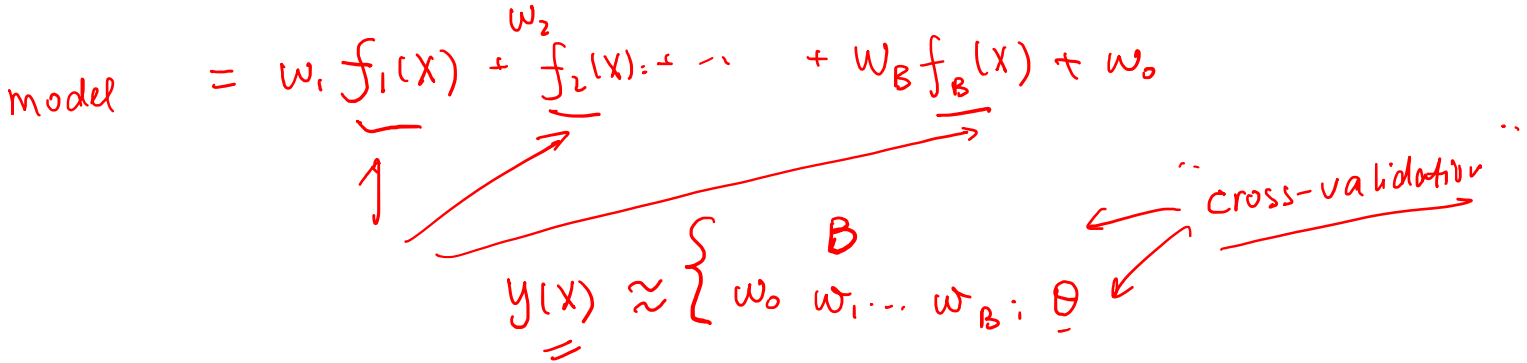


$f$ instance 3



$f$ instance 4






# The capacity and optimization dials

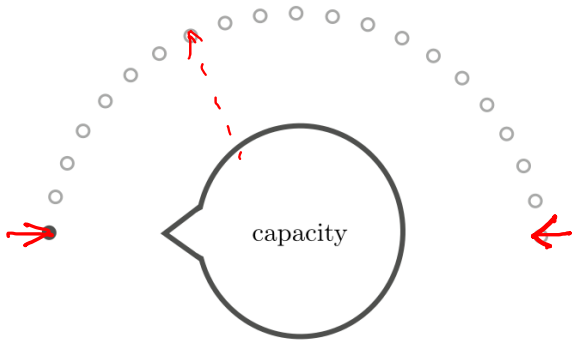
$$\text{model}\,(\mathbf{x}, \Theta) = w_0 + f_1\,(\mathbf{x})\,w_1 + f_2\,(\mathbf{x})\,w_2 + \cdots + f_B\,(\mathbf{x})\,w_B \approx y(x)$$

With any of the major universal approximators (i.e., fixed-shape, neural networks, or trees) we can attain universal approximation to any given precision, provided that the generic nonlinear model above:

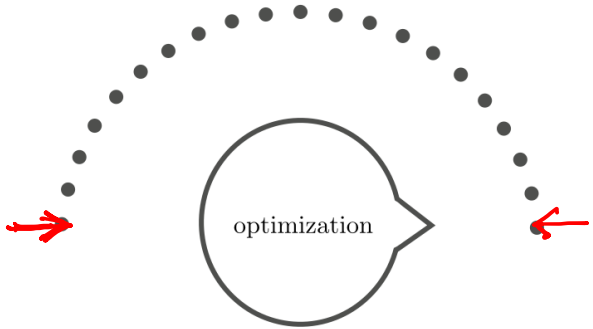
**1. has sufficiently large *capacity* (e.g., by making B large enough)**

**2. and that its parameters are tuned sufficiently well through *optimization* of an associated cost function**

- The *__capacity dial__* visually summarizes the amount of capacity we allow into a given model.
- When set all the way to the left we admit as little capacity as possible, i.e., we employ a *linear* model.
- As we move the capacity dial from left to right (clockwise) we adjust the model, adding more and more capacity.

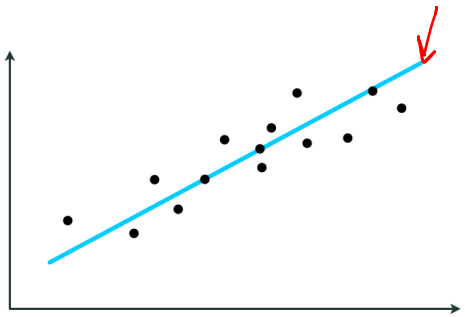


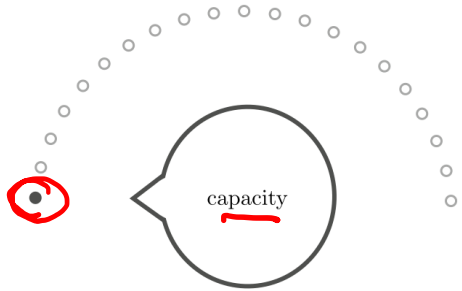
- The ***optimization dial*** visually summarizes how well we minimize the cost function of a given model whose capacity is already set.
- The setting all the way to the left denotes the initial point of whatever local optimization technique we use.
- As we turn the optimization dial from left to right, we move further and further along the particular optimization run, with the final step being represented visually as the dial set all the way to the right.

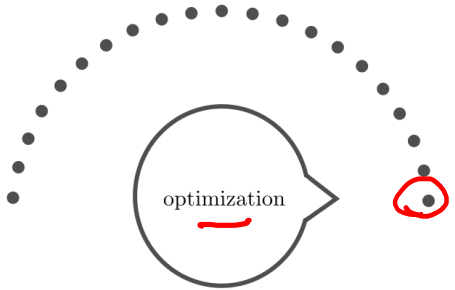


# Example: Linear regression and the two dials

With linear regression we set the capacity dial all the way to the *left* and the optimization dial all the way to the *right* in order to find the best possible set of parameters for a low capacity linear model (drawn in blue) that fits the given regression data.

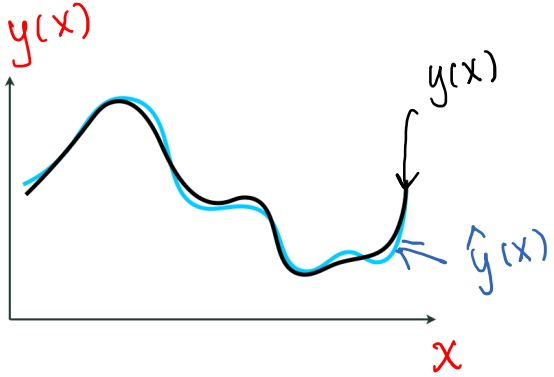


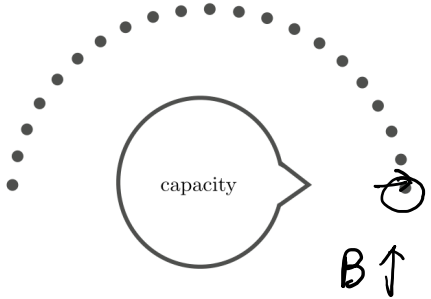


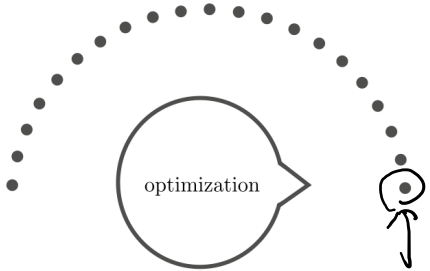



# Example: Universal function approximation and the two dials

With universal approximation of a continuous function (drawn in black) we set both dials to the *right*, admitting infinite capacity into the model and tuning its parameters by optimizing to completion.



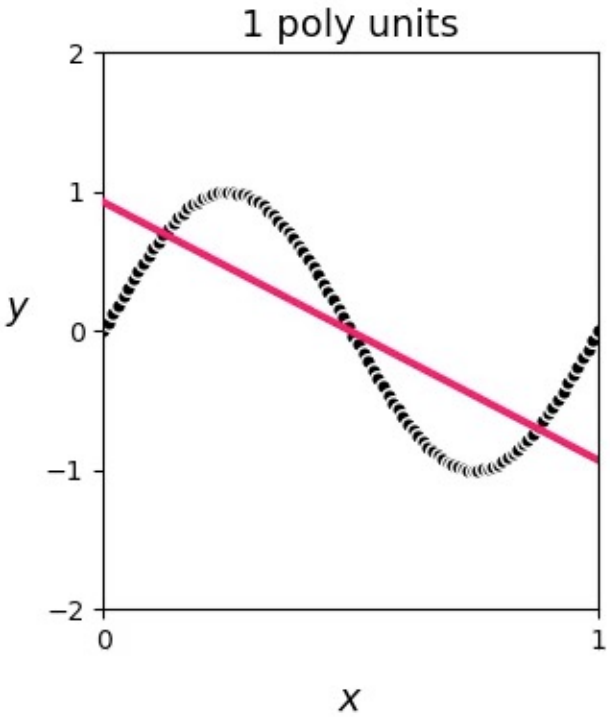



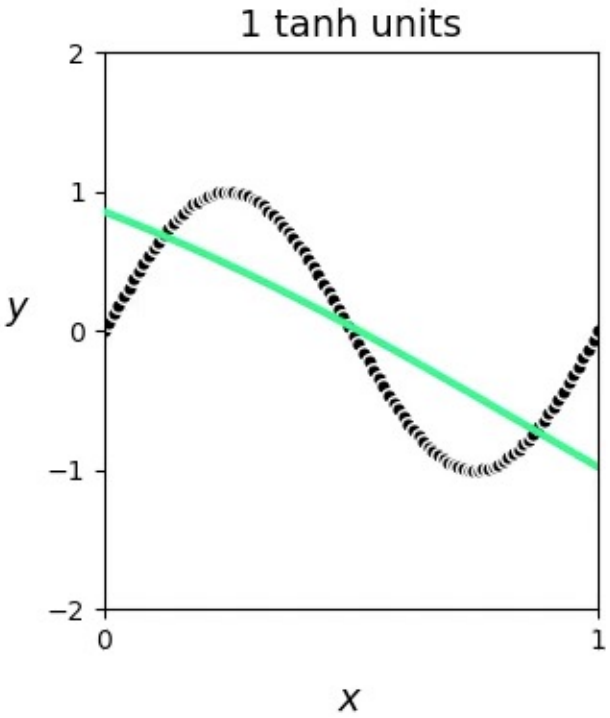



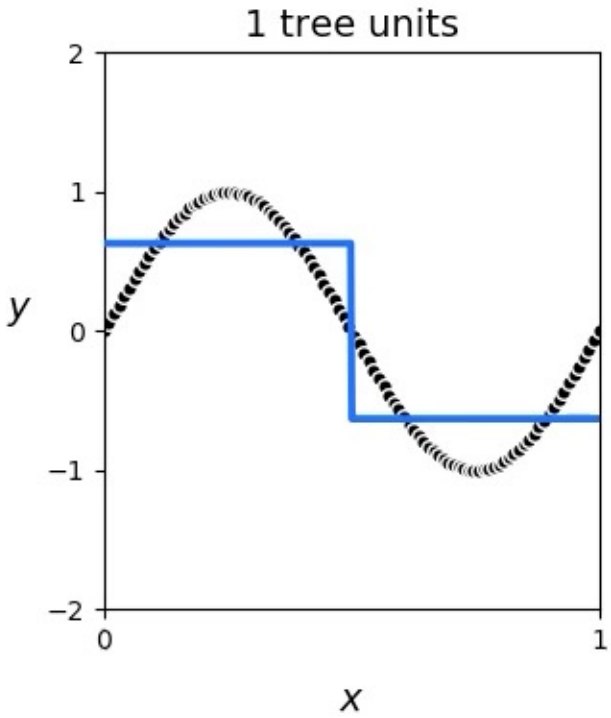
# Example: Universal approximation of near-perfect regression data

In the animation below we illustrate universal approximation of a near-perfect regression dataset consisting of (P = 10,000) evenly sampled points from an underlying sinusoidal function, using

- polynomial (left)
- neural network (middle)
- tree units (right)


1 poly units
2
1
0
−1
−2
y
0
1
x


1 tanh units
2
1
0
−1
−2
y
0
1
x


1 tree units
2
1
0
−1
−2
y
0
1
x

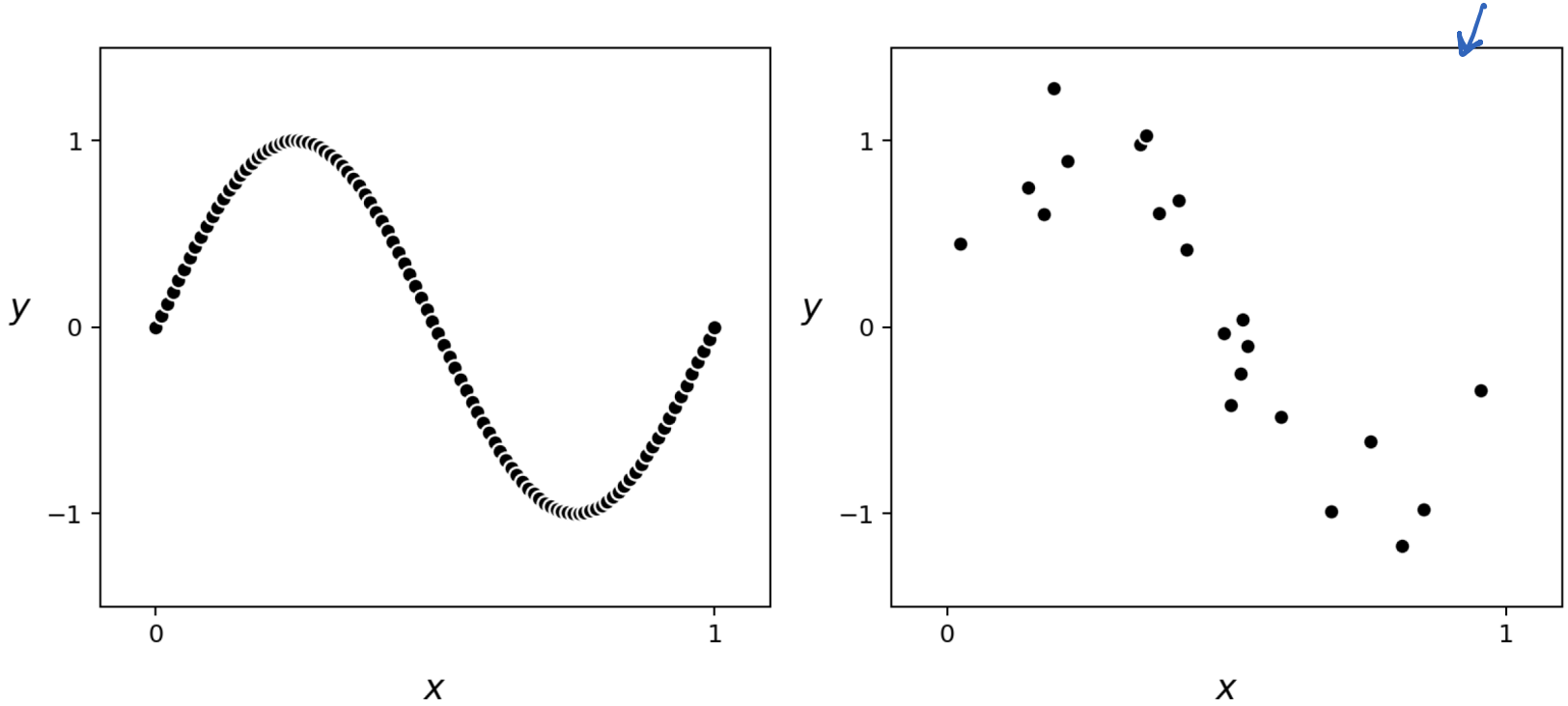
# 11.3 Universal Approximation of Real Data

① limited

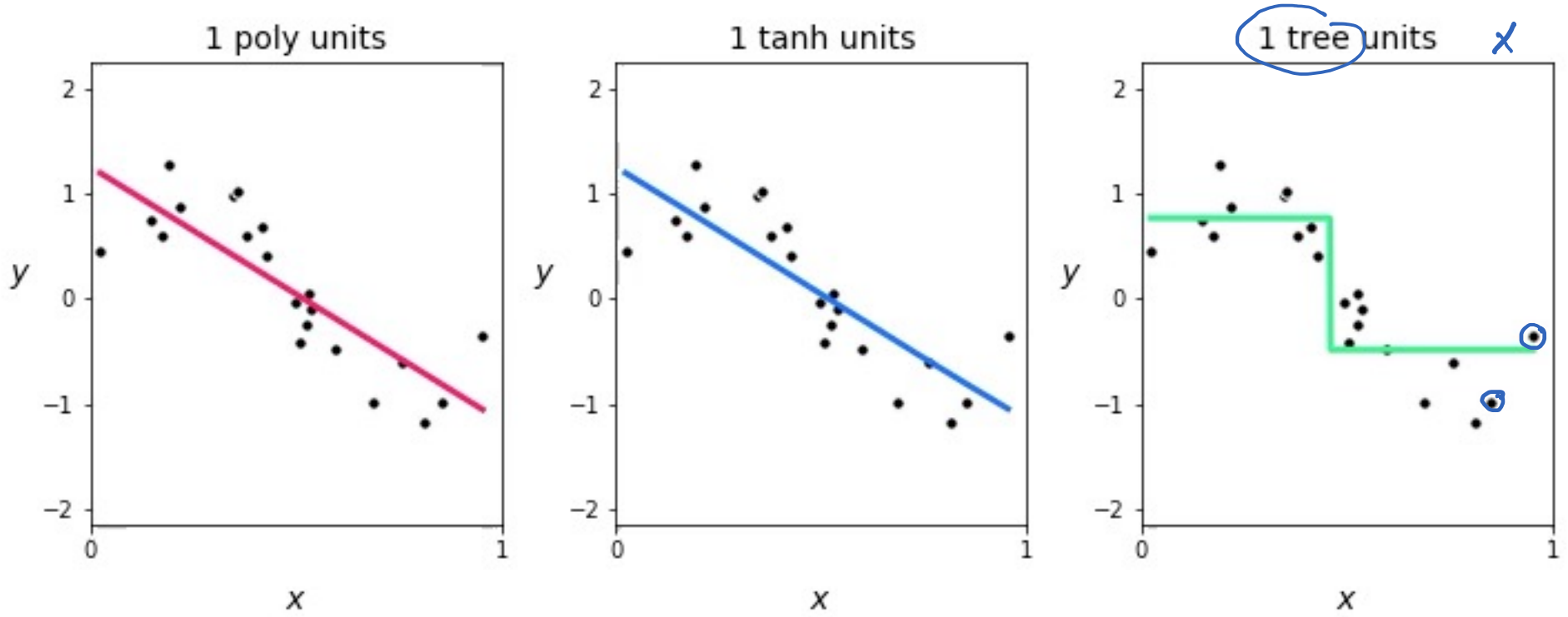
② noisy

(left panel) A ***near-perfect*** sinusoidal dataset.

(right panel) A ***real*** regression dataset formed by adding random noise to the output of a small subset of the dataset on the right.



In the following animation we illustrate the fully tuned nonlinear fit of a model employing 1 through 20 polynomial, neural network, and stump units.



Notice how:

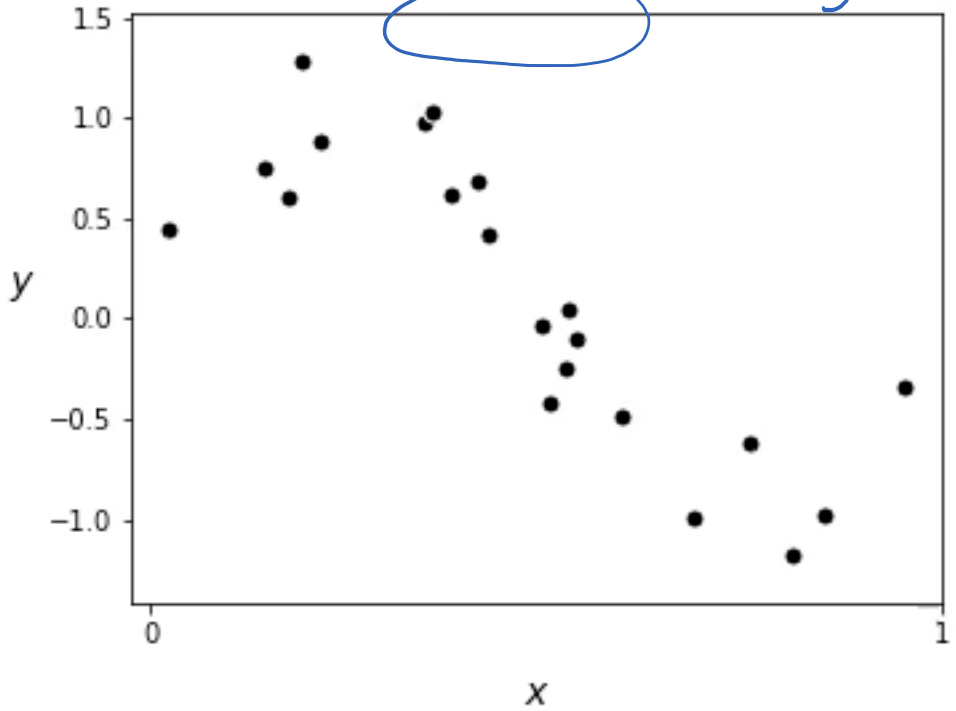
- with each of the universal approximators, all three models *underfit* the data when using only B = 1 unit in each case.
- each model improves as we increase B, but only up to a certain point after which each *tuned* model becomes far too complex and overfits the data.

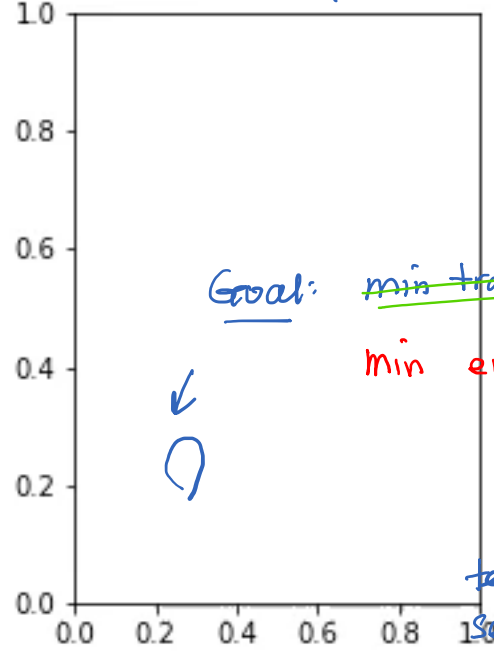
{X} "20 points" D-degree "D?

The following animation shows several polynomial-based models along with the corresponding Least Squares cost value each attains, i.e., the model's *training error*.

2
3

1 $wx+b$
2 $w_1x^2 + w_2x + b$




70%
10%
20%
train
vali
test

Goal: ~~min train error~~

min error ( unseen data points )

~~test~~ Set

validation Set

Notice how:

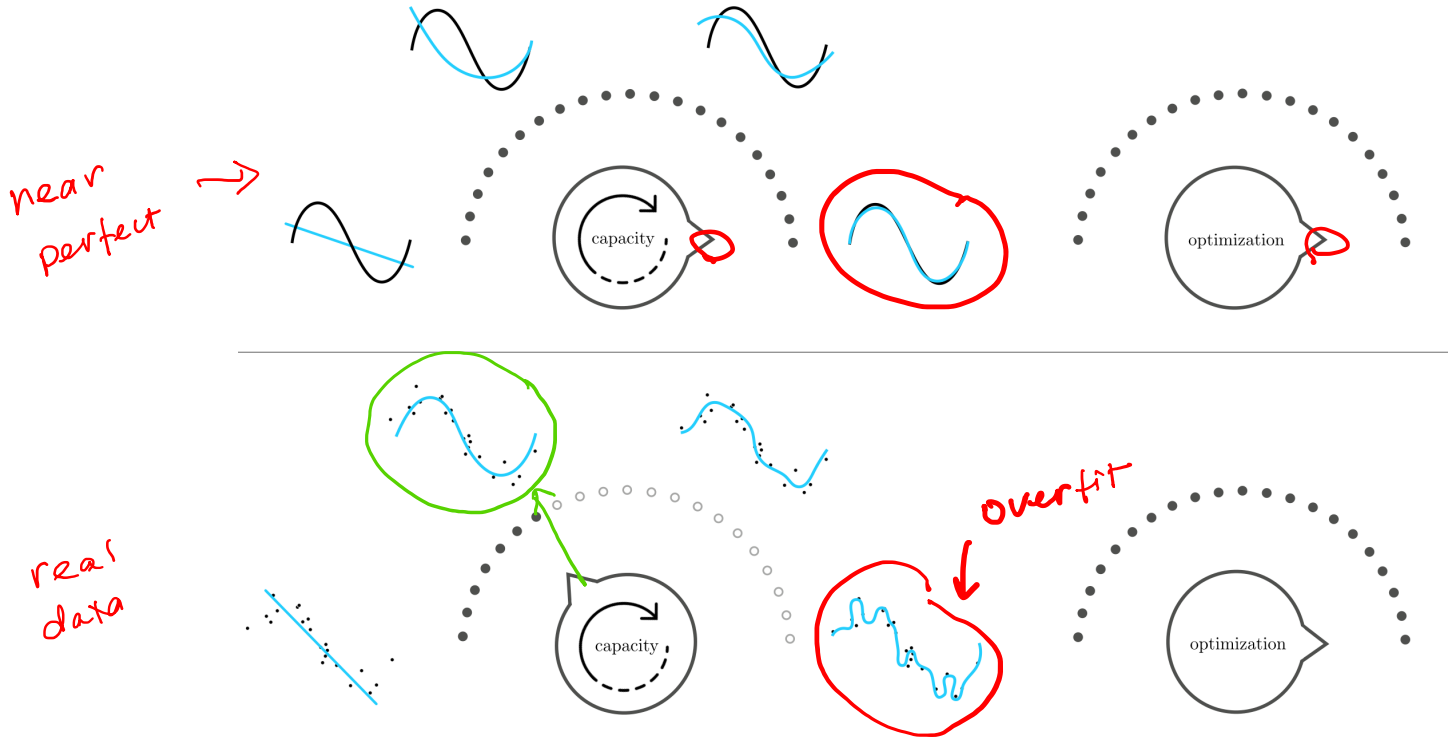
- in adding more polynomial units we turn up the capacity of our model and, optimizing each model to completion, the resulting tuned models achieve lower and lower training error.
- the resulting fit provided by each fully tuned model (after a certain point) becomes far too complex and starts to get *__worse__* in terms of how it represents the general regression phenomenon.
- as a measurement tool the training error only tells us how well a tuned model fits the *training data*, but fails to tell us when our tuned model becomes too complex.

# The capacity and optimization dials, revisited

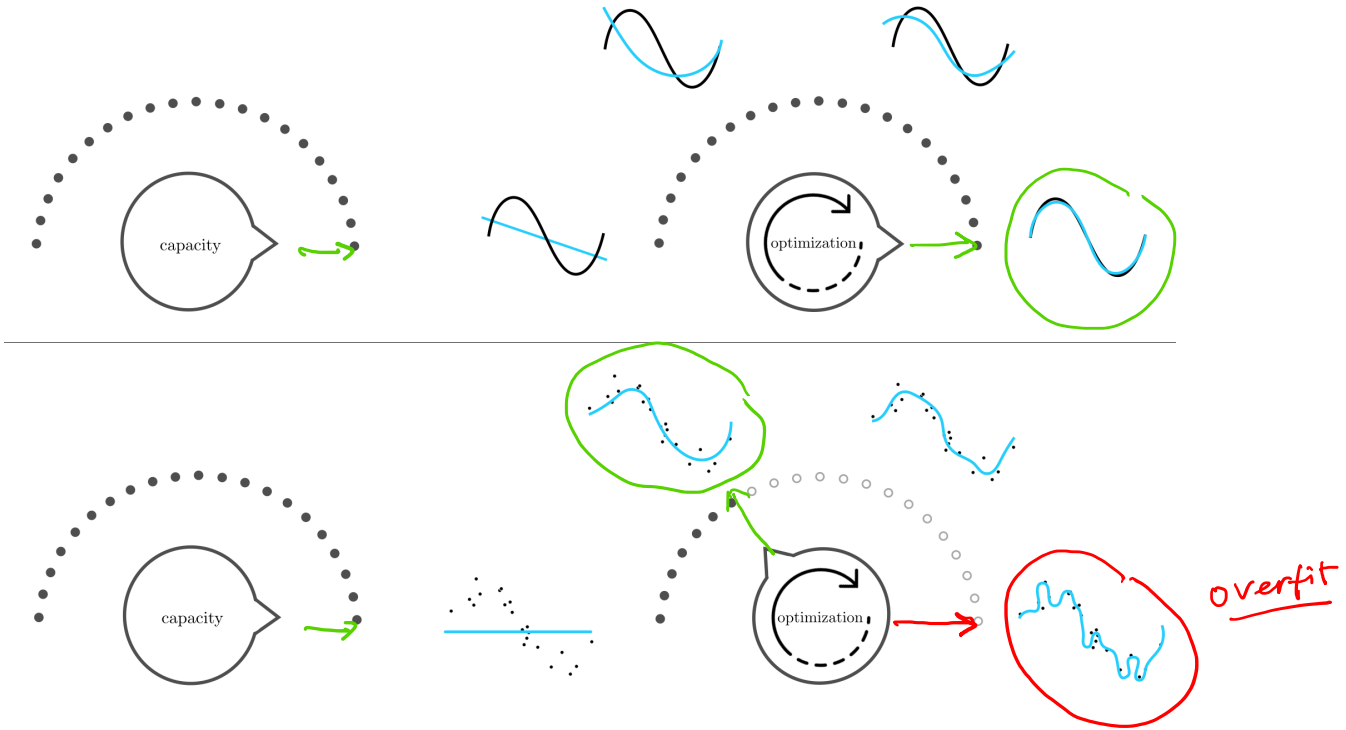
- When dealing with perfect data the best setting of capacity/optimization dials is when both dials are set all the way to the right.
- This means employing a model with maximum possible capacity and minimizing its corresponding cost to completion.
- The prototypical examples we just saw illustrate **how with real data we cannot simply set our capacity and optimization dials all the way to the right**.

noisy
limited

Setting the **optimization dial** all the way to the right, with real data (unlike perfect data) we cannot simply set our capacity dial all the way to the right as well!



Setting the **capacity dial** all the way to the right, with real data (unlike perfect data), we cannot simply set our optimization dial all the way to the right as well!

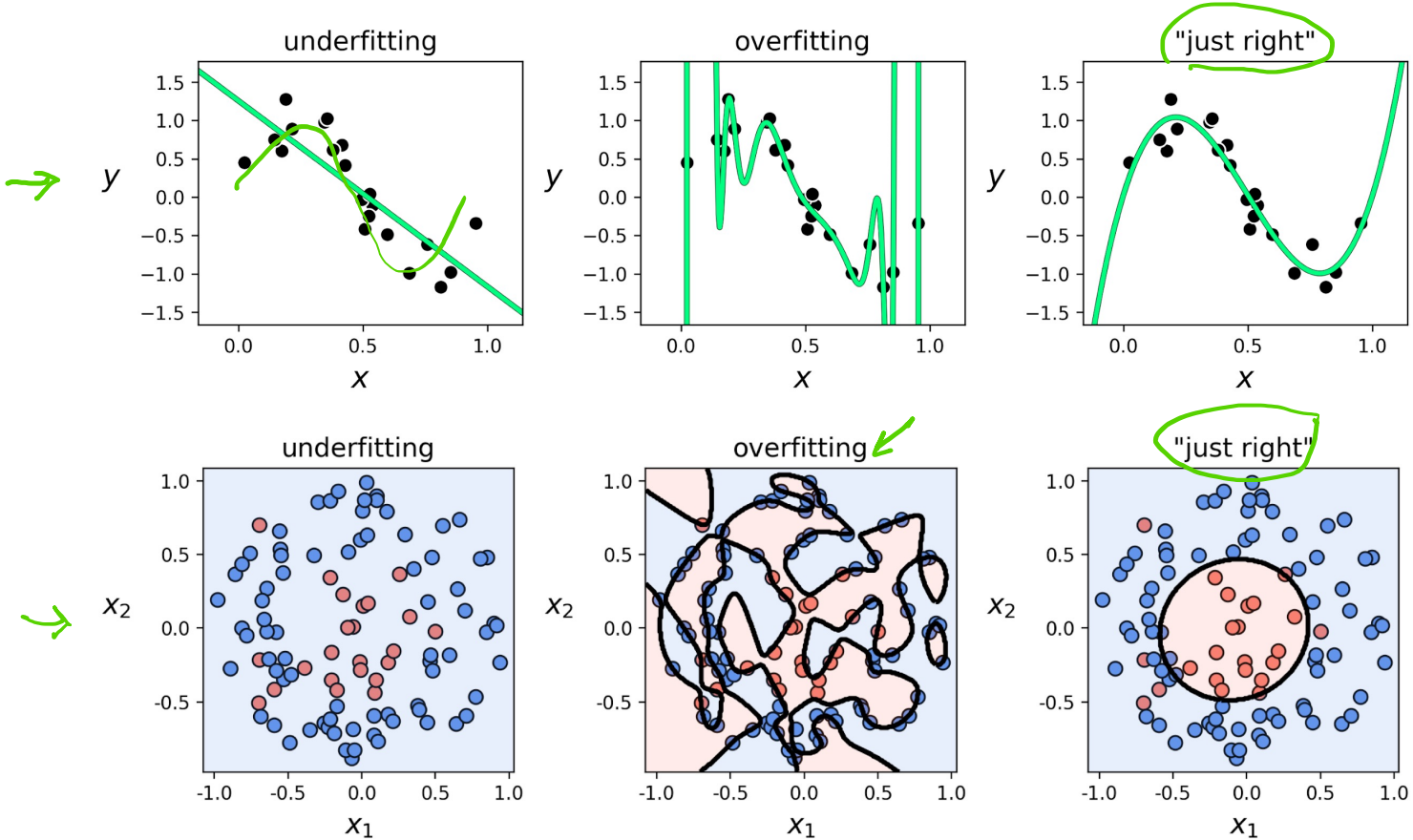


# Motivating a new measurement tool

optimization or capacity

How we set our capacity and optimization dials in order to achieve a final tuned model that has ***just the right amount of complexity*** for a given dataset is the main challenge we face when employing universal approximator-based models with real data.

What do both the underfitting and overfitting patterns have in common, that the "just right" model does not?



## Observation

While the underfitting and overfitting models differ in how well they represent data *we already have*, **they will both fail to adequately represent *new data* generated via the same process by which the current data was made**.

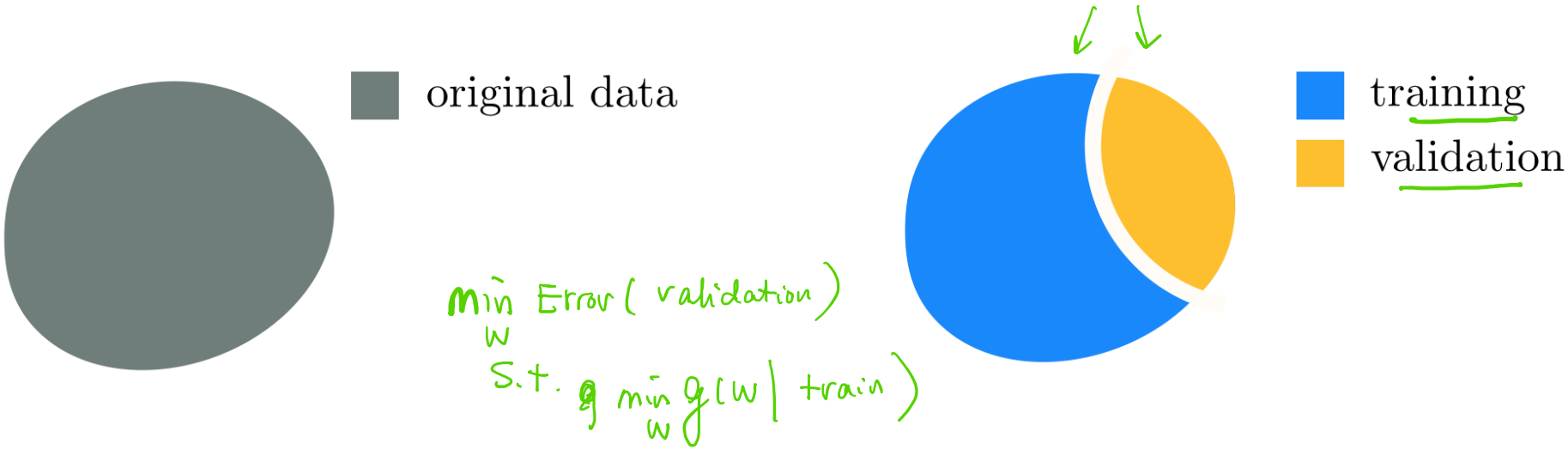
## Question

But how do we quantify something we will receive in the future?

# The validation error

To simulate the data we receive in the future, we simply remove a random portion of our data and treat it as "new data we might receive in the future." We train our selection of models on only the portion of data that remains, and *validate* the performance of each model on this randomly removed chunk of "new" data.

- The random portion of the data we remove to validate our model(s) is commonly called the *validation data*
- The remaining portion we use to train models is likewise referred to as the *training data*



- There is no precise rule for what portion of a given dataset should be saved for validation.
- In practice, typically between $\frac{1}{10}$ to $\frac{1}{3}$ of the data is assigned to the validation set.
- Generally speaking, the larger and/or more representative (of the true phenomenon from which the data is sampled) a dataset is the larger the portion of the original data may be assigned to the validation set (e.g., $\frac{1}{3}$ ).

# 11.4 Naive Cross-Validation

- By carefully searching through a set of models ranging in complexity we can identify the best of the bunch, the one that provides minimal error on the validation set.
- This comparison of models is called *cross-validation* (or sometimes *model search* or *selection*).
- Cross-validation is the basis of feature learning as it provides a systematic way to *learn* (as opposed to *engineer*) the proper form a nonlinear model should take for a given dataset.

Suppose we want to select one of the M models below that has the ideal amount of complexity for a given dataset:

$$\begin{aligned}
&\text{model}_1\left(\mathbf{x}, \Theta_1\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 \\
&\text{model}_2\left(\mathbf{x}, \Theta_2\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 \\
&\qquad\qquad\qquad\qquad\vdots \\
&\text{model}_M\left(\mathbf{x}, \Theta_M\right) = w_0 + f_1\left(\mathbf{x}\right) w_1 + f_2\left(\mathbf{x}\right) w_2 + \cdots + f_M\left(\mathbf{x}\right) w_M .
\end{aligned}$$

Q: Can we do better

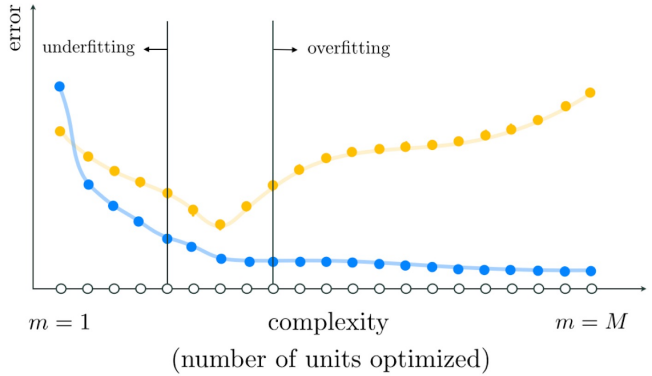
Naive cross-validation entails taking the following steps:

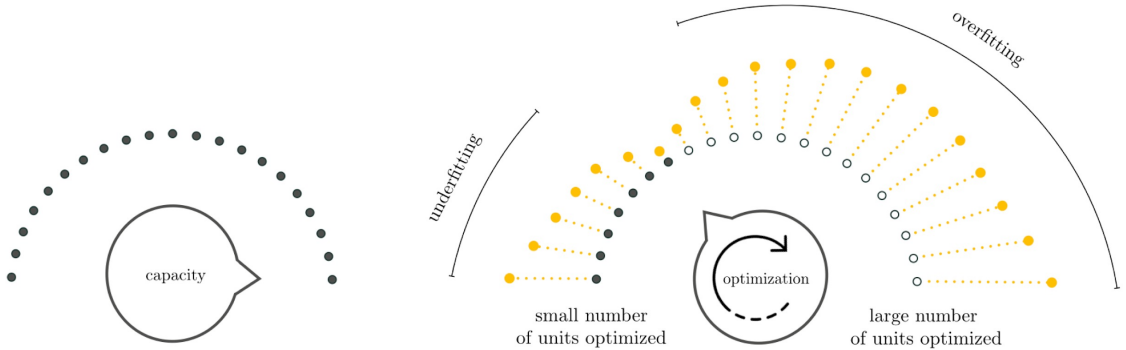
- split original data randomly into training and validation portions
- optimize every model to completion using training split
- measure the error of all M fully trained models on each portion of the data
- pick the one that achieves *minimum validation*

training split & valid split

# 11.5 Efficient Cross-Validation via Boosting

The basic principle behind boosting based cross-validation is to progressively build a high capacity model one unit at-a-time, using units from a single type of universal approximator.




(top panel)Prototypical training and validation error curves associated with a completed run of boosting.(bottom panels) We fix the capacity dial all the way to the right and the optimization dial all the way to the left, slowly turning it from left to right, with each notch denoting the complete optimization of one additional unit of the model.

# Technical details

- For boosting we need a set of M nonlinear features or units from a single family of universal approximators

$$\mathcal{F} = \{f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_M(\mathbf{x})\}.$$

poly unit / unit NN / tree

- We add these units sequentially or one-at-a-time building a set of M tuned models $\left[\text{model}_m\right]_{m=1}^{M}$ that increase in complexity with respect to the training data, from m=1 to m=M, ending with a generic nonlinear model composed of M units.

We will express this final boosting-made model as

$$\text{model}\left(\mathbf{x}, \Theta\right) = w_0 + f_{s_1}\left(\mathbf{x}\right) w_1 + f_{s_2}\left(\mathbf{x}\right) w_2 + \cdots + f_{s_M}\left(\mathbf{x}\right) w_M .$$

Here we have re-indexed the individual units to $f_{s_m}$ (and the corresponding weight $w_m$ ) to denote the unit from the entire collection in $\mathcal{F}$ added at the $m^{th}$ round of the boosting process.

The linear combination weights $w_0$ through $w_M$ along with any additional weights internal to $f_{s_1}, f_{s_2}, \ldots, f_{s_M}$ are represented collectively in the weight set $\Theta$ .

- The process of boosting is performed in a total of M rounds.
- At each round we determine which unit, when added to the running model, best lowers its training error.
- We then measure the corresponding validation error provided by this update, and in the end after all rounds of boosting are complete, use the lowest validation error measurement found to decide which round provided the best overall model.

- For the sake of simplicity, we only discuss nonlinear regression on the training dataset $\{(\mathbf{x}_p, y_p)\}_{p=1}^{P}$ employing the Least Squares cost.
- However, the principles of boosting we will see remain *exactly* the same for other learning tasks (e.g., two-class and multi-class classification) and their associated costs.

# Round 0 of boosting

- Round 0 starts with the model

$$\text{model}_0\left(\mathbf{x}, \Theta_0\right) = w_0$$

whose weight set $\Theta_0 = \{w_0\}$ contains a

- single bias weight, whose optimal value $w_0^\star$ is found by by minimizing the Least Squares cost

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0\left(\mathbf{x}_p, \Theta_0\right) - y_p\right)^2 = \frac{1}{P}\sum_{p=1}^{P}\left(w_0 - y_p\right)^2$$

- We fix the bias weight at the value of $w_0^\star$ forever more throughout the process.

# Round 1 of boosting

- Having tuned the only parameter of $\text{model}_0$ we now *boost* its complexity by adding the weighted unit $f_{s_1}(\mathbf{x})\, w_1$ to it:

$$\text{model}_1(\mathbf{x}, \Theta_1) = \text{model}_0(\mathbf{x}, \Theta_0) + f_{s_1}(\mathbf{x})\, w_1.$$

- To determine which unit in our set $\mathcal{F}$ best lowers the training error, we press $\text{model}_1$ against the data by minimizing the following cost for every unit $f_{s_1} \in \mathcal{F}$

$$\begin{aligned} &\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_0(\mathbf{x}_p, \Theta_0) + f_{s_1}(\mathbf{x}_p)\, w_1 - y_p\right)^2 \\ &= \frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + f_{s_1}(\mathbf{x}_p)\, w_1 - y_p\right)^2 \end{aligned}$$

Round $m > 1$ of boosting

We begin with $\text{model}_{m-1}$ consisting of a bias term and $m-1$ units of the form

$$\text{model}_{m-1}\left(\mathbf{x}, \Theta_{m-1}\right) = w_0^\star + f_{s_1}^\star\left(\mathbf{x}\right) w_1^\star + \cdots + f_{s_{m-1}}^\star\left(\mathbf{x}\right) w_{m-1}^\star.$$

Basic idea: build a set of decision trees using boosting

"XGBoost" "LBoost" ... "tabular"

- We then seek out the best weighted unit $f_{s_m}(\mathbf{x})\, w_m$ to add to our running model

$$\text{model}_m(\mathbf{x}, \Theta_m) = \text{model}_{m-1}(\mathbf{x}, \Theta_{m-1}) + f_{s_m}(\mathbf{x})\, w_m$$

$f_2(X) \sim X_2$ . $f_1(X) \sim X_1$ "learnable"

previously found — fixed

① tree: $f_i(\,) = \begin{cases} v_1, & x_i > \delta \\ v_2, & x_i < \delta \end{cases}$

- by minimizing the following cost

$$\frac{1}{P}\sum_{p=1}^{P}\left(\text{model}_{m-1}(\mathbf{x}_p, \Theta_{m-1}) + f_{s_m}(\mathbf{x}_p)\, w_m - y_p\right)^2 =$$

$$\frac{1}{P}\sum_{p=1}^{P}\left(w_0^\star + w_1^\star f_{s_1}^\star + \cdots + f_{s_{m-1}}^\star(\mathbf{x}_p)\, w_{m-1}^\star + f_{s_m}(\mathbf{x}_p)\, w_m - y_p\right)^2$$

$f_i(\,) = \begin{cases} v_1, & x_1 + x_2 > \delta \\ v_2, & w_1 x_1 + w_2 x_2 < \delta \end{cases}$

$(x_1\ x_2 \cdots x_N)$

② NN: $\tanh(\vec{x}\vec{w})$ $x \in \mathbb{R}^N$

③ "poly"

# 11.6 Efficient Cross-Validation via Regularization

Overfitting occurs when:

- **the *capacity* of a machine learning model is too high** ✓
- ***and*, its corresponding cost function (over the training data) is *optimized* too well.**


Regularization - a potential solution:

- we set the model parameters purposefully away from the global minimum of its associated cost function, so as to find where the validation error (not training error) is at its lowest.

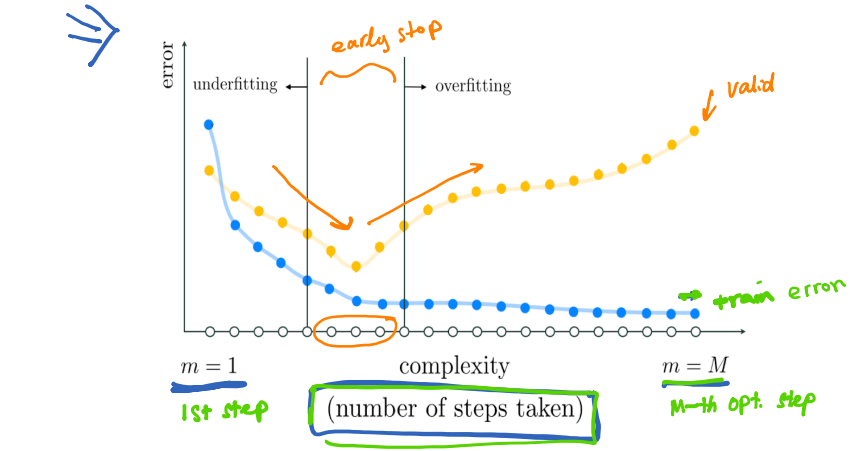
Two most popular approaches to regularization:

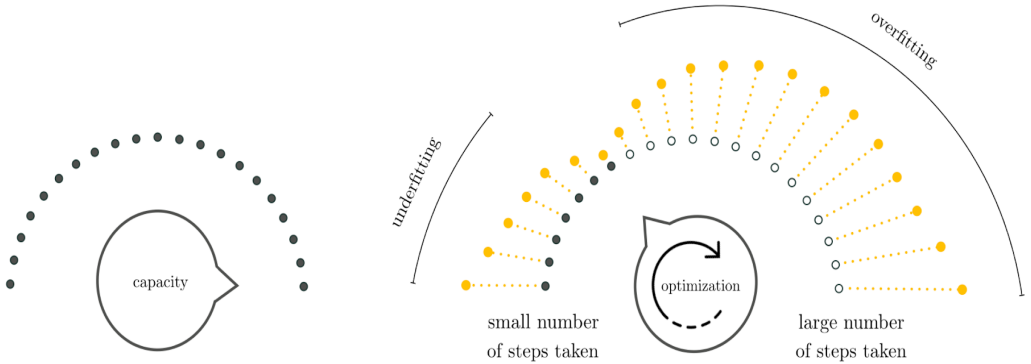
- Early stopping
- Adding a regularizer to the cost function

# Early stopping based regularization

**(top panel)** A prototypical pair of training/validation error curves associated with a generic run of the early stopping regularization.

**(bottom panels)** We set our capacity dial all the way to the right and our optimization dial all the way to the left, slowly moving it from left to right in search of a model with minimum validation error. Here each notch on the optimization dial abstractly denotes a step of local optimization.



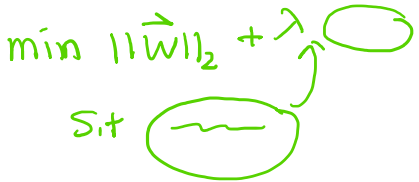
When is validation error really at its lowest?

- While generally speaking validation error decreases at the start of an optimization run and eventually increases (making somewhat of a 'U' shape) it can certainly fluctuate up and down during optimization.
- Often in practice a reasonable engineering choice is made as to when to stop based on how long it has been since the validation error has *not* decreased.
- Moreover, one need not truly halt a local optimization procedure to employ the thrust of early stopping, and can simply run the optimizer to completion and select the best set of weights from the run (that minimize validation error) after completion.

# Regularizer based methods

A regularizer is a simple function that can be added to a machine learning cost for a variety of purposes:

- as a natural part of relaxing the support vector machine and multi-class learning scenarios
- for feature selection $\|\vec{\ell}\|_1$
- and, for regularization (which we discuss here) $\|\vec{w}\|_1$, $\|\vec{w}\|_2$, $\|\vec{w}\|_p$



Denoting the original cost function by g, and the associated regularizer by h, the regularized cost is given as the linear combination of g and h as

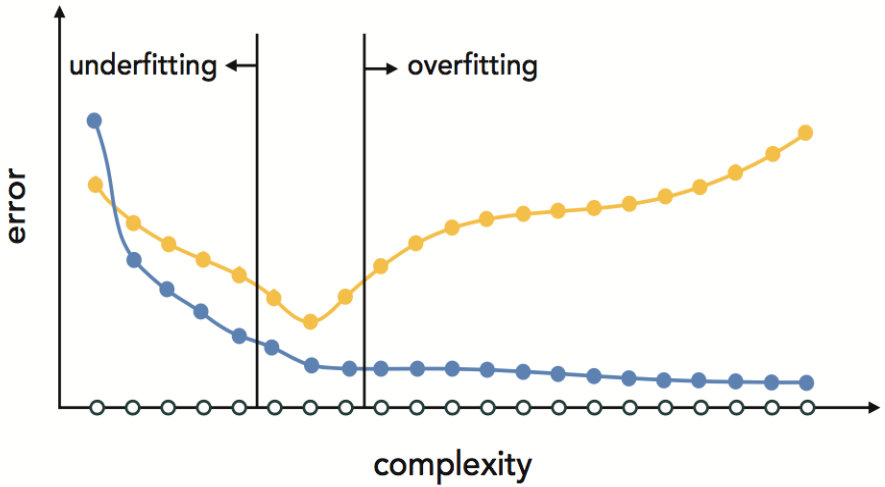
$$g(\Theta) + \lambda h(\Theta)$$

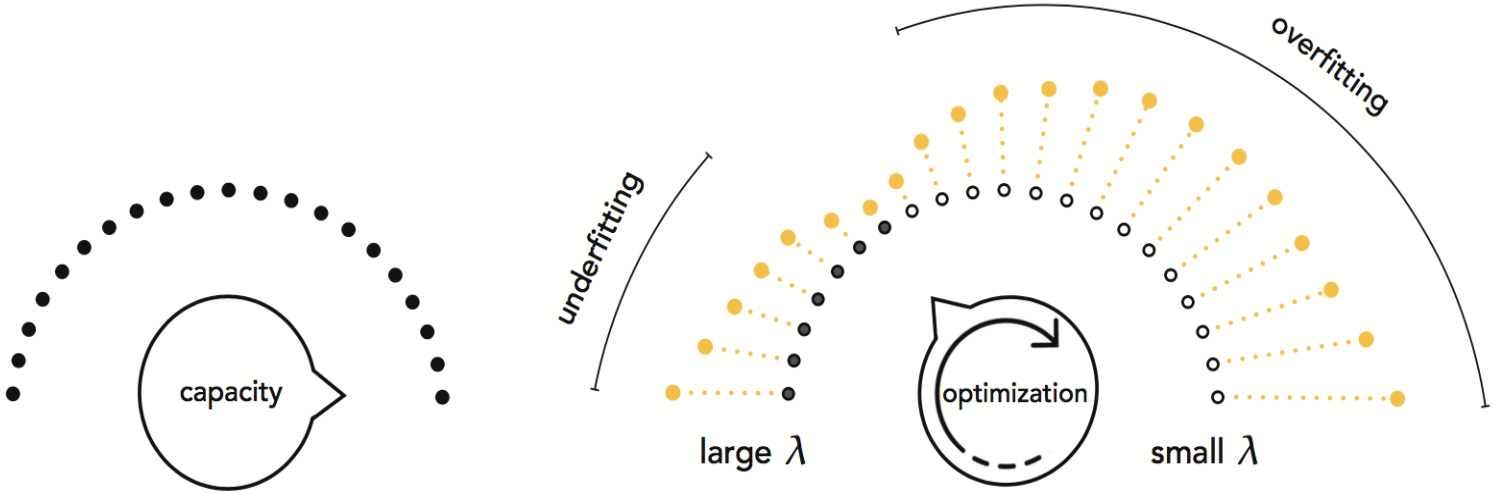
reg.

Where $\lambda$ is referred to as the *regularization parameter*.

The regularization parameter

- is always non-negative $\lambda \geq 0$
- controls the mixture of the cost and regularizer
- when set very small, the regularized cost is essentially just g
- when set very large the regularizer h dominates in the linear combination and drowns out g


underfitting
overfitting
error
complexity


capacity
underfitting
overfitting
optimization
large λ
small λ

Implementation notes

- Bias weights are often not included in the regularizer.
- How many values we can try to tune the regularization parameter is often limited by computation and time restrictions, since for every value of $\lambda$ tried a complete minimization of a corresponding regularized cost function must be performed.
- While the squared $\ell_2$ norm is a very popular regularizer, one can use - in principle - any simple function as a regularizer. A few examples follow

Popular regularizers in machine learning

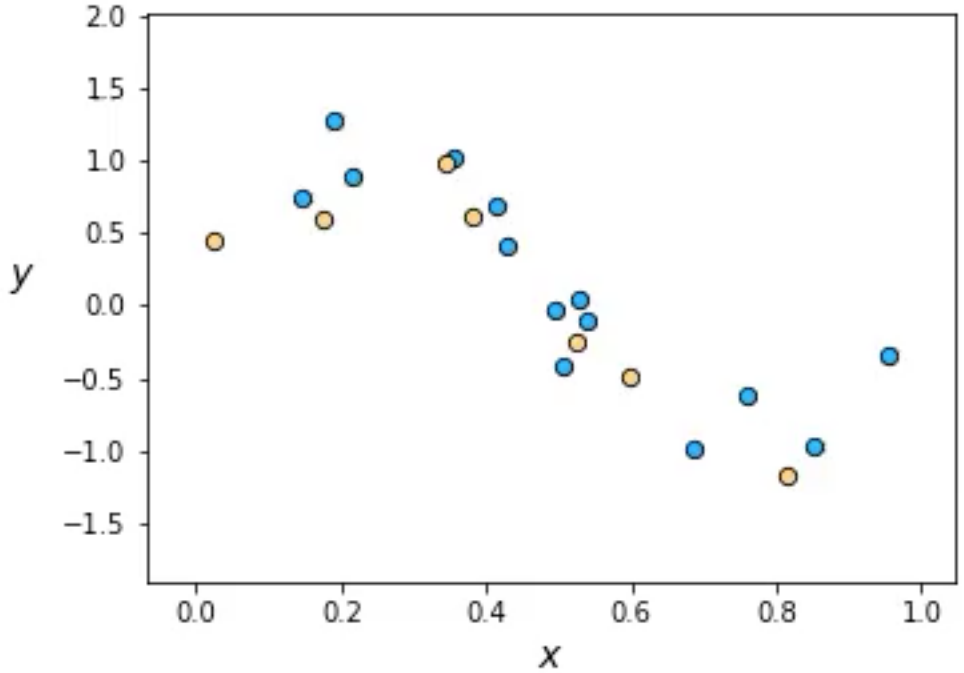
**The squared $\ell_2$ norm:** defined as $\|\mathbf{w}\|_2^2 = \sum_{n=1}^{N} w_n^2$ , this regularizer incentivizes weights to be *small* (in Euclidean sense).

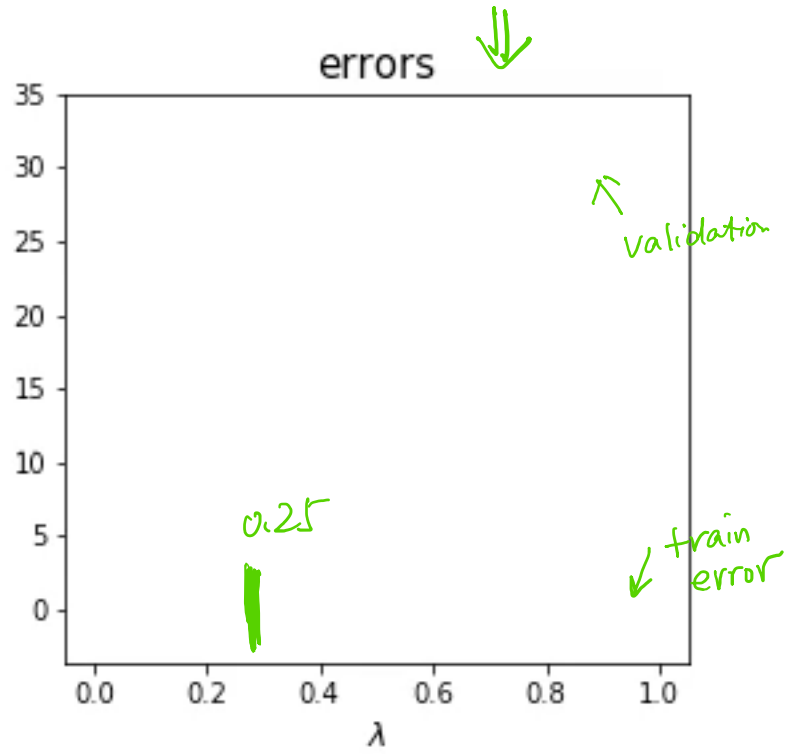
**The $\ell_1$ norm:** defined as $\|\mathbf{w}\|_1 = \sum_{n=1}^{N} |w_n|$, this regularizer tends to produce sparse weights.

**The total variation norm:** defined as $\|\mathbf{w}\|_{\mathrm{TV}} = \sum_{n=1}^{N-1} |w_{n+1} - w_n|$, this regularizer tends to produce *smoothly varying* weights.

# Example: Tuning a regularization parameter for regression

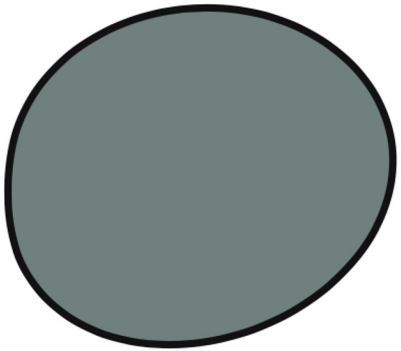
- In this example we use an $\ell_2$ -regularized degree-10 polynomial model.
- The training set/error is shown in light blue while the validation set/error is shown in yellow.
- We try out 100 values of $\lambda$ (i.e., the regularization parameter) between 0 and 1.

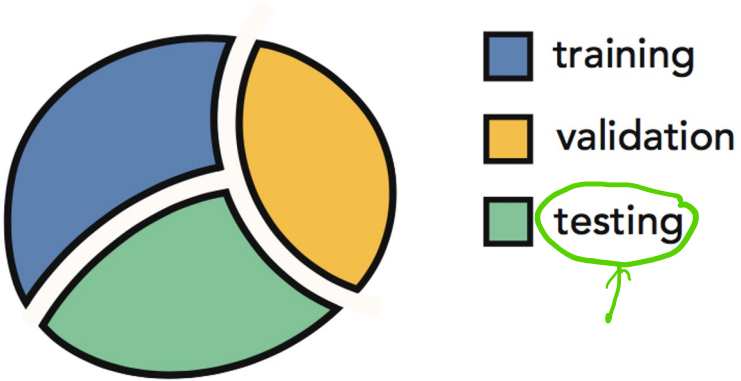

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
0.0
0.2
0.4
0.6
0.8
1.0
x


errors
35
30
25
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
λ
validation
train error
0.25

# 11.7 Testing data

- As with training data, **it is also possible to overfit to validation data as well during the cross-validation procedure**.
- Provided the dataset is large enough, we can ameliorate this risk by splitting our original data into not two sets (training and validation), but three: training, validation, and testing sets.





- After the cross-validated model is constructed its quality is measured on the testing data, on which it has neither been trained nor validated on.
- This testing error gives an unbiased estimate of the cross-validated model's performance, and is generally closer to capturing the true error of our model on 'future data generated by the same phenomenon'.

What portion of the original dataset should we assign to our testing set?

- As with the portioning of training and validation, there is no general rule for portioning.
- Note: the use of testing data is a luxury we can indulge in only when we have a large amount of data. When data is scarce we must leverage it all just to build a halfway reasonable cross-validated model.
- When data is plentiful, often the size of validation and testing sets are chosen similarly, e.g., if $\frac{1}{10}$ of a dataset is used for validation, often for simplicity the same portion is used for testing as well.

# 11.8 Which Universal Approximator Works Best in Practice?

- It is virtually never clear a priori which, if any, of the universal approximators will work best.
- Indeed cross-validation is the toolset one uses in practice to decide which type of universal approximator based model works best for a particular problem.
- However, broad understanding of a dataset can, in some instances, direct the choice of universal approximator.

## Naturally discontinuous data

tabular

Because oftentimes business, census, and (more generally) structured datasets consist of broad mixtures of continuous and discontinuous categorical input features, **tree-based universal approximators** with their discontinuous step-like shapes, often provide stronger results on average than other universal approximator types.

Naturally continuous data

Data that is naturally continuous (e.g., data generated by natural processes or sensor data) is often better matched with a continuous universal approximator: **fixed-shape or neural network**.

## Interpolation vs. extrapolation

When future predictions need be made outside the input domain of the original dataset, **fixed-shape or neural network approximators** can be preferred over trees – the latter by their very nature always creating perfectly flat predictions outside of the original data's input domain.

Human interpretability

Due to their discrete branching structure, **tree-based universal approximators** can often be much easier to interpret than other approximators (particularly neural networks).

# 11.9 Bagging Cross-Validated Models

- The random nature of splitting data into training and validation poses a potential flaw to our cross-validation process: bad training-validation splits.
- Such bad splits are not desirable representatives of the underlying phenomenon that generated them, which can result in poorly representative cross-validated models.

A practical solution to this fundamental problem is to simply perform several different training-validation splits, determine an appropriate cross-validated model on each split, and then ***average* the resulting cross-validated models**. This is called bagging.

By averaging a set of cross-validated models we can *very often* both 'average out' the potentially undesirable characteristics of each model while synergizing their positive attributes.

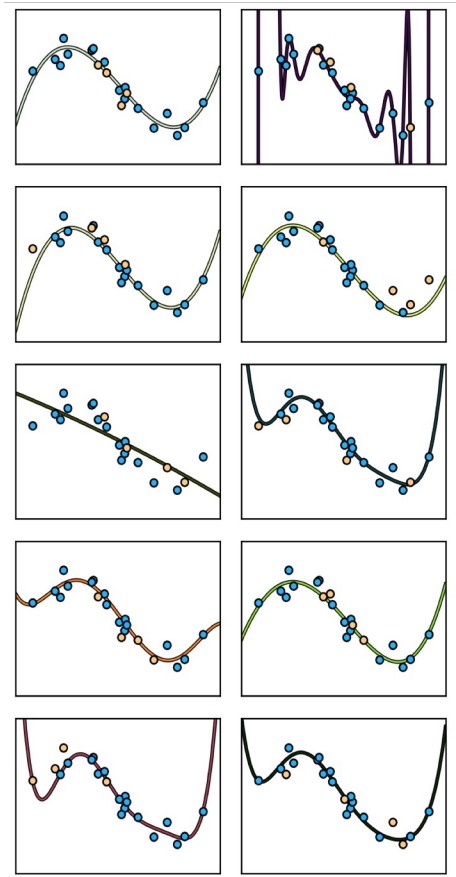
# Bagging regression models

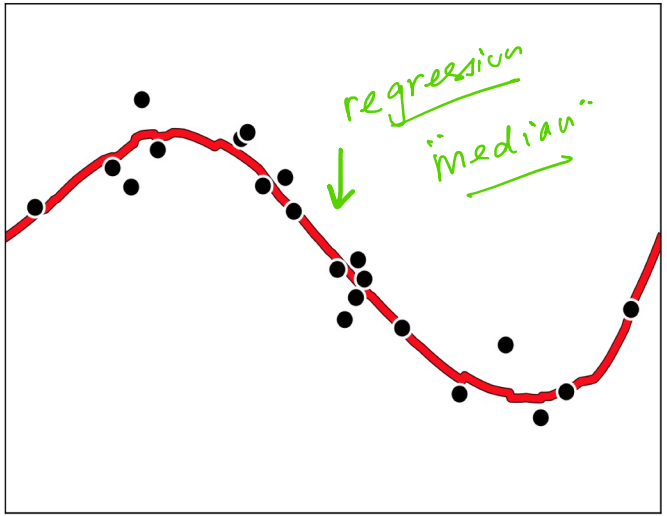
Generally the best way to bag (or average) cross-validated regression models is by taking their median (as opposed to their mean).

# Example:  Bagging cross-validated regression models

(small panels) Ten different random training-validation splits of a nonlinear regression dataset (blue: training, yellow: validation), with the best cross-validated model drawn in each panel.

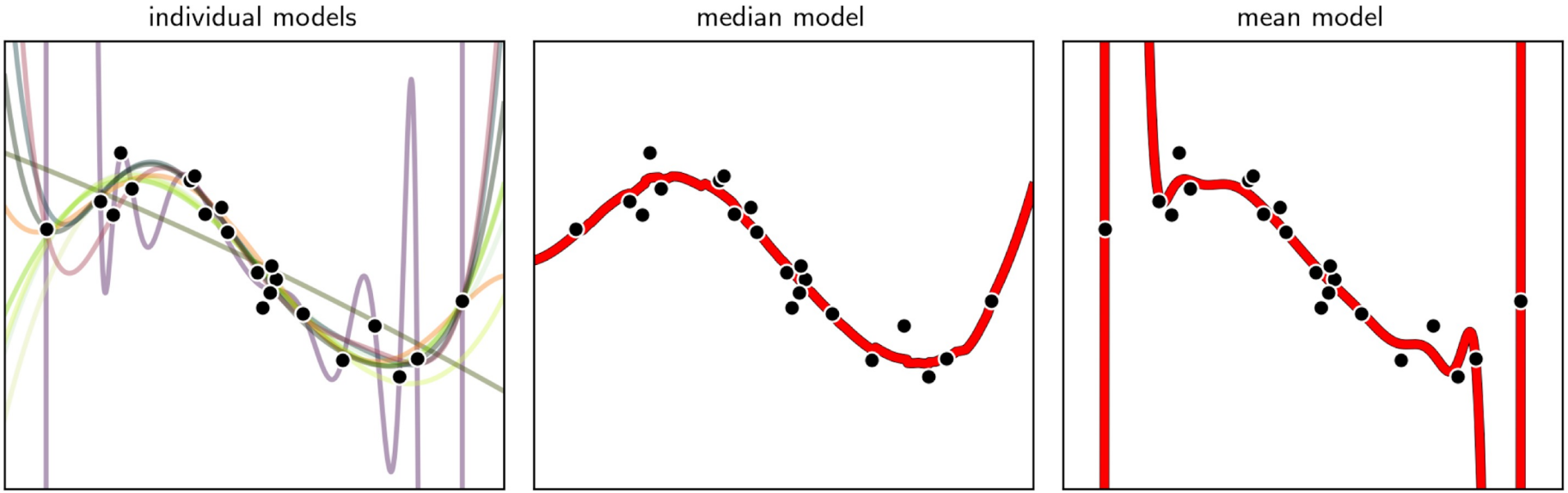
(large panel) The bagged (median) model of the 10 models whose fits are shown on the left.





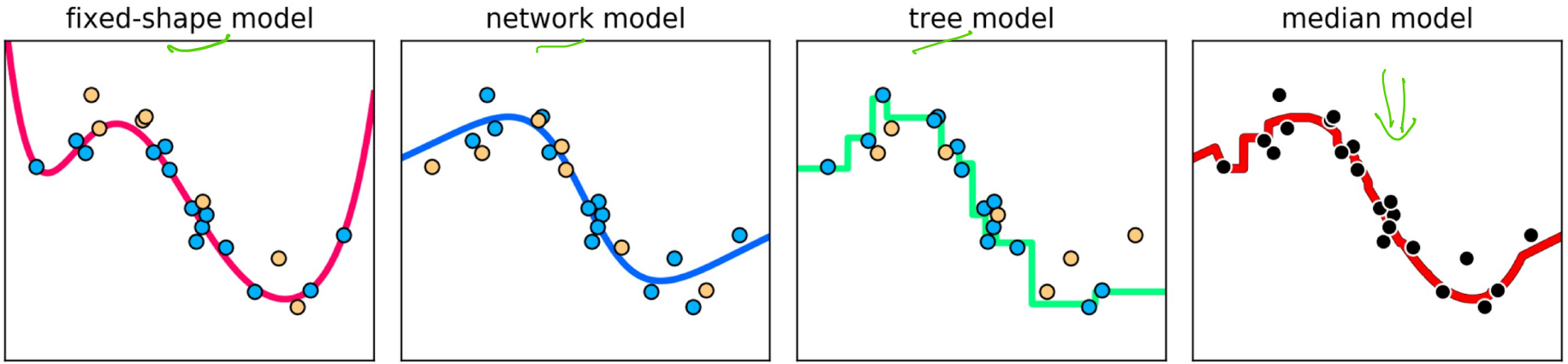
Why the median, not the mean?

Because generally speaking, **the mean is more sensitive to *outliers* the median.**



## Bagging polynomials, neural networks, and trees together

With bagging we can also effectively combine cross-validated models built from different universal approximators.



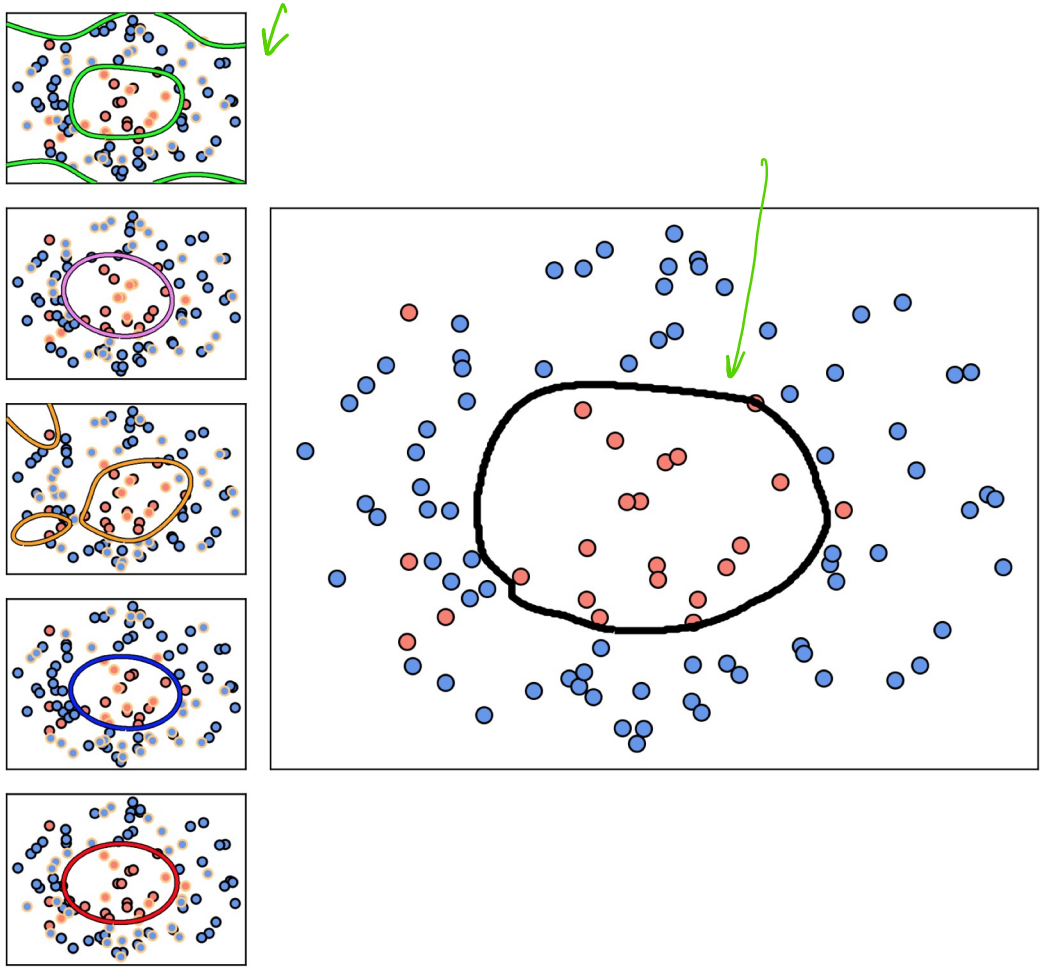
# Bagging classification models

Because the predicted output of a classification model is a *discrete* label, the average used to bag such cross-validated models is the mode (i.e., the most popularly predicted label).

# Example: Bagging cross-validated two-class classification models

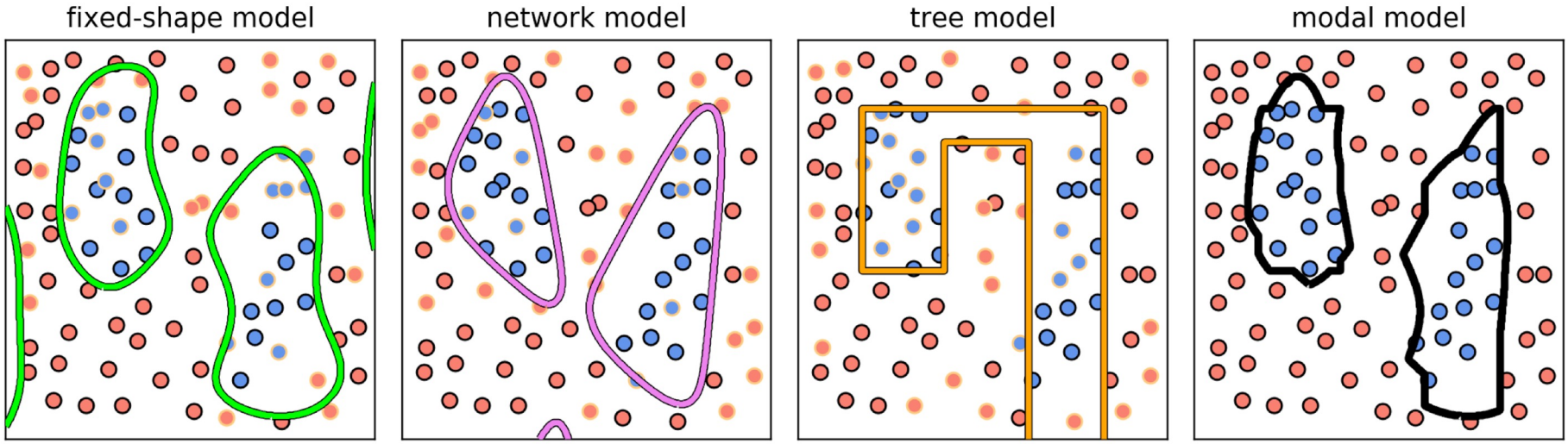
(small panels) Five models cross-validated on random training-validation splits of the data, with the validation data in each instance highlighted with a yellow outline.

(large panel) The bagged (modal) model of the 5 models shown on the left.



## Bagging different types of universal approximators

As with regression, with classification we can also combine cross-validated models built from different universal approximators.



How many models should we bag in practice?

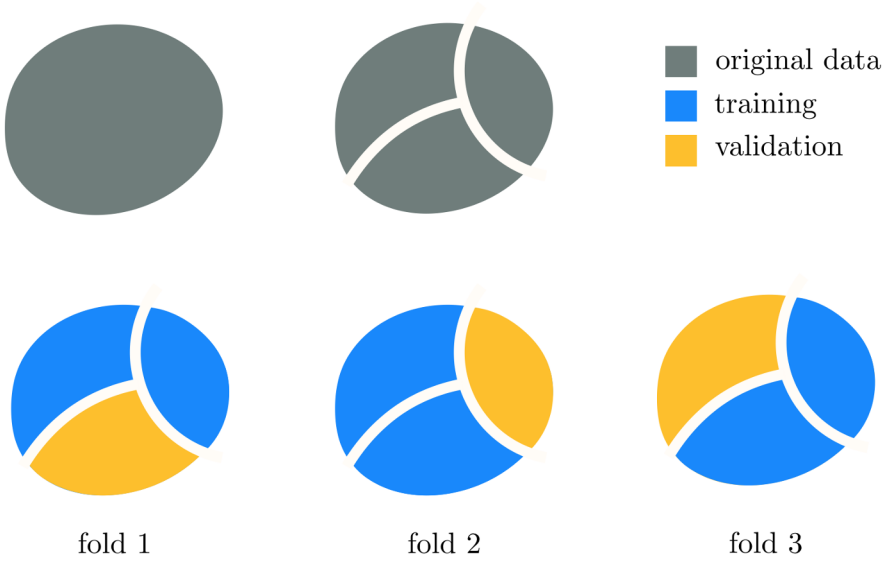
- There is general no magic number.
- The smaller the dataset, the less we could trust in the faithfulness of a random validation portion of it to represent the underlying phenomenon that generated the data, and hence we might wish to ensemble more of them to help average our poorly performing models resulting from bad splits of the data.
- Usually practical considerations like computation power as well as dataset size determine if bagging is used and if so how many models are employed in the average.

# Bagging vs. Boosting

- Bagging and boosting are both "ensembling" methods, as they are used to ensemble or combine different models to improve efficacy.
- With boosting we build up a single cross-validated model by gradually adding together simple models consisting of a single universal approximator unit. Each of these units are trained in a way that makes each individual model dependent on its predecessors (that are trained first).
- With bagging we average together multiple models that have been trained independently of each other. Indeed any one of those cross-validated models in a bagged ensemble can itself be a boosted model.

# 11.10 K-Fold Cross-Validation

- With ensembling **human interpretability** is typically lost as the final model is an average of many potentially very different nonlinearities.
- **K-fold cross-validation** is often applied when interpretability of a final model is important.
- Instead of averaging a set of cross-validated models over many splits of the data (ensembling), with K-fold cross-validation we choose a single model that has minimum ***average validation error*** over all splits of the data.
- This produces a potentially less accurate final model, but one that is significantly simpler and hence more easily understood by humans.




Schematic illustration of K-fold cross-validation for K=3. The original data (top left) is split into K non-overlapping sets or folds. In each instance we keep a different portion of the split data as validation while merging the remaining K-1 pieces as training.

# 11.11 When Feature Learning Fails

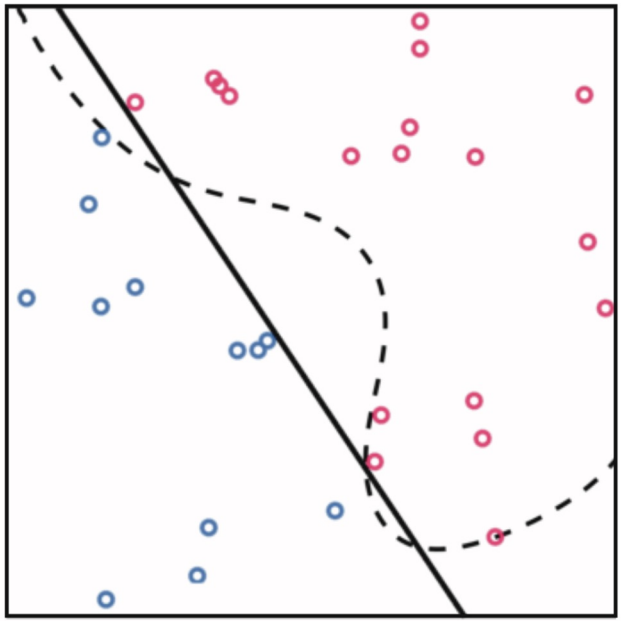
Feature learning fails when our data fails to sufficiently reflect the underlying phenomenon that generated it. This can happen when one or more of the following occur.

## When a dataset is too small.

When a dataset is too small to represent the true underlying phenomenon feature learning can inadvertently determine an incorrect nonlinearity.

dashed curve: true data-generating function

solid curve: learned function

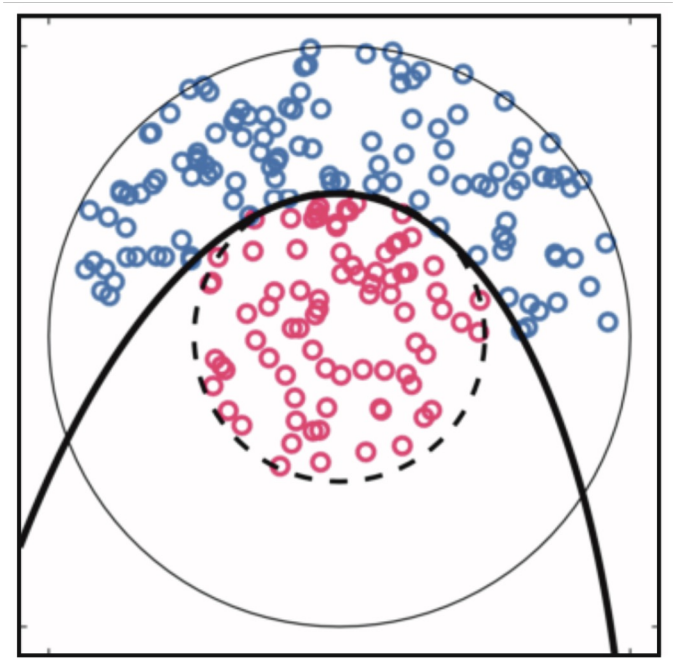


**When a dataset is poorly distributed.**

Even if a dataset is large it can still fail to reflect the true nature of the underlying phenomenon that generated it if the data is poorly distributed.

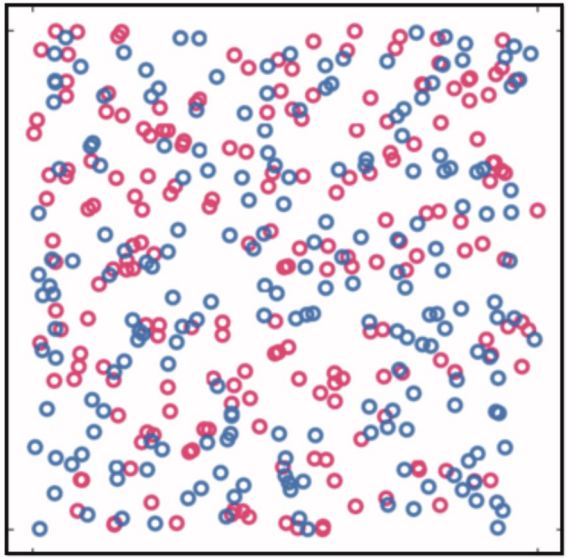
dashed curve: true data-generating function

solid curve: learned function



**When a dataset has no inherent structure.**

If there is little or no relation- ship present in the data (due to improper measurement, experimentation, or selection of inputs) the nonlinear model learned via feature learning will be useless.



# Summary

- What is Feature Learning
- Universal approximator
- Naïve Cross-Validation
- Cross-Validation via Boosting
- Cross-Validation via Regularization
- Testing data
- Which Universal Approximator works the Best
- Bagging
- When Feature Learning Fails