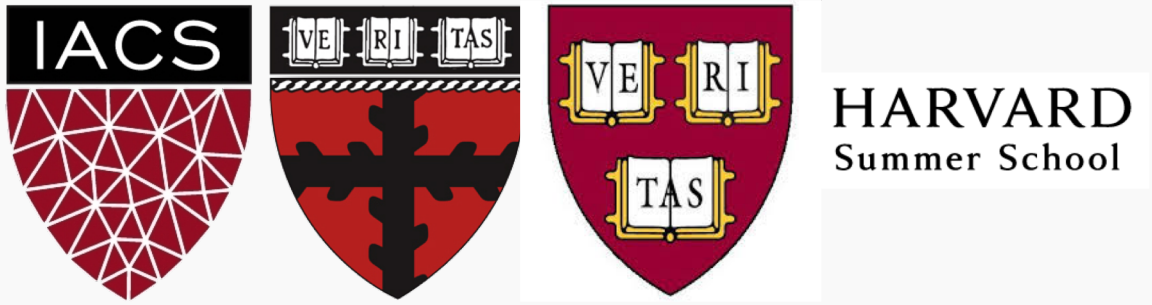
Lecture #10: Boosting & Stacking and an Introduction to Neural Network

CS-S109A: Introduction to Data Science

Kevin Rader



ANNOUNCEMENTS

- **HW5** is posted and due Tues night at 11:59pm (July 28)
- The **Final Exam** (individual work) will be posted July 27 (Monday) and due Monday, Aug. 3. It is cumulative and will be more open-ended. More details coming on Monday.
- **Friday's lab** will be very useful for implementing Neural Networks for the HW.

Outline

- Review of Ensemble Methods
- Boosting
  - Set-up and intuition
  - Connection to Gradient Descent
  - The Algorithm
- Stacking
- Artificial Neural Networks: an introduction
- Anatomy of a NN
- NN Design choices

Review of Ensemble Methods

(Bagging and Random Forests)

Bags and Forests of Trees

Last time we examined how the short-comings of single decision tree models can be overcome by ensemble methods - making one model out of many trees.

We focused on the problem of training large trees, these models have low bias but high variance.

We compensated by training an ensemble of full decision trees and then averaging their predictions - thereby reducing the variance of our final model.

Bags and Forests of Trees (cont.)

Bagging:

- create an ensemble of full trees, each trained on a bootstrap sample of the training set;
- average the predictions

Random forest:

- create an ensemble of full trees, each trained on a bootstrap sample of the training set;
- in each tree and each split, randomly select a subset of predictors, choose a predictor from this subset for splitting;
- average the predictions

Note that the ensemble building aspects of both method are embarrassingly parallel!

Motivation for Boosting

Could we address the shortcomings of single decision trees models in some other way?

For example, rather than performing variance reduction on complex trees, can we decrease the bias of simple trees - make them more expressive?

A solution to this problem, making an expressive model from simple trees, is another class of ensemble methods called ***boosting***.

Boosting Algorithms

Gradient Boosting

The key intuition behind boosting is that one can take an ensemble of simple models $\{T_h\}_{h \in H}$ and additively combine them into a single, more complex model.

Each model T_h might be a poor fit for the data, but a linear combination of the ensemble

$$T = \sum_h \lambda_h T_H$$

can be expressive/flexible.

But which models should we include in our ensemble? What should the coefficients or weights in the linear combination be?

Gradient Boosting: the algorithm

Gradient boosting is a method for iteratively building a complex regression model *T* by adding simple models. Each new simple model added to the ensemble compensates for the weaknesses of the current ensemble.

1. Fit a simple model $T^{(0)}$ on the training data

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

   Set $T \leftarrow T^{(0)}$. Compute the residuals $\{r_1, \ldots, r_N\}$ for T.

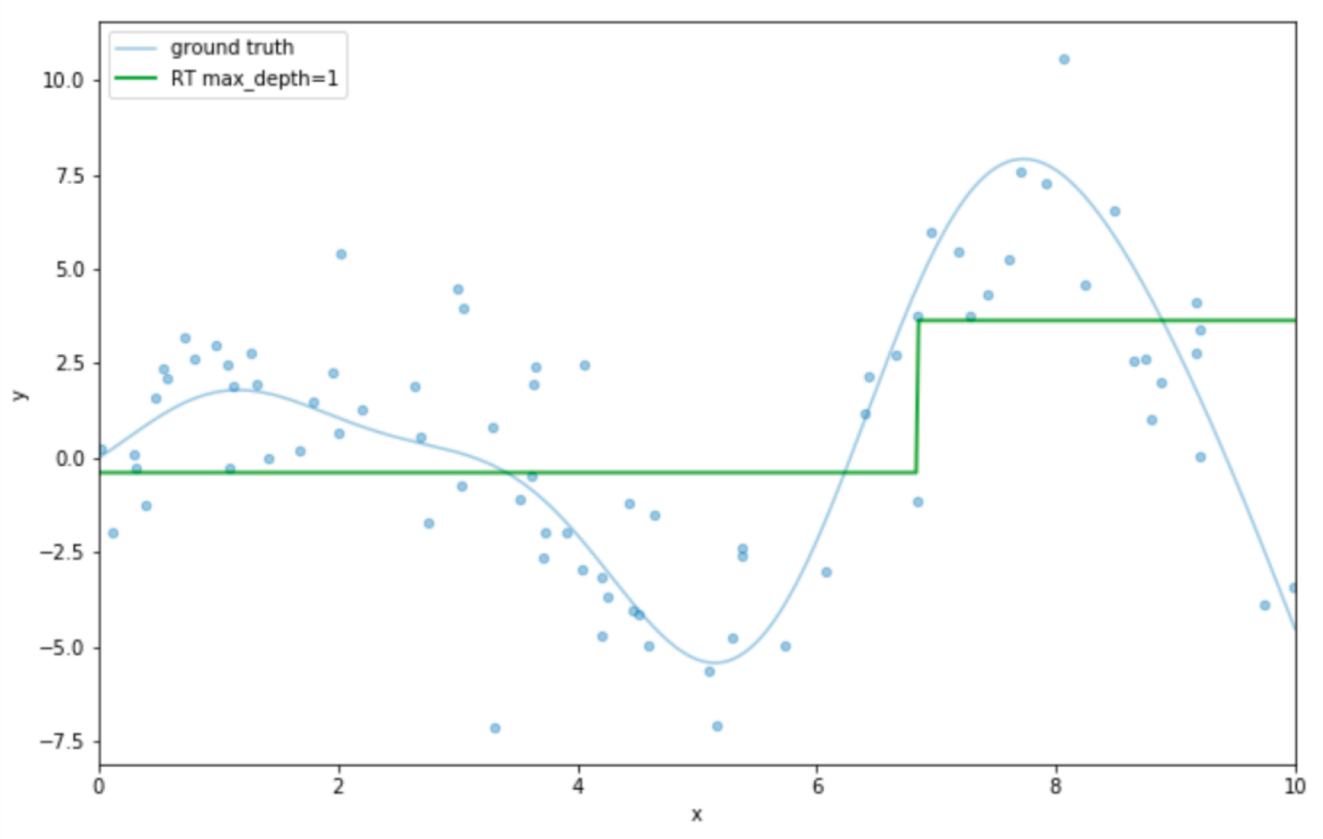
2. Fit a simple model, T^i, to the current ***residuals***, i.e. train using

$$\{(x_1, r_1), \ldots, (x_N, r_N)\}$$

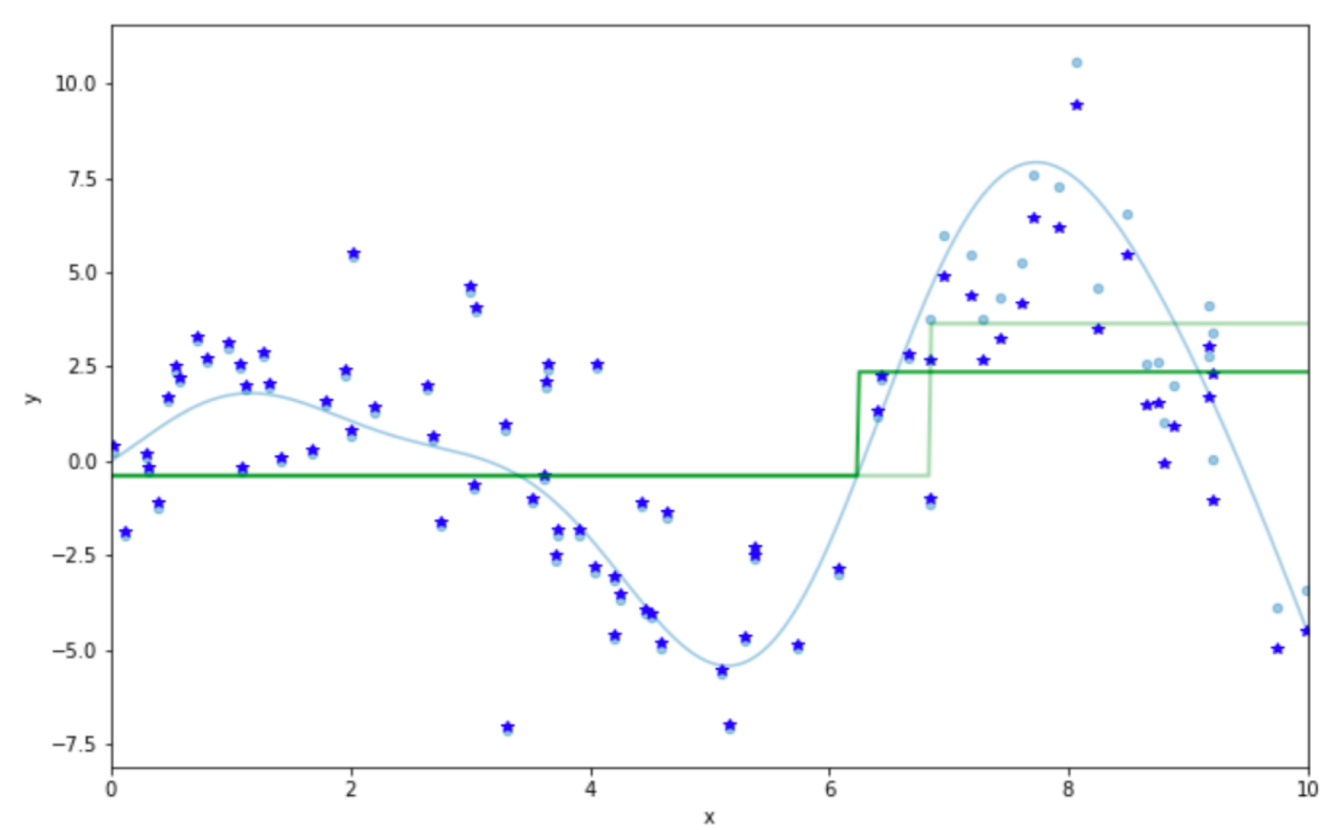
3. Set $T \leftarrow T + \lambda T^i$
4. Compute residuals, set $r_n \leftarrow r_n - \lambda T^i(x_n)$, $n = 1,\ldots,N$
5. Repeat steps 2-4 until ***stopping*** condition met.

where λ is a constant called the ***learning rate***.

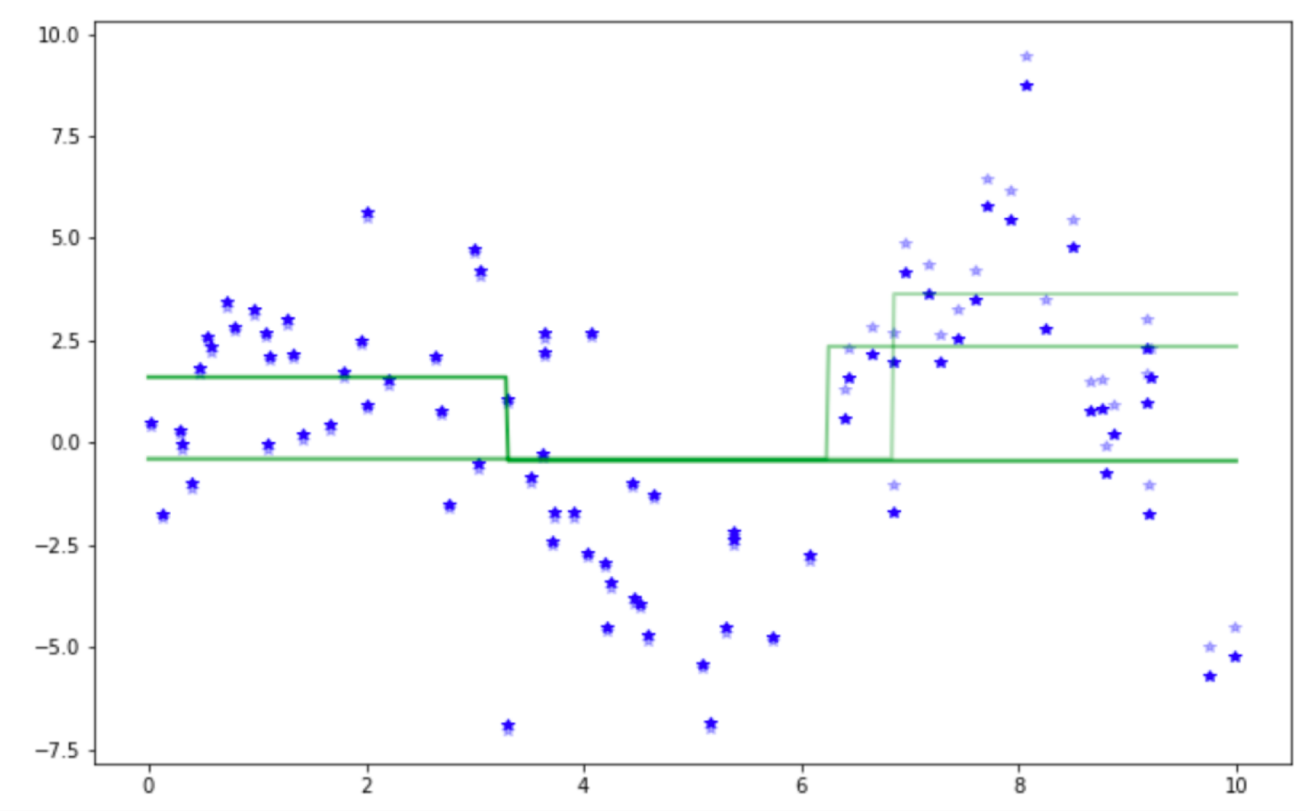
Gradient Boosting: illustration (0)



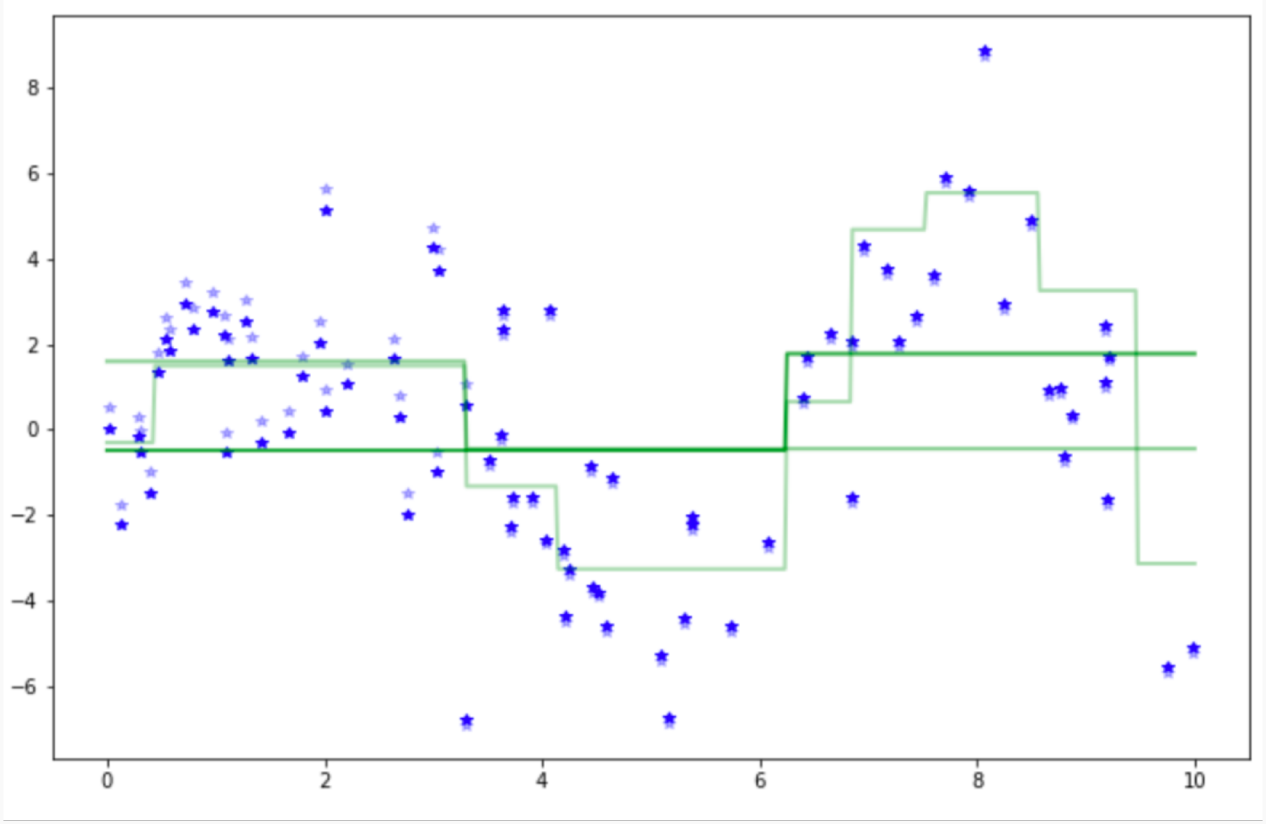
Gradient Boosting: illustration 1



Gradient Boosting: illustration 2



Gradient Boosting: illustration 3



Why Does Gradient Boosting Work?

Intuitively, each simple model $T^{(i)}$ we add to our ensemble model T, models the errors of T.

Thus, with each addition of $T^{(i)}$, the residual is reduced

$$r_n - \lambda T^{(i)}(x_n)$$

Note that gradient boosting has a tuning parameter, λ.

If we want to easily reason about how to choose λ and investigate the effect of λ on the model T, we need a bit more mathematical formalism.

In particular, how can we effectively descend through this optimization via an iterative algorithm?

We need to formulate gradient boosting as a type of ***gradient descent***.

Review: A Brief Sketch of Gradient Descent

In optimization, when we wish to minimize a function, called the ***objective function***, over a set of variables, we compute the partial derivatives of this function with respect to the variables.

If the partial derivatives are sufficiently simple, one can analytically find a common root - i.e. a point at which all the partial derivatives vanish; this is called a ***stationary point.***

If the objective function has the property of being ***convex***, then the stationary point is precisely the min.

Review: A Brief Sketch of Gradient Descent the Algorithm

In practice, our objective functions are complicated and analytically find the stationary point is intractable.

Instead, we use an iterative method called ***gradient descent***:

1. Initialize the variables at any value:

$$x = [x_1, ..., x_J]$$

2. Take the gradient of the objective function at the current variable values:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \ldots, \frac{\partial f}{\partial x_J}(x)\right]$$

3. Adjust the variables values by some negative multiple of the gradient:

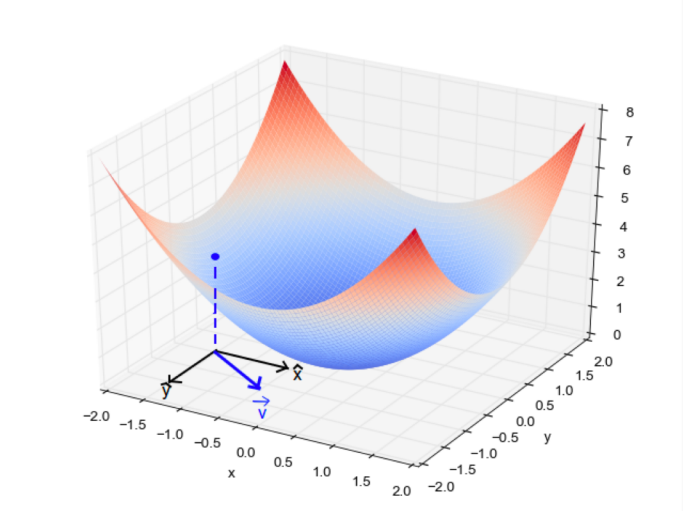
$$x \leftarrow x - \lambda \nabla f(x)$$

The factor λ is often called the learning rate.

Why Does Gradient Descent Work?

Claim: If the function is convex, this iterative methods will eventually move x close enough to the minimum, for an appropriate choice of λ.

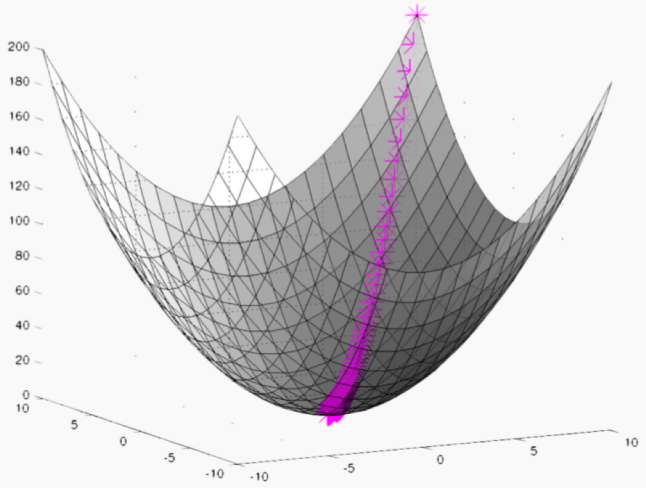
Why does this work? Recall, that as a vector, the gradient at at point gives the direction for the greatest possible rate of increase.



Why Does Gradient Descent Work?

Subtracting a λ multiple of the gradient from x, moves x in the ***opposite*** direction of the gradient (hence towards the steepest decline) by a step of size λ.

If *f* is convex, and we keep taking steps descending on the graph of *f*, we will eventually reach the minimum.



Gradient Boosting as Gradient Descent

Often in regression, our objective is to minimize the MSE

$$\text{MSE}(\hat{y}_1, \ldots, \hat{y}_N) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Treating this as an optimization problem, we can try to directly minimize the MSE with respect to the predictions

$$\begin{aligned} \nabla \text{MSE} &= \left[\frac{\partial \text{MSE}}{\partial \hat{y}_1}, \ldots, \frac{\partial \text{MSE}}{\partial \hat{y}_N} \right] \\ &= -2 \left[y_1 - \hat{y}_1, \ldots, y_N - \hat{y}_N \right] \\ &= -2 \left[r_1, \ldots, r_N \right] \end{aligned}$$

The update step for gradient descent would look like

$$\hat{y}_n \leftarrow \hat{y}_n + \lambda r_n, \quad n = 1, \ldots, N$$

Gradient Boosting as Gradient Descent (cont.)

There are two reasons why minimizing the MSE with respect to $\hat{y}_n$'s is not interesting:

- We know where the minimum MSE occurs: $\hat{y}_n = y_n$, for every n.
- Learning sequences of predictions, $\hat{y}_n^1, \ldots, \hat{y}_n^i, \ldots$, does not produce a model. The predictions in the sequences do not depend on the predictors!

Gradient Boosting as Gradient Descent (cont.)

The solution is to change the update step in gradient descent. Instead of using the gradient - the residuals - we use an ***approximation*** of the gradient that depends on the predictors:

$$\hat{y} \leftarrow \hat{y}_n + \lambda \hat{r}_n(x_n), \quad n = 1, \ldots, N$$

In gradient boosting, we use a simple model to approximate the residuals, $\hat{r}_n(x_n)$, in each iteration.

Motto: gradient boosting is a form of gradient descent with the MSE as the objective function.

Technical note: note that gradient boosting is descending in a space of models or functions relating x_n to y_n!

Gradient Boosting as Gradient Descent (cont.)

But why do we care that gradient boosting is gradient descent?

By making this connection, we can import the massive amount of techniques for studying gradient descent to analyze gradient boosting.

For example, we can easily reason about how to choose the learning rate λ in gradient boosting.

Choosing a Learning Rate

Under ideal conditions, gradient descent iteratively approximates and converges to the optimum.

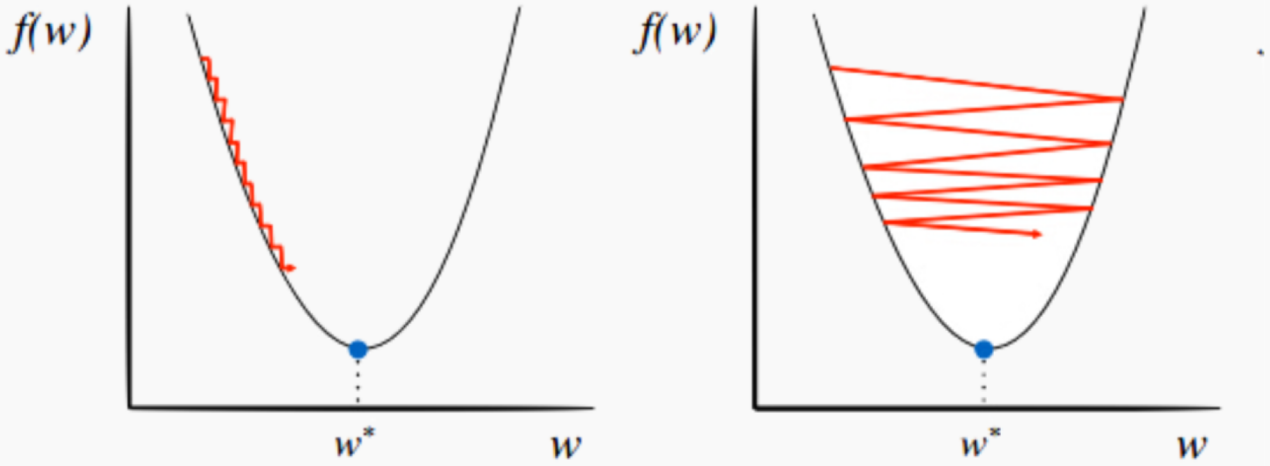
When do we terminate gradient descent?

- We can limit the number of iterations in the descent. But for an arbitrary choice of maximum iterations, we cannot guarantee that we are sufficiently close to the optimum in the end.
- If the descent is stopped when the updates are sufficiently small (e.g. the residuals of T are small), we encounter a new problem: the algorithm may never terminate!

Both problems have to do with the magnitude of the learning rate, λ.

Choosing a Learning Rate

For a constant learning rate, λ, if λ is too small, it takes too many iterations to reach the optimum.



If λ is too large, the algorithm may 'bounce' around the optimum and never get sufficiently close.

Choosing a Learning Rate

Choosing λ:

- If λ is a constant, then it should be tuned through cross validation.
- For better results, use a variable λ. That is, let the value of λ depend on the gradient

$$\lambda = h(\|\nabla f(x)\|),$$

where $\|\nabla f(x)\|$ is the magnitude of the gradient, $\nabla f(x)$. So

- around the optimum, when the gradient is small, λ should be small
- far from the optimum, when the gradient is large, λ should be larger

Motivation for AdaBoost

Using the language of gradient descent also allow us to connect gradient boosting for regression to a boosting algorithm often used for classification, AdaBoost.

In classification, we typically want to minimize the classification error:

$$\text{Error} = \frac{1}{N}\sum_{n=1}^{N} \mathbb{1}(y_n \neq \hat{y}_n), \quad \mathbb{1}(y_n \neq \hat{y}_n) = \begin{cases} 0, & y_n = \hat{y}_n \\ 1, & y_n \neq \hat{y}_n \end{cases}$$

Naively, we can try to minimize Error via gradient descent, just like we did for MSE in gradient boosting.

Unfortunately, Error is not differentiable with respect to the predictions, $\hat{y}$ ☹

Motivation for AdaBoost (cont.)

Our solution: we replace the Error function with a differentiable function that is a good indicator of classification error.

The function we choose is called ***exponential loss***

$$\text{ExpLoss} = \frac{1}{N}\sum_{n=1}^{N}\exp\left(-y_n\hat{y}_n\right),\ y_n \in \{-1, 1\}$$

Exponential loss is differentiable with respect to $\hat{y}_n$ and it is an upper bound of Error.

Gradient Descent with Exponential Loss

We first compute the gradient for ExpLoss:

$$\nabla \text{Exp} = [-y_1 \exp(-y_1 \hat{y}_1), \ldots, -y_N \exp(-y_N \hat{y}_N)]$$

It's easier to decompose each $y_n \exp(-y_n \hat{y}_n)$ as $w_n y_n$, where $w_n = \exp(-y_n \hat{y}_n)$.

This way, we see that the gradient is just a re-weighting applied the target values

$$\nabla \text{Exp} = [-w_1 y_1, \ldots, -w_N y_N]$$

Notice that when $y_n = \hat{y}_n$, the weight w_n is small; when $y_n \neq \hat{y}_n$, the weight is larger.

Gradient Descent with Exponential Loss

The update step in the gradient descent is

$$\hat{y}_n \leftarrow \hat{y}_n - \lambda w_n y_n, \quad n = 1, \ldots, N$$

Just like in gradient boosting, we approximate the gradient, $\lambda w_n y_n$ with a simple model, $T^{(i)}$, that depends on x_n.

This means training $T^{(i)}$ on a re-weighted set of target values,

$$\{(x_1, w_1 y_1), \ldots, (x_N, w_N, y_N)\}$$

That is, gradient descent with exponential loss means iteratively training simple models that ***focuses on the points misclassified by the previous model.***

AdaBoost

With a minor adjustment to the exponential loss function, we have the algorithm for gradient descent:

1. Choose an initial distribution over the training data, $w_n = 1/N$.
2. At the i^{th} step, fit a simple classifier $T^{(i)}$ on weighted training data

$$\{(x_1, w_1 y_1), ..., (x_N, w_N, y_N)\}$$

3. Update the weights:

$$w_n \leftarrow \frac{w_n \exp(-\lambda^{(i)} y_n T^{(i)}(x_n))}{Z}$$

where Z is the normalizing constant for the collection of updated weights

4. Update T: $T \leftarrow T + \lambda^{(i)} T^{(i)}$

where λ is the learning rate.

Choosing the Learning Rage

Unlike in the case of gradient boosting for regression, we can analytically solve for the optimal learning rate for AdaBoost, by optimizing:

$$\underset{\lambda}{\text{argmin}} \frac{1}{N} \sum_{n=1}^{N} \exp\left[-y_n(T + \lambda^{(i)} T^{(i)}(x_n))\right]$$

Doing so, we get that

$$\lambda^{(i)} = \frac{1}{2} \ln \frac{1-\epsilon}{\epsilon}, \quad \epsilon = \sum_{n=1}^{N} w_n \mathbb{1}(y_n \neq T^{(i)}(x_n))$$

Boosting in sklearn

Python has boosting algorithms implemented for you:

- **`sklearn.ensemble.AdaBoostClassifier`**
- **`sklearn.ensemble.AdaBoostRegressor`**
- With arguments of **base_estimator** (what models to use), **n_estimators** (max number of models to use), **learning_rate** (λ), etc…

Stacking

Motivation for Stacking

Recall that in boosting, the final model T, we learn is a weighted sum of simple models, T_h,

$$T = \sum_h \lambda_h T_H$$

where λ_h is the learning rate. In AdaBoost for example, we can analytically determine the optimal values of λ_h for each simple model T_h.

On the other hand, we can also determine the final model T implicitly by ***learning any model, called meta-learner, that transforms the outputs of T_h into a prediction***.

Stacked Generalization

The framework for ***stacked generalization*** or ***stacking*** (Wolpert 1992) is:

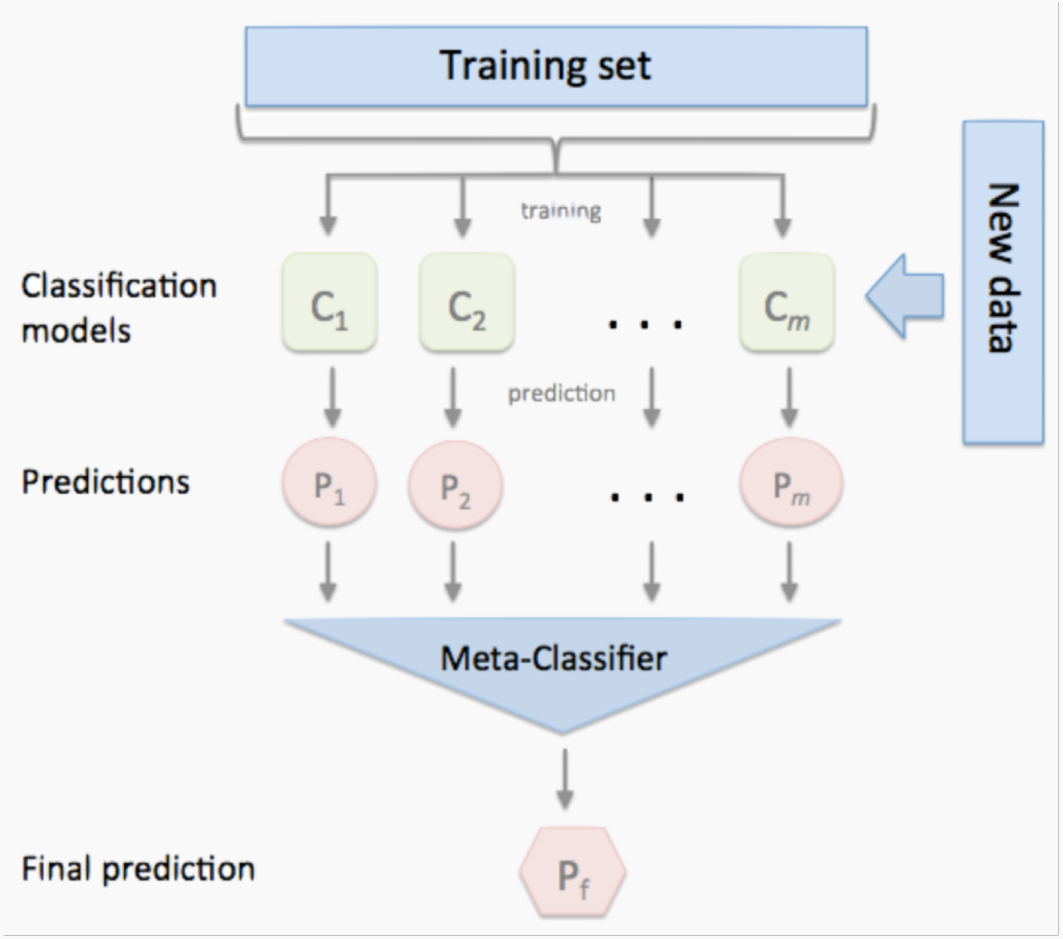
$$\{(x_1, y_1), ..., (x_N, y_N)\}$$

- train L number of models, T_l on the training data

$$\{(T_1(x_1), ..., T_L(x_1), y_1), ..., (T_1(x_N), ..., T_L(x_N), y_N)\}$$

- train a meta-learner $\tilde{T}$ on the predictions of the ensemble of models, i.e. train using the data

Stacking: an Illustration



Stacking: General Guidelines

The flexibility of stacking makes it widely applicable but difficult to analyze theoretically. Some general rules have been found through empirical studies:

- models in the ensemble should be diverse, i.e. their errors should not be correlated
- for binary classification, each model in the ensemble should have error rate < 1/2
- if models in the ensemble outputs probabilities, it's better to train the meta-learner on probabilities rather than predictions
- apply regularization to the meta-learner to avoid overfitting

Stacking: Subsemble Approach

We can extend the stacking framework to include ensembles of models that specialize on small subsets of data (Sapp et. al. 2014), for de-correlation or improved computational efficiency:

- divide the data in to J subsets
- train models, T_j , on each subset
- train a meta-learner $\tilde{T}$ on the predictions of the ensemble of models, i.e. train using the data

$$\{(T_1(x_1), ..., T_J(x_1), y_1), ..., (T_1(x_N), ..., T_J(x_N), y_N)\}$$

Again, we want to make sure that each $T_j(x_i)$ is an out of sample prediction.

Stacking in sklearn

Unfortunately, Python does not have stacking algorithms implemented for you ☹

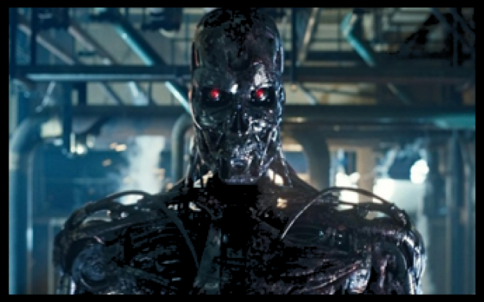
So how can we do it?

We can set it up by 'manually' fitting several base models, take the outputs of those models, and fitting the meta model on the outputs of those base models.

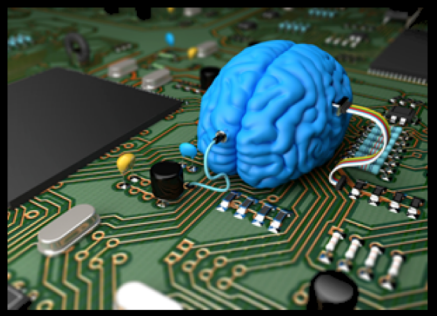
It's a model on models!

Artificial Neural Networks (ANN)

Deep Learning

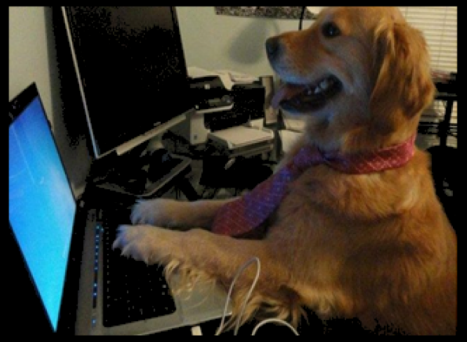


What society thinks I do



What my friends think I do

What other computer scientists think I do



What mathematicians think I do

What I think I do

```
In [1]:
import keras
Using TensorFlow backend.
```

What I actually do

Today's news

An AI just beat top lawyers at their own game

Share on F Share on +

IMAGE: BOB AL-GREEN/MASHABLE

BY MONICA CHIN

FEB 26 2018

The nation's top lawyers recently battled artificial intelligence in a competition to interpret contracts — and they lost.

A new study, conducted by legal AI platform LawGeex in consultati Share Quote ssors from Stanford University, Duke University School of Law, and University of Southern California, pitted twenty experienced lawyers against an AI trained to evaluate legal contracts.

Competitors were given four hours to review five non-disclosure agreements (NDAs) and identify 30 legal issues, including arbitration, confidentiality of relationship, and indemnification. They were scored by how accurately they identified each issue.

SEE ALSO: Google's new AI can predict heart disease by simply scanning your eyes

Google's new AI can predict heart disease by simply scanning your eyes

Share on F Share on +

IMAGE: BEN BRAIN/DIGITAL CAMERA MAGAZINE VIA GETTY IMAGES

BY MONICA CHIN

FEB 26 2018

The secret to identifying certain health conditions may be hidden in our eyes.

Researchers from Google and its health-tech subsidiary Verily announced on Monday that they have successfully created algorithms to predict whether someone has high blood pressure or is at risk of a heart attack or stroke simply by scanning a person's eyes, the *Washington Post* reports.

SEE ALSO: This fork helps you stay healthy

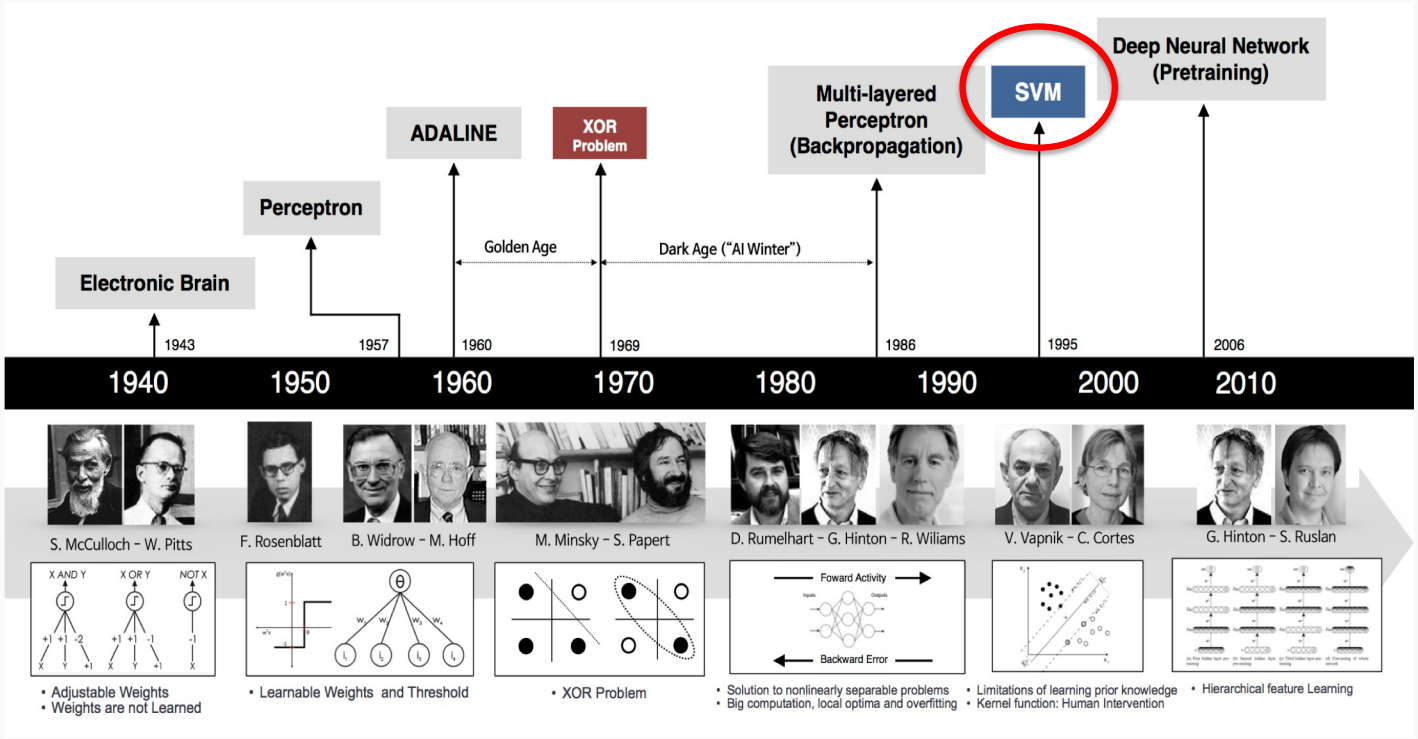
Google's researchers trained the algorithm with images of scanned retinas from more than 280,000 patients. By reviewing this massive database, Google's algorithm trained itself to recognize the patterns that designated people as at-risk.

This algorithm's success is a sign of exciting developments in healthcare on the horizon. As Google fine-tunes the technology, it could one day

AlphaGo (2015)

First program to beat a professional Go player

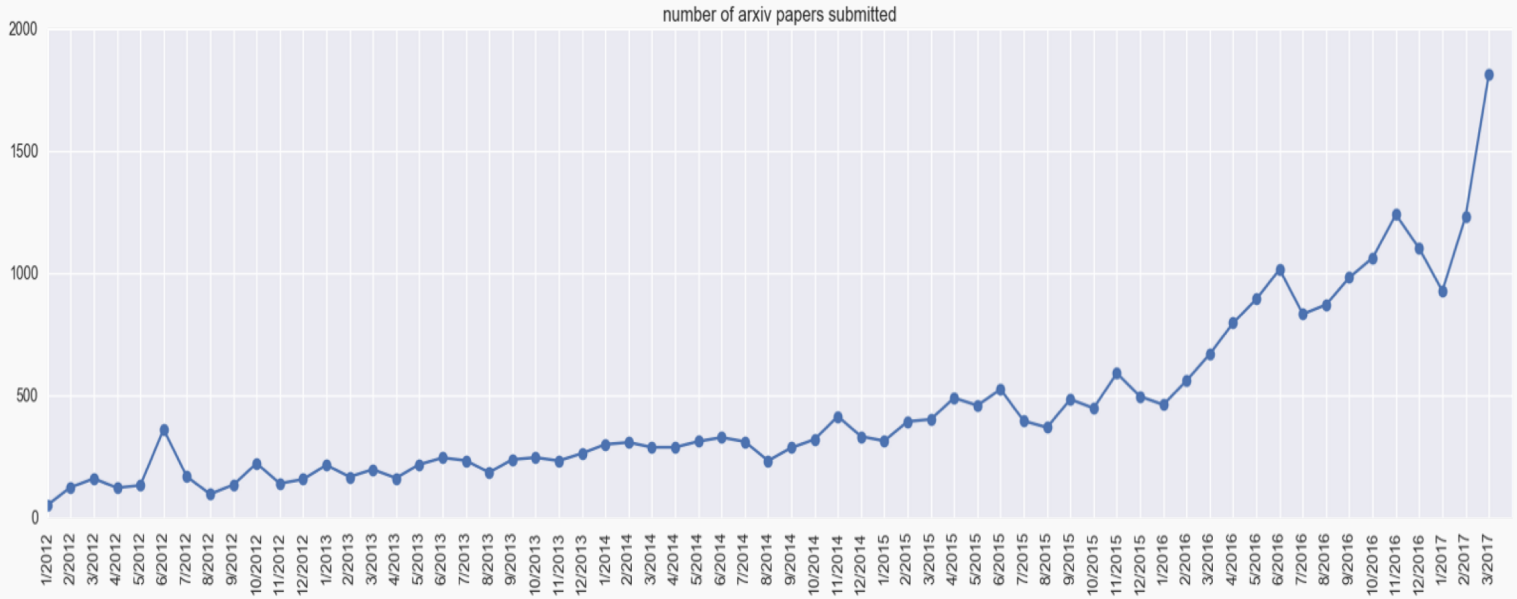
Historical Trends



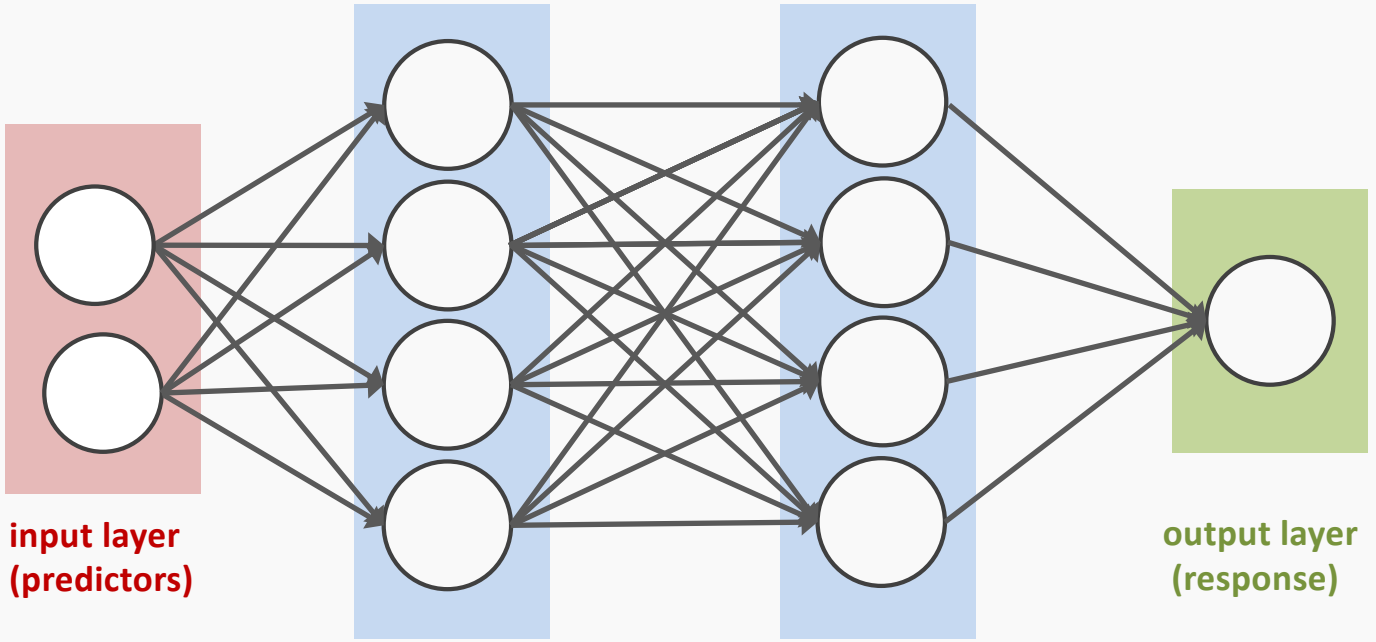
beamandrew.github.io/

Historical Trends

ArXiv papers on deep learning: 2012-2017



Anatomy of artificial neural network (ANN)



Translating Logistic Regression into NN language

Regression and Classification

Methods that are centered around modeling and prediction of a **quantitative** response variable (ex, number of taxi pickups, number of bike rentals, etc) are called **regressions** (and Ridge, LASSO, etc).

When the response variable is **categorical**, then the problem is no longer called a regression problem but is instead labeled as a **classification problem**.

The goal is to attempt to classify each observation into a category (aka, class or cluster) defined by *Y*, based on a set of predictor variables *X*.

Typical Classification Examples

The motivating examples for this lecture(s), homework, and labs are based on classification. Classification problems are common in these domains:

- Trying to determine where to set the *cut-off* for some diagnostic test (pregnancy tests, prostate or breast cancer screening tests, etc...)
- Trying to determine if cancer has gone into remission based on treatment and various other indicators
- Trying to classify patients into types or classes of disease based on various genomic markers

Heart Data

response variable ***Y*** is Yes/No

Age	Sex	ChestPain	RestBP	Chol	Fbs	RestECG	MaxHR	ExAng	Oldpeak	Slope	Ca	Thal	AHD
63	1	typical	145	233	1	2	150	0	2.3	3	0.0	fixed	No
67	1	asymptomatic	160	286	0	2	108	1	1.5	2	3.0	normal	Yes
67	1	asymptomatic	120	229	0	2	129	1	2.6	2	2.0	reversable	Yes
37	1	nonanginal	130	250	0	0	187	0	3.5	3	0.0	normal	No
41	0	nontypical	130	204	0	2	172	0	1.4	1	0.0	normal	No

Logistic Regression

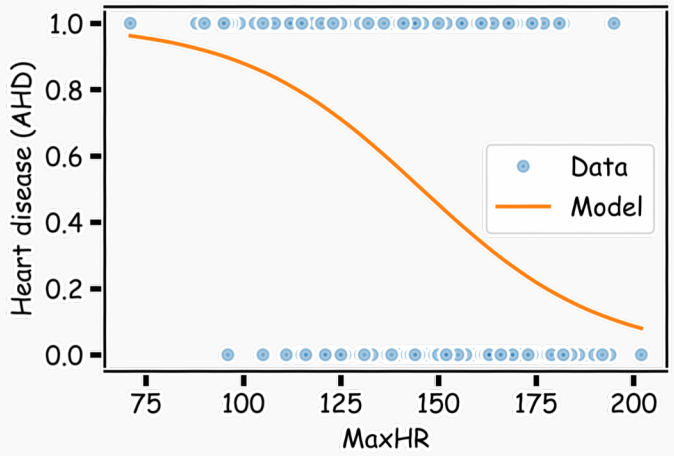
Logistic Regression addresses the problem of estimating a probability, $P(y = 1)$, given an input X. The logistic regression model uses a function, called the ***logistic*** function, to model $P(y = 1)$:

$$P(Y = 1) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

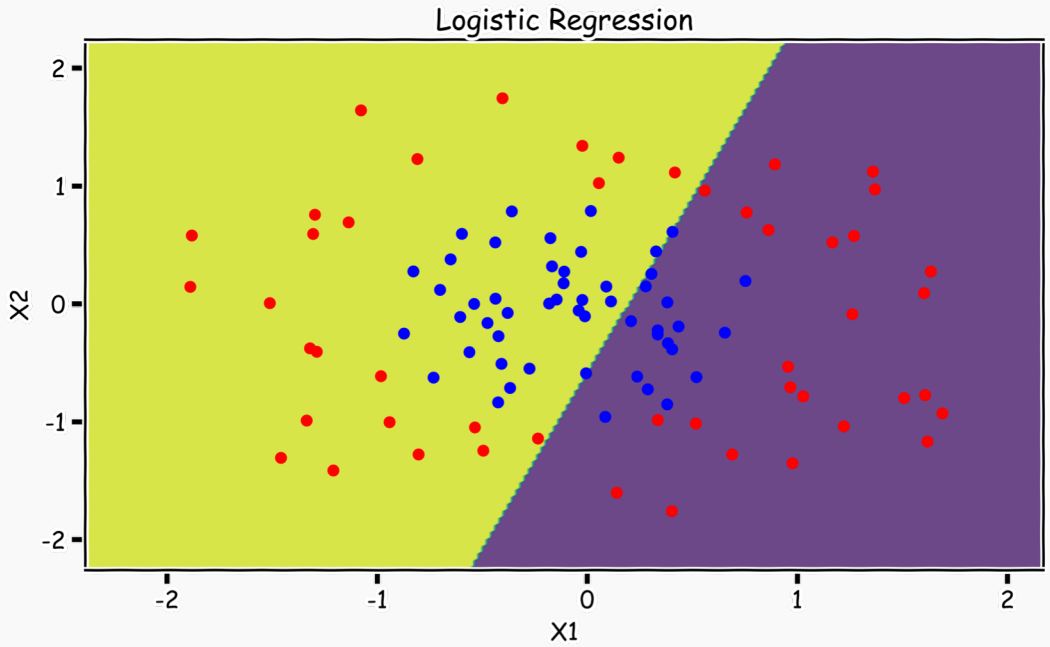
Estimating the coefficients for Logistic Regression

Find the coefficients that minimize the loss function

$$\mathcal{L}(\beta_0, \beta_1) = -\sum_i [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

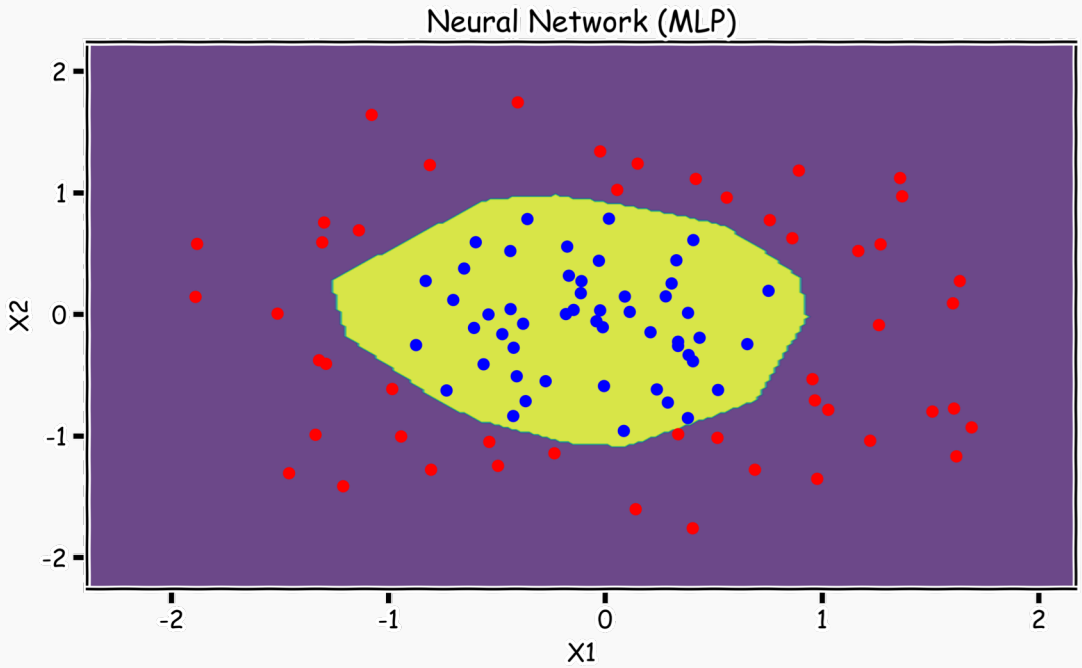


Need for Non-Linearity



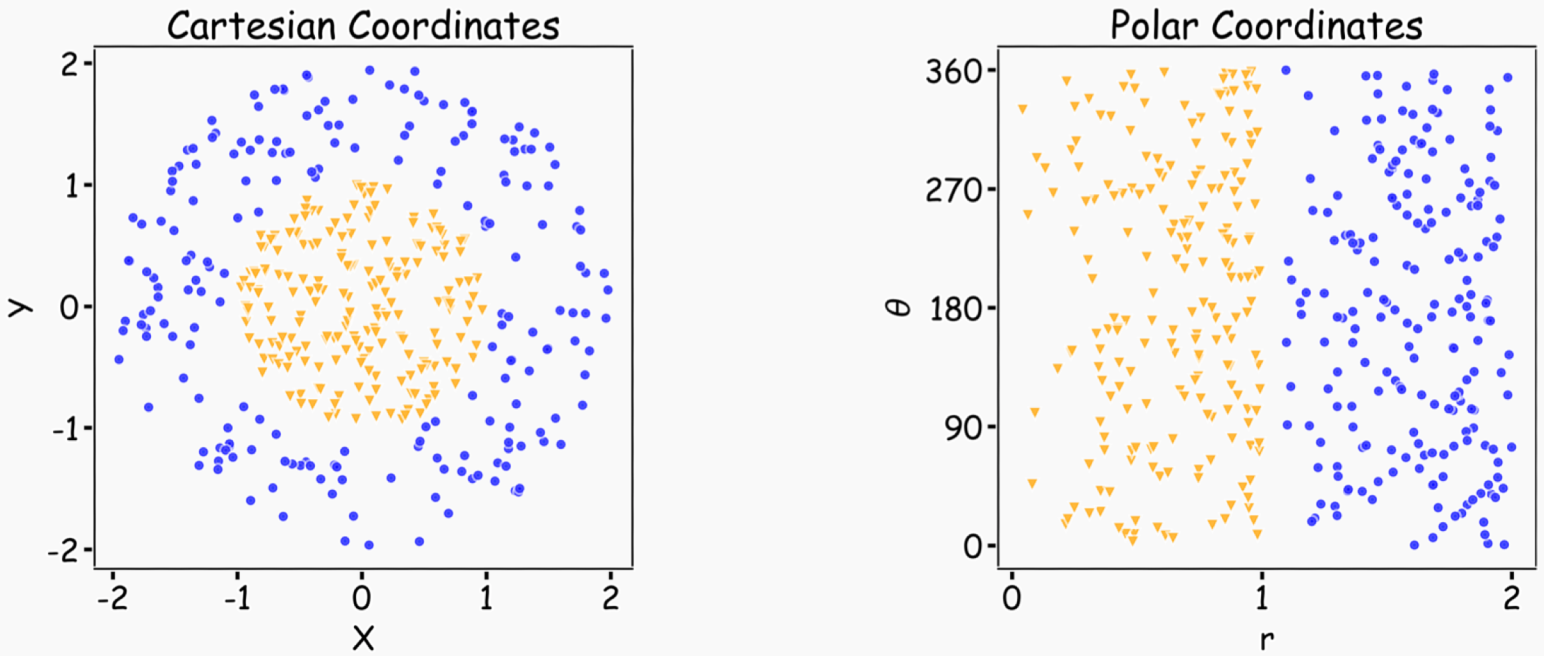
Without **augmenting the features** (i.e. without adding X_1^2 or X_2^2 non-linear features), **Logistic Regression is incapable of modeling the correct decision boundary.**

Neural Networks to The Rescue



A **neural network is a powerful non-linear** model that can easily **model the non-linear decision boundary correctly.**

Representation Matters



Neural networks can **learn useful representations** for the problem. This is another reason why they can be so powerful!

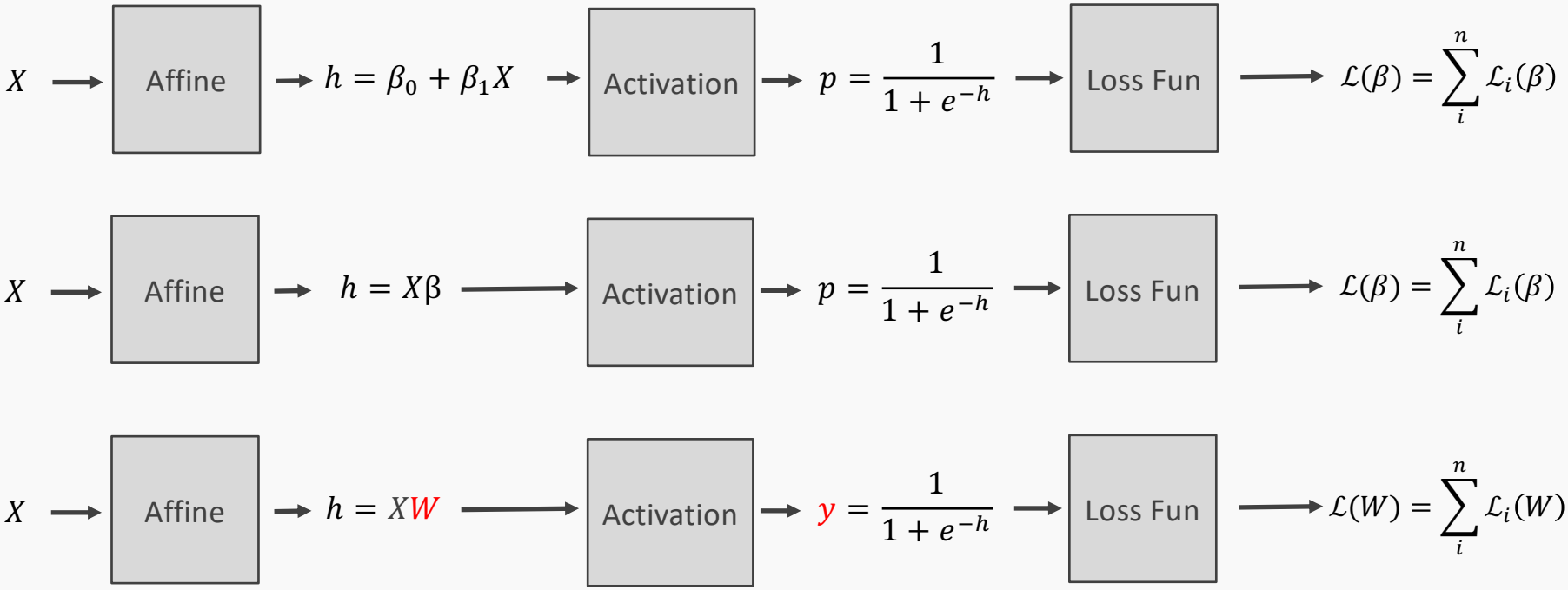
Logistic Regression Revisited (with new terminology)

$$P(Y = 1) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 X)}}$$

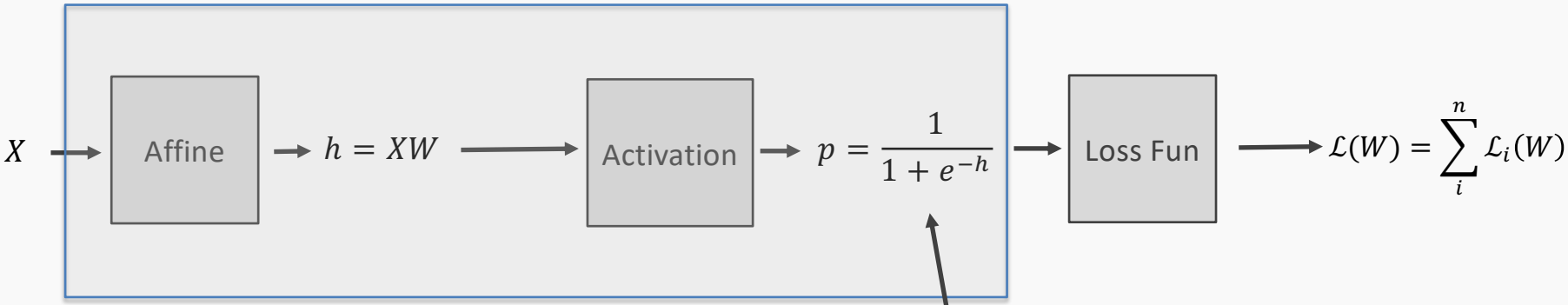
$$\mathcal{L}(\beta_0, \beta_1) = -\sum_i [y_i \log p_i + (1 - y_i) \log(1 - p_i)]$$

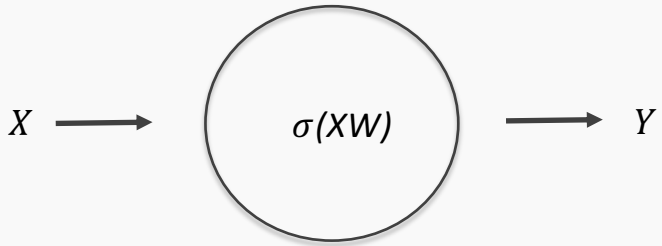
x_i → Affine → $h_i = \beta_0 + \beta_1 x_i$ → Activation → $p_i = \frac{1}{1 + e^{-h_i}}$ → Loss Fun → $\mathcal{L}_i(\beta) = -y_i \ln(p_i) - (1 - y_i)\ln(1 - p_i)$

Build our first ANN



Build our first ANN



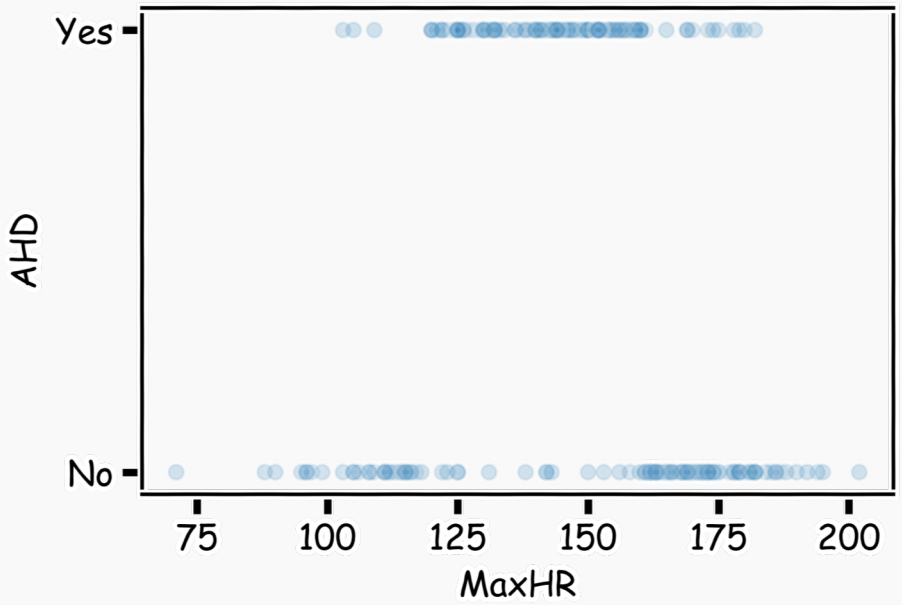


Single Neuron Network

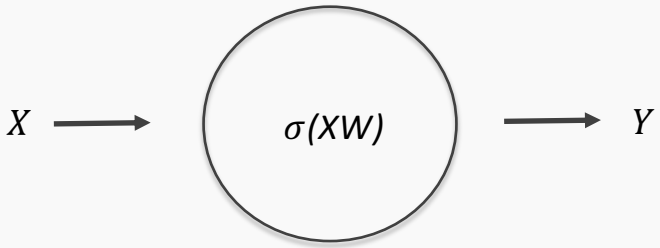
Very similar to Perceptron

Example Using Heart Data

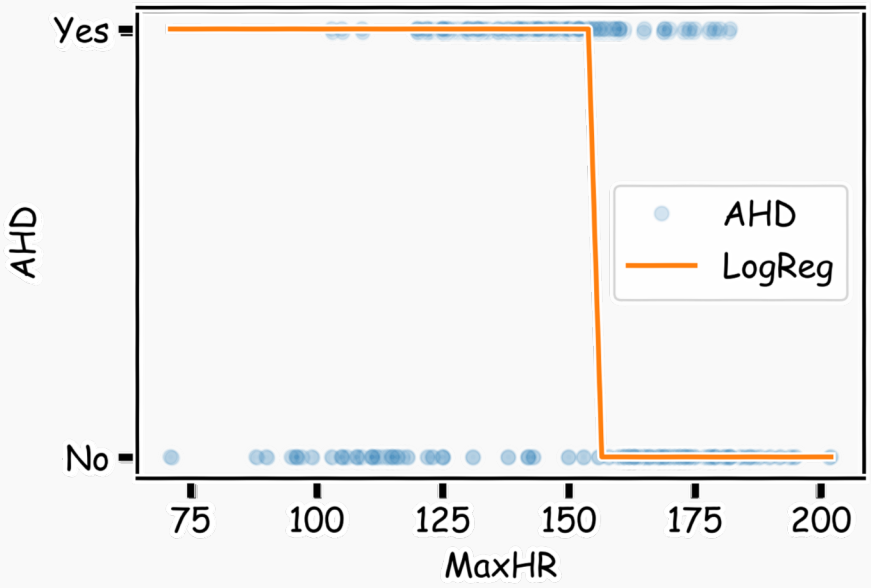
Slightly modified data to illustrate concepts.



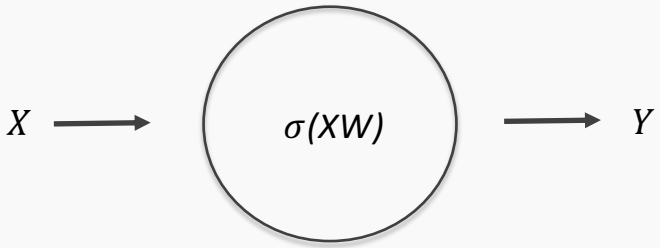
Example Using Heart Data



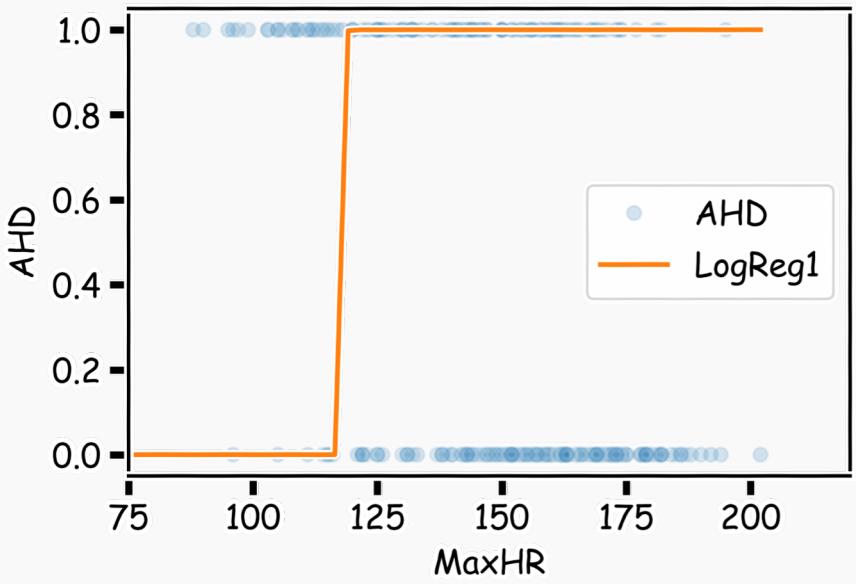
Choose W such as



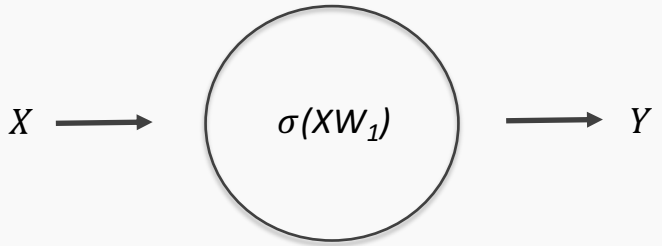
Example Using Heart Data

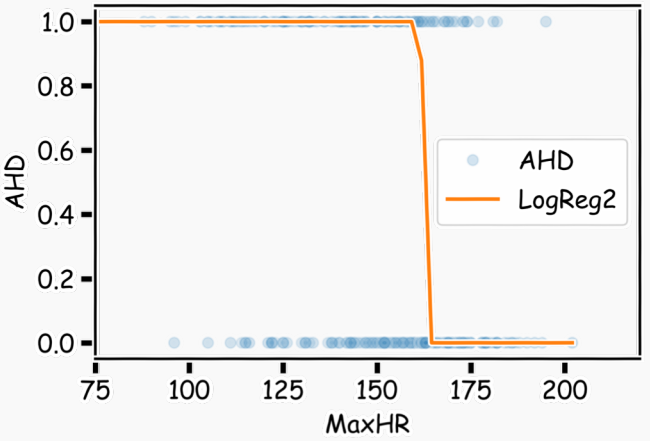


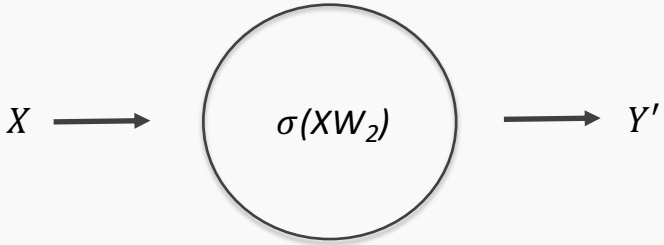
Choose W such as

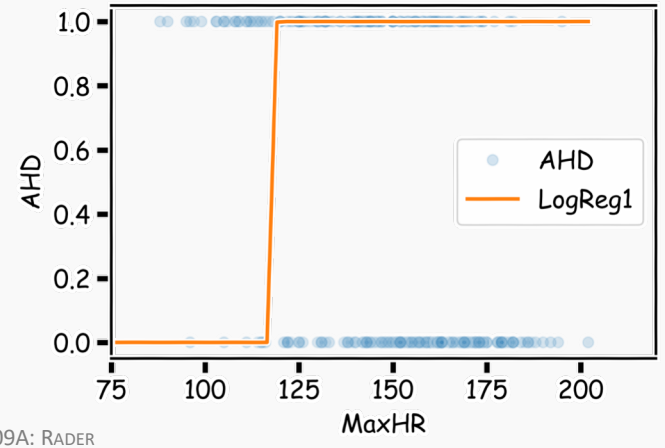


Example

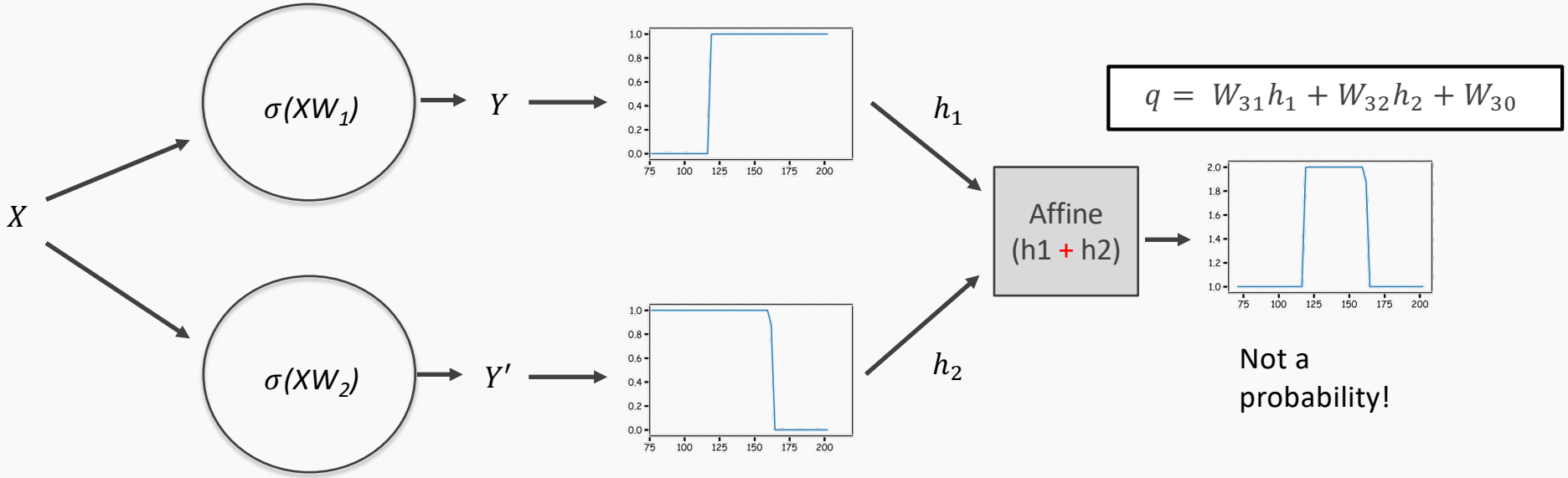

X
$\sigma(XW_1)$
Y


1.0
0.8
0.6
0.4
0.2
0.0
AHD
75
100
125
150
175
200
MaxHR
AHD
LogReg2

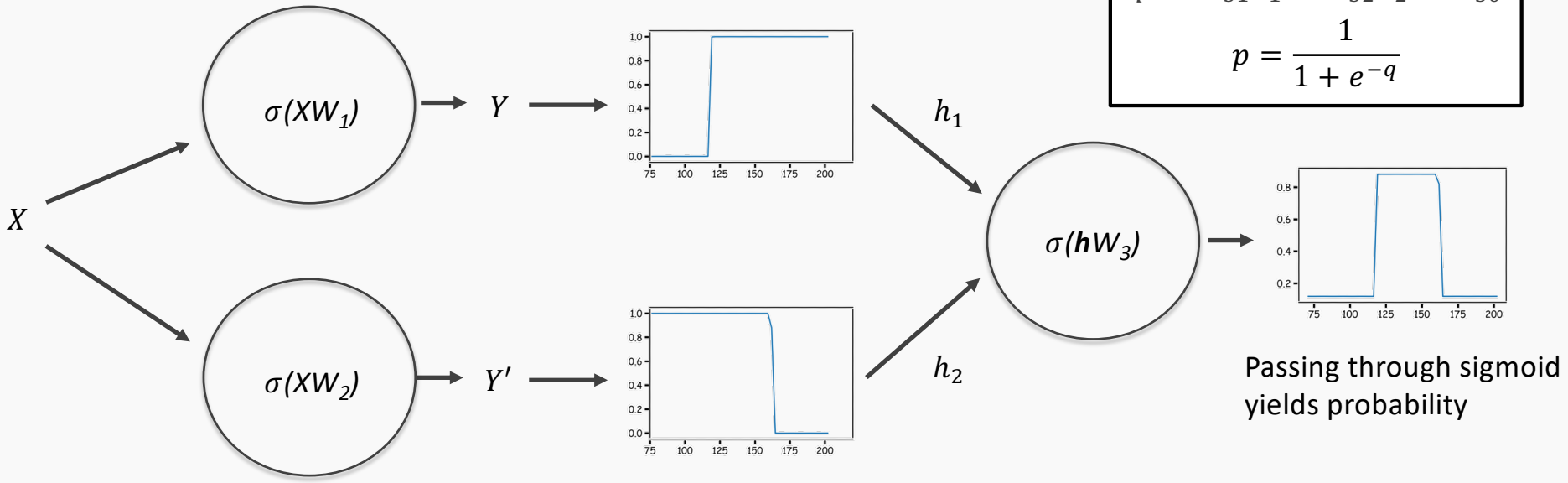

X
$\sigma(XW_2)$
Y'


1.0
0.8
0.6
0.4
0.2
0.0
AHD
75
100
125
150
175
200
MaxHR
AHD
LogReg1

Let's combine them!

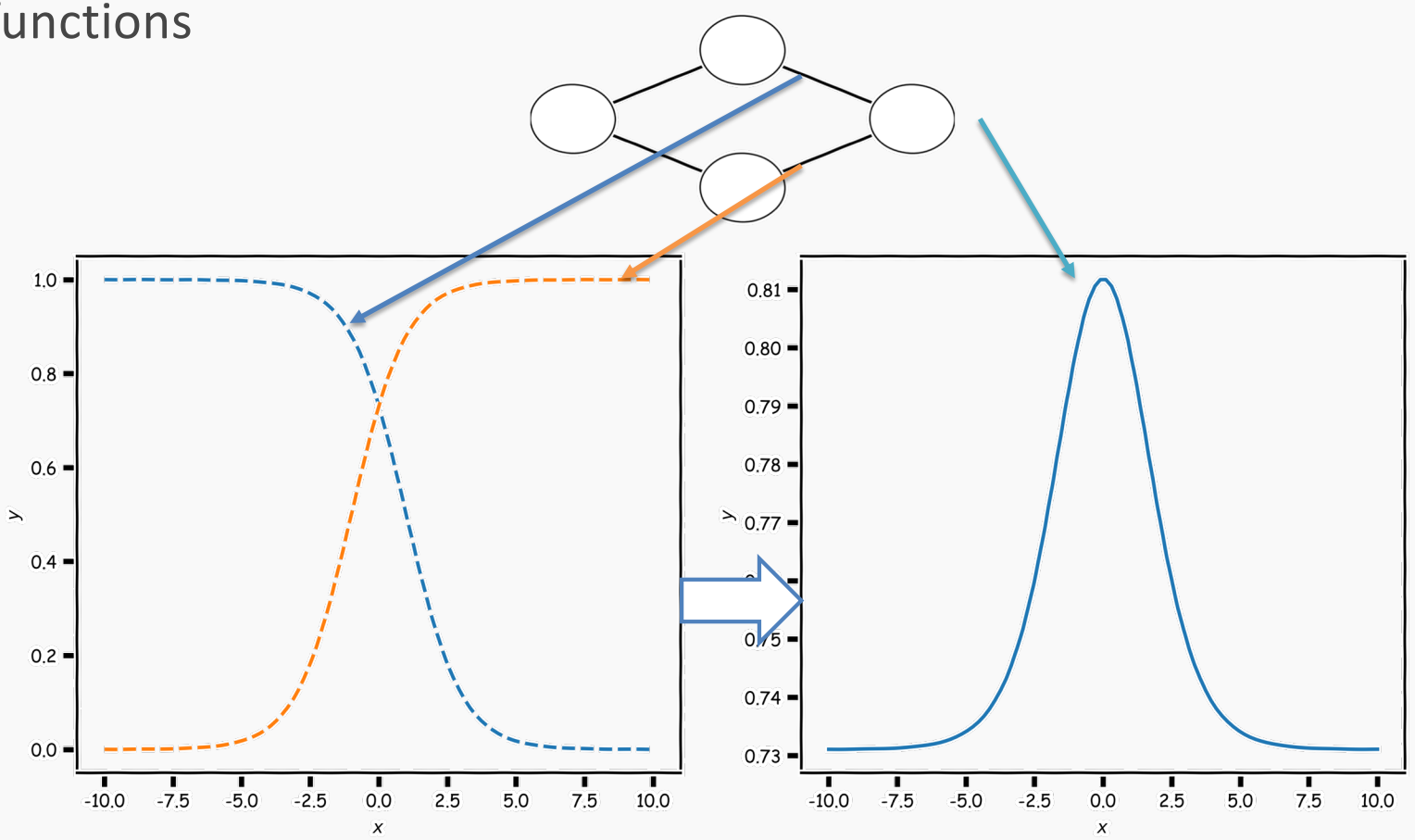


What's the output?

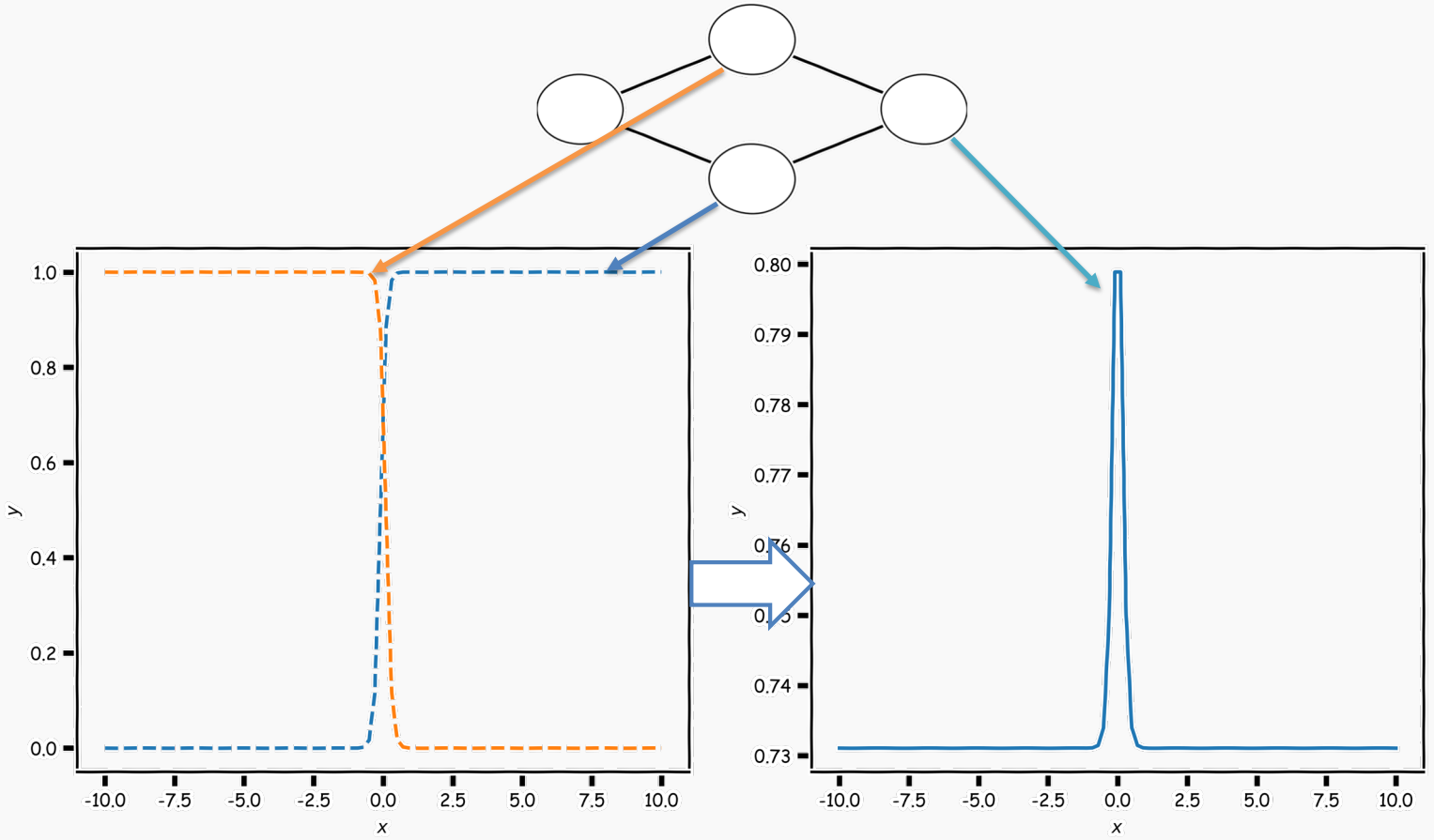


Need to learn W_1 (2 parameters), W_2 (2 parameters), and W_3 (3 parameters).

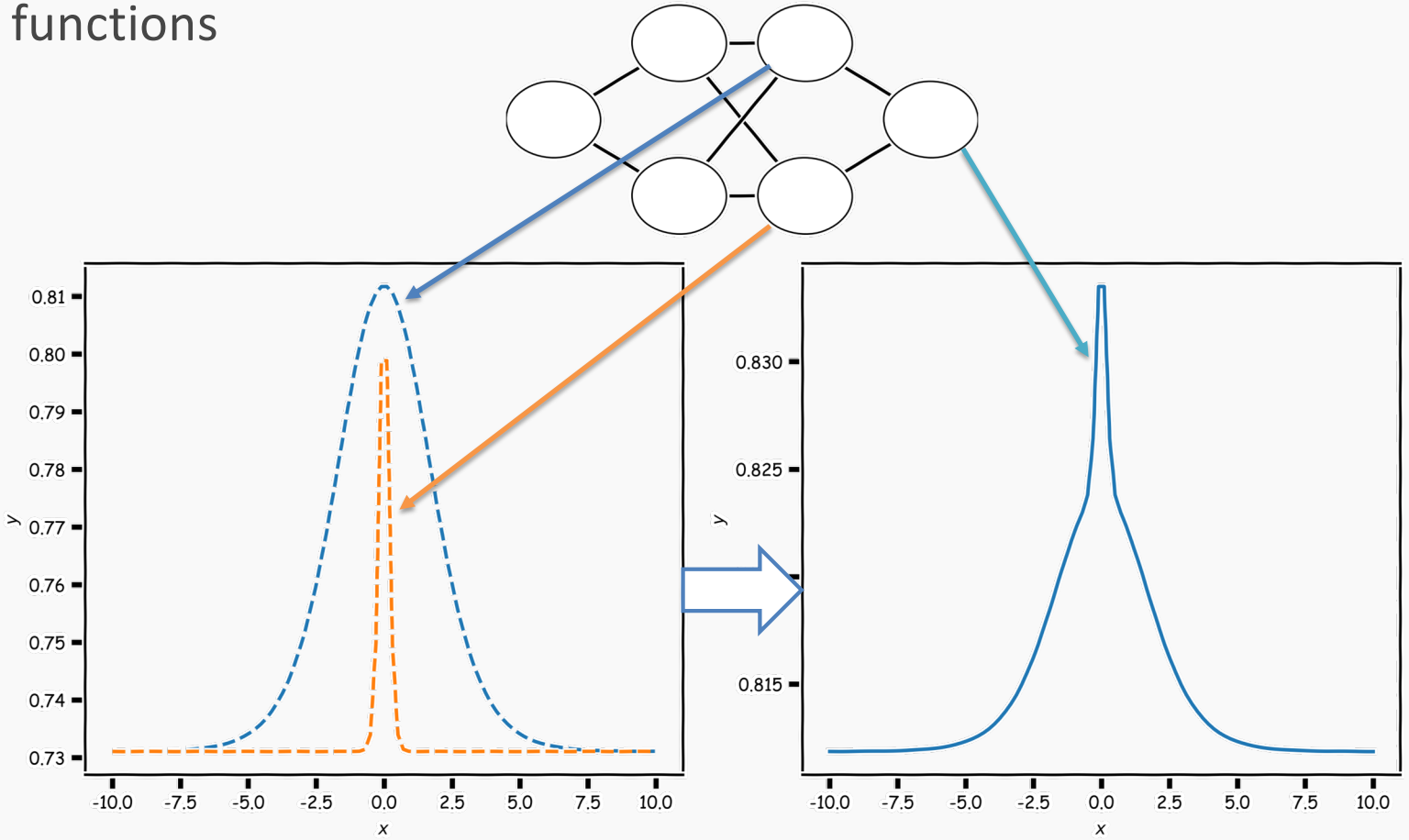
Combining neurons allows us to model interesting functions



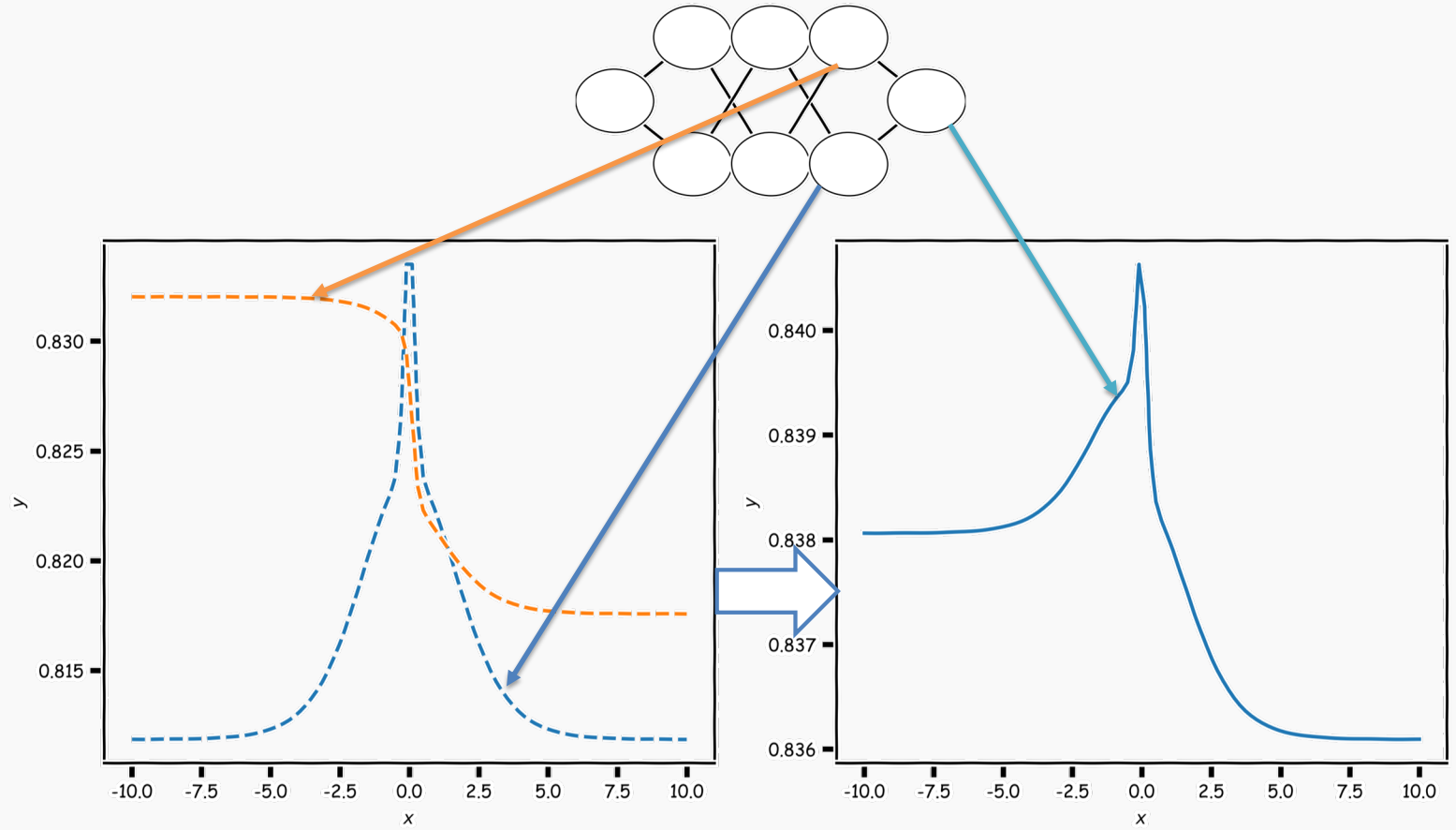
Different ‘weights’ change the shape and position



Adding layers allows us to model increasingly complex functions



Neural networks can model *any* reasonable function



Summary (so far)

So far:

- A single neuron can be a logistic regression unit. We will soon see other choices.
- A neural network is a combination of logistic regression (or other types) units.
- A neural network can approximate non-linear functions.

Next:

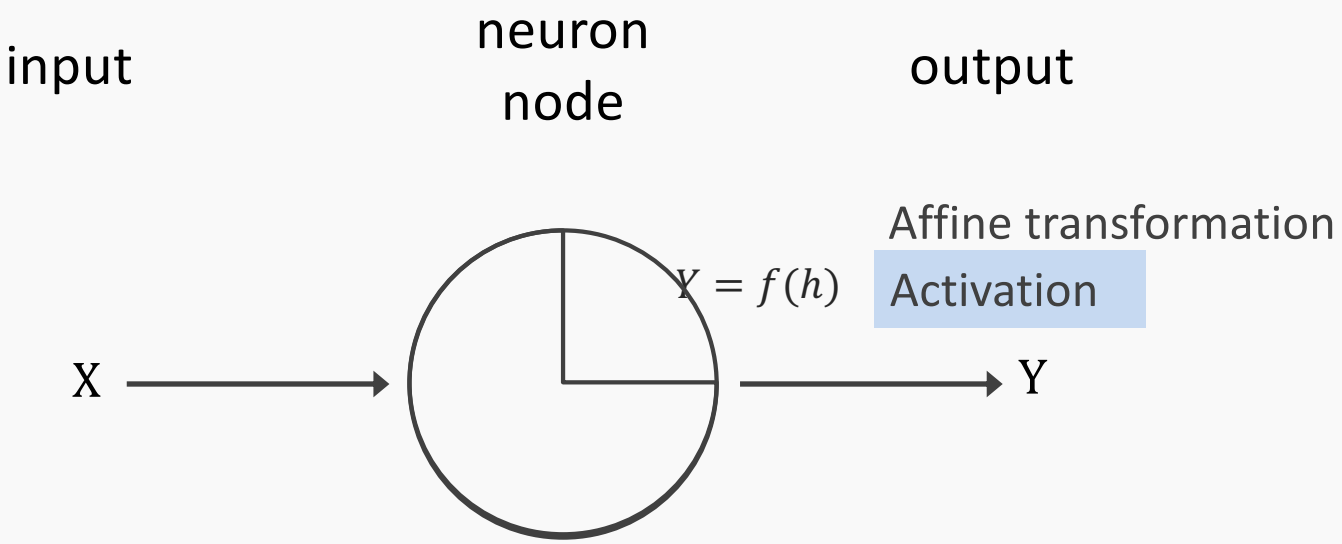
- How many neurons?
- What ”other choices”?

And beyond:

- How do we estimate weights and biases?
- Regularize Neural Networks.

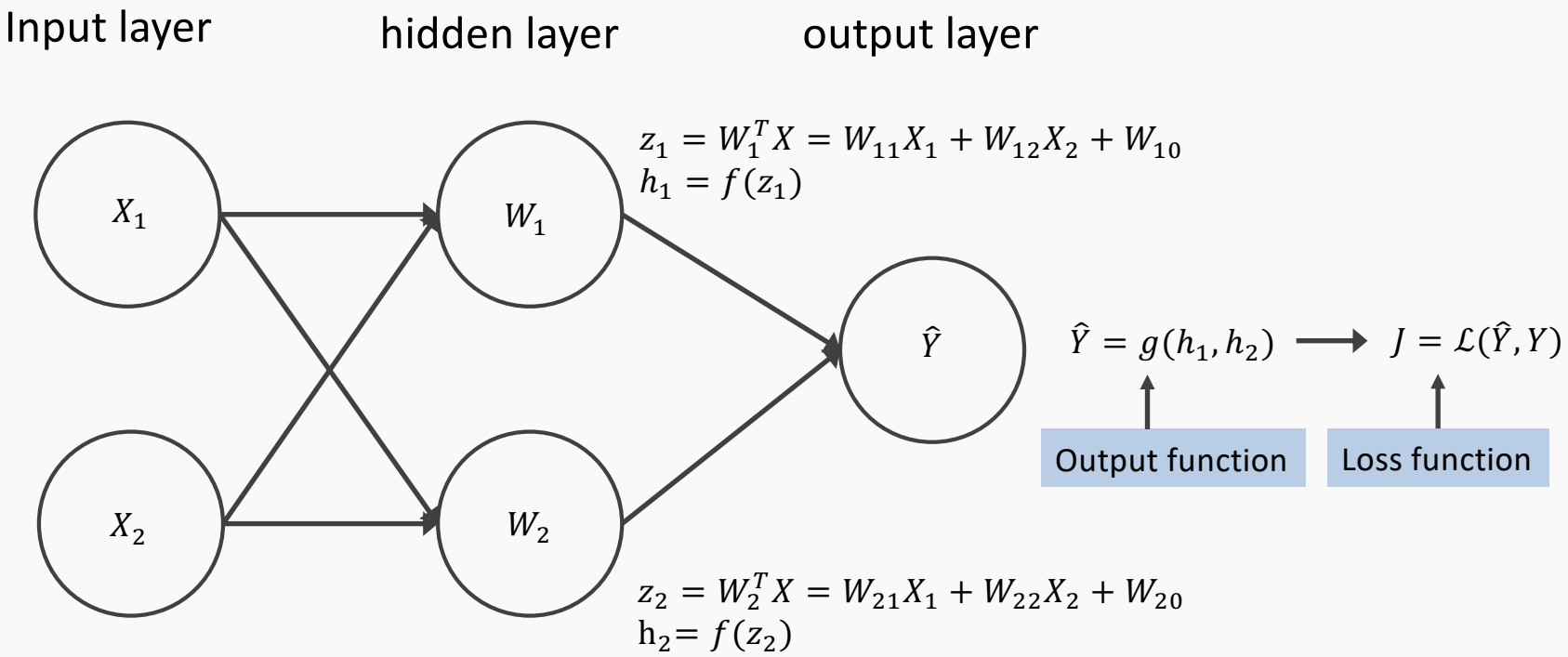
Anatomy of a NN

Anatomy of artificial neural network (ANN)



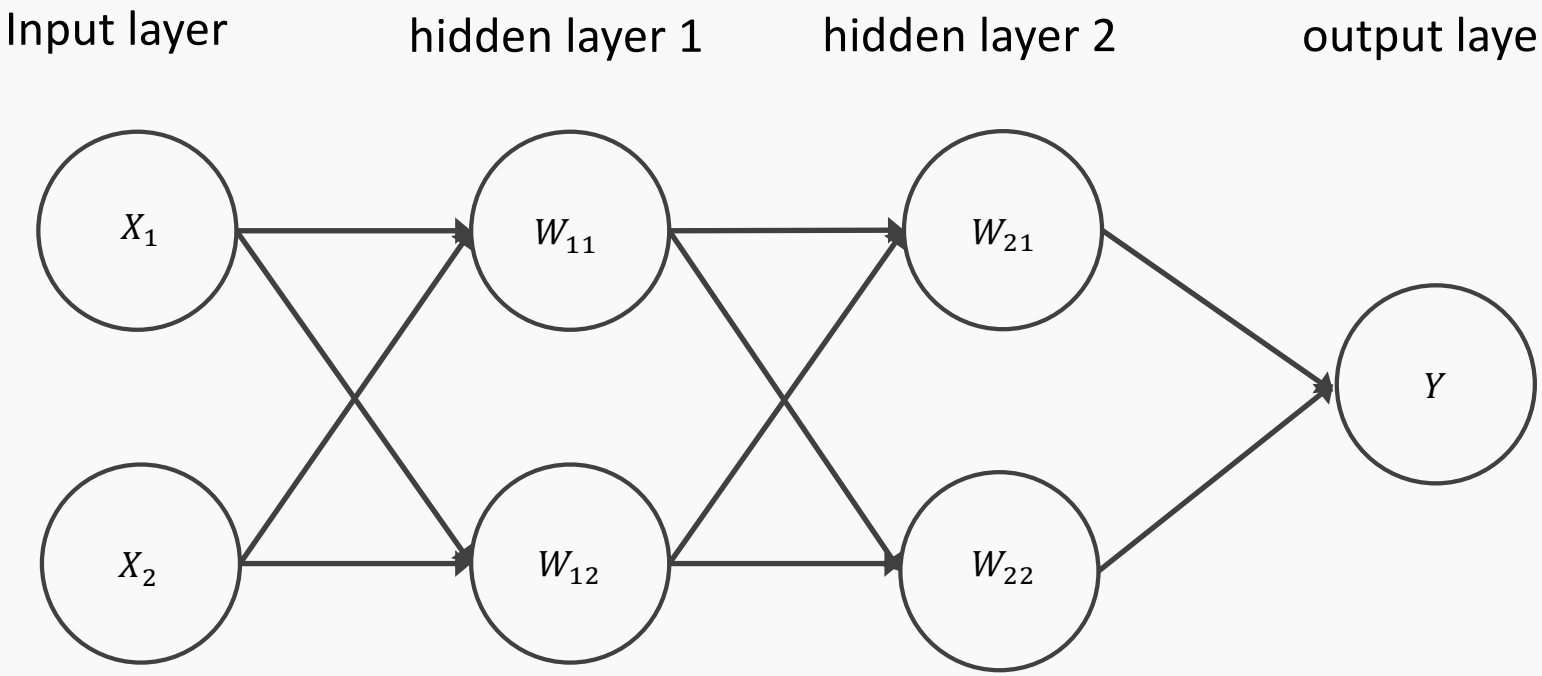
We will talk later about the choice of activation function. So far we have only talked about sigmoid as an activation function but there are other choices.

Anatomy of artificial neural network (ANN)

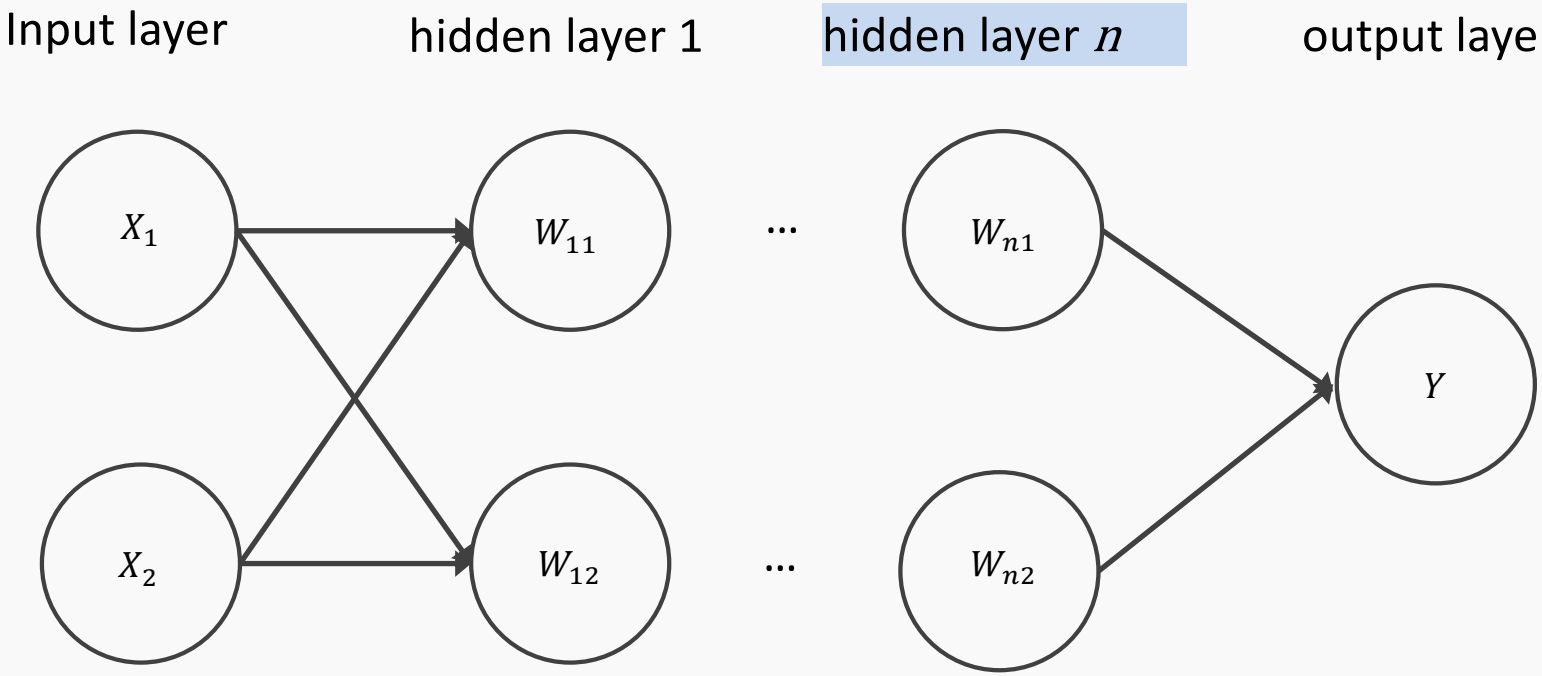


We will talk later about the choice of the output layer and the loss function. So far we consider sigmoid as the output and log-bernouli.

Anatomy of artificial neural network (ANN)



Anatomy of artificial neural network (ANN)



We will talk later about the choice of the number of layers.

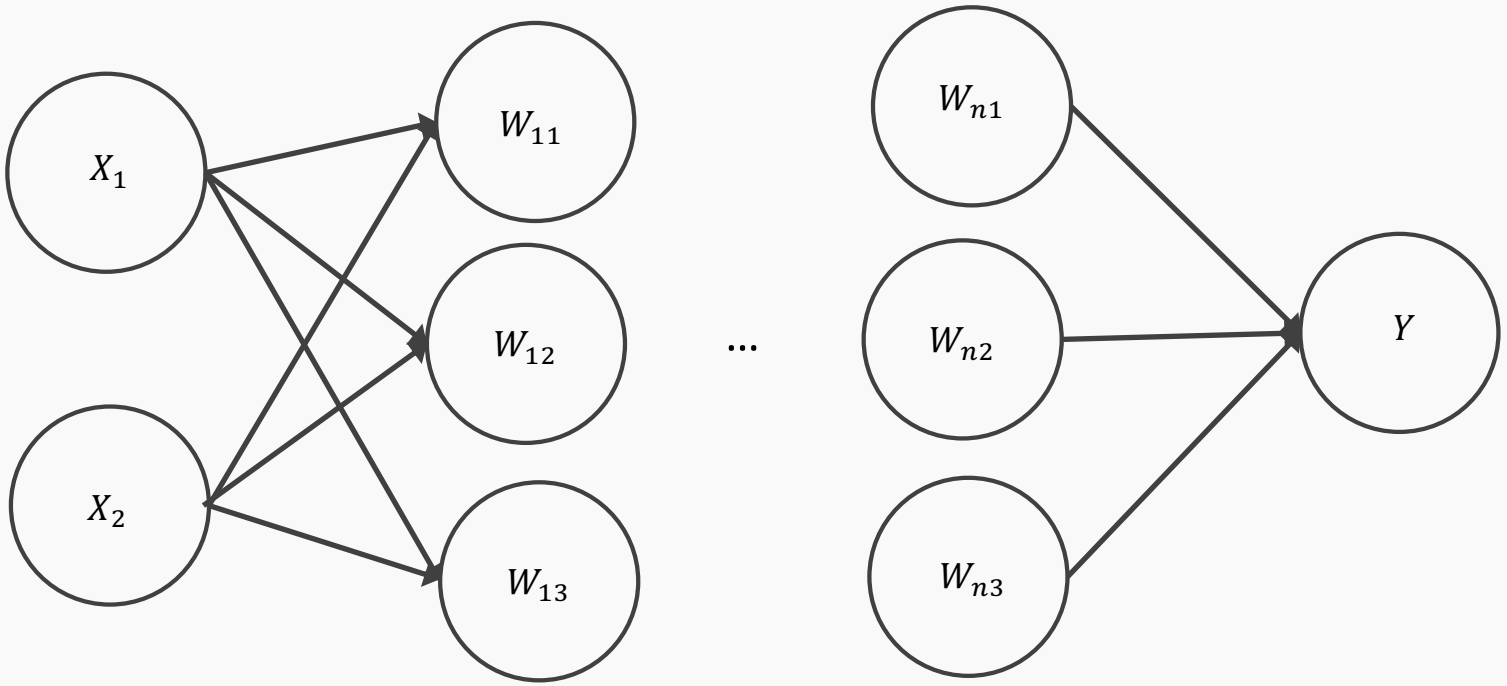
Anatomy of artificial neural network (ANN)

Input layer

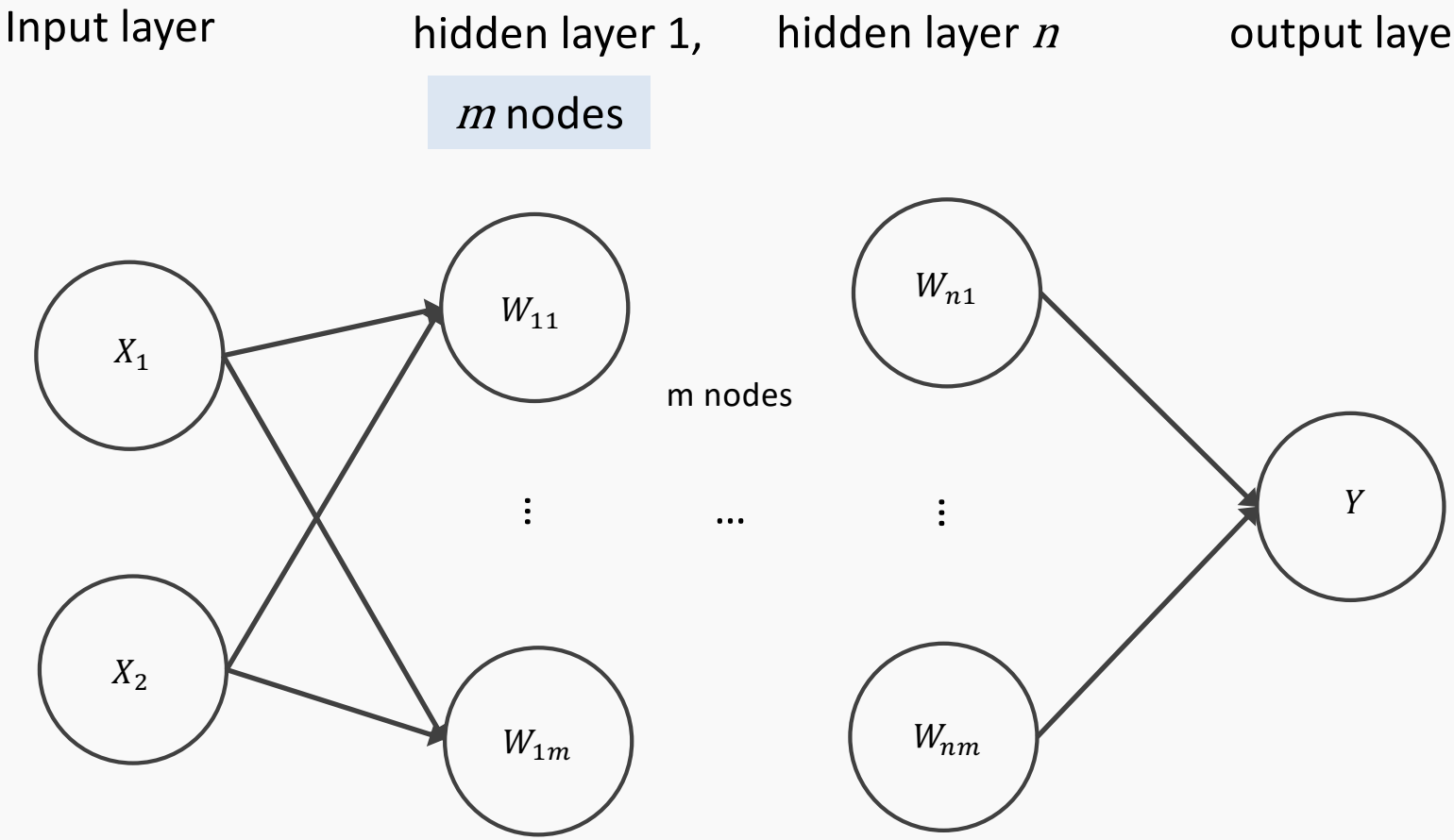
hidden layer 1, 3 nodes

hidden layer n *3 nodes*

output layer

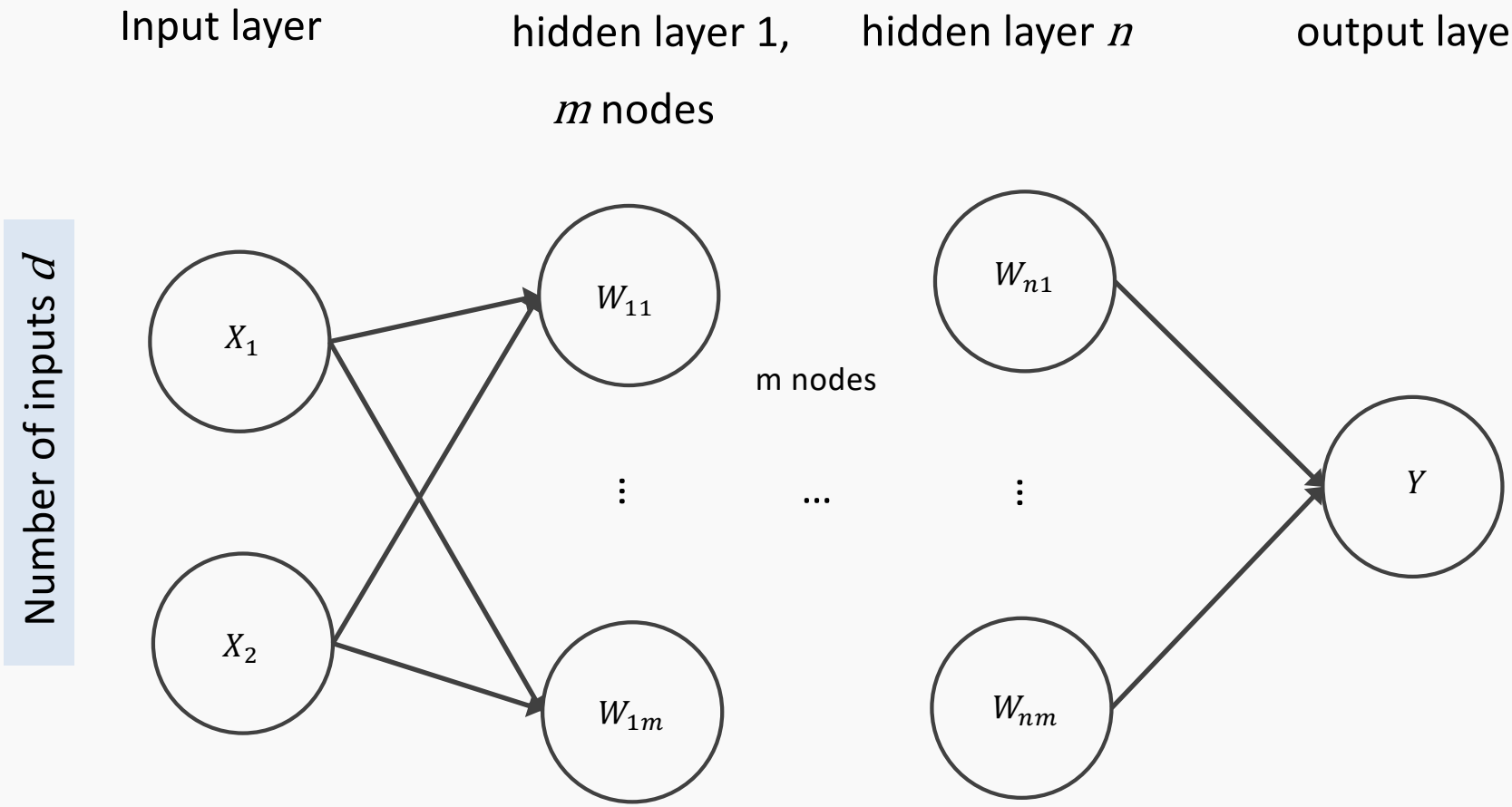


Anatomy of artificial neural network (ANN)



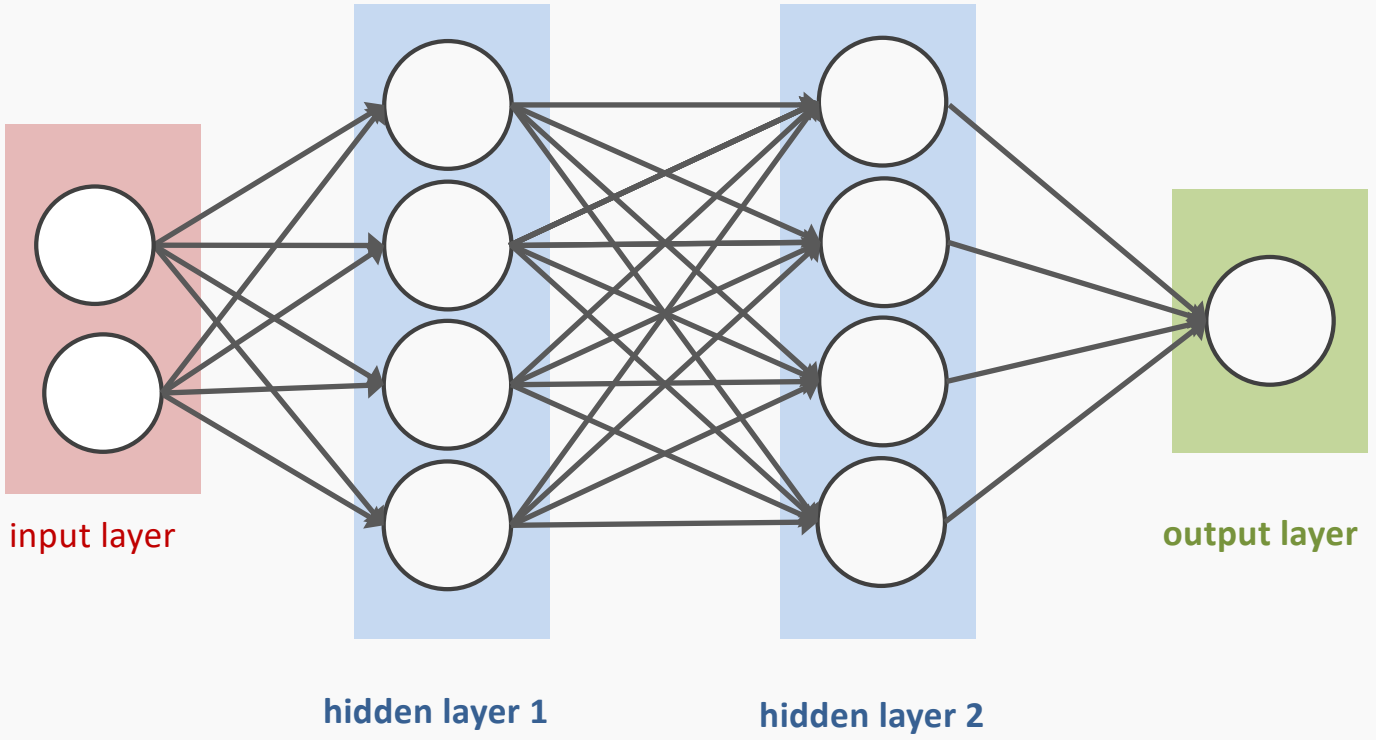
We will talk later about the choice of the number of nodes.

Anatomy of artificial neural network (ANN)

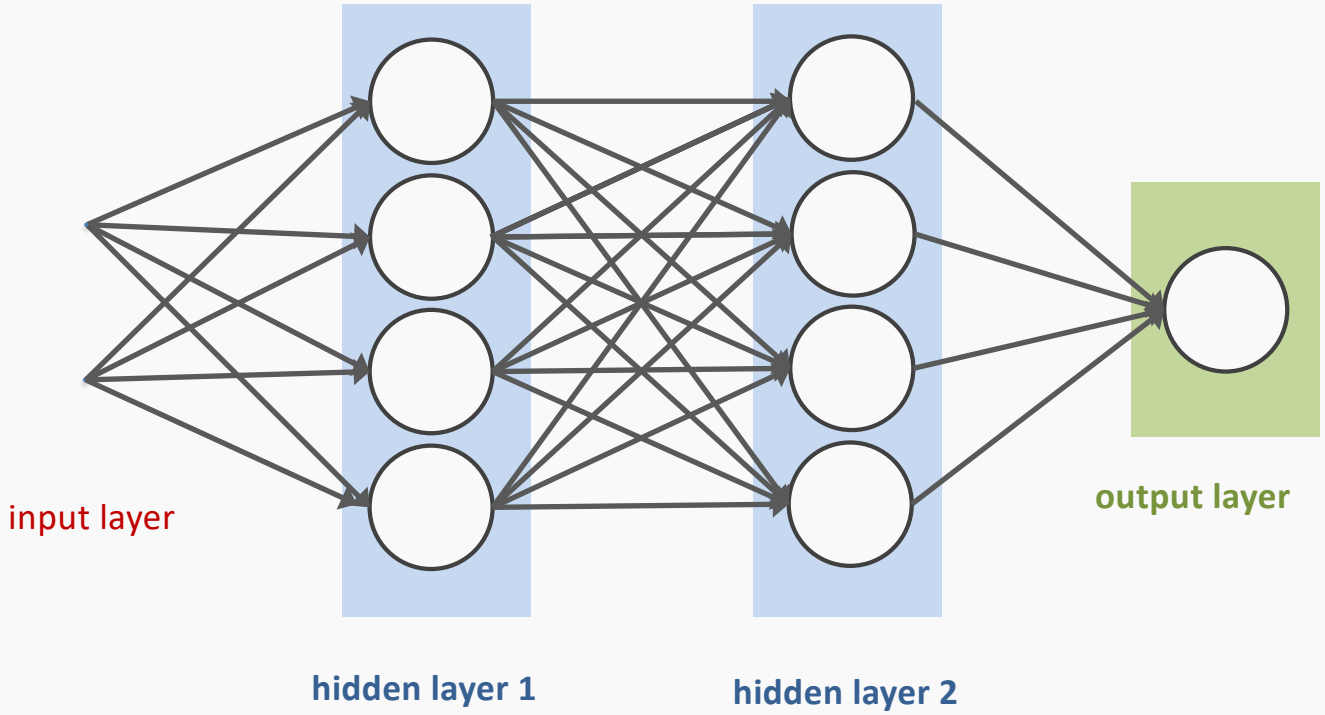


Number of inputs is specified by the data

Anatomy of artificial neural network (ANN)

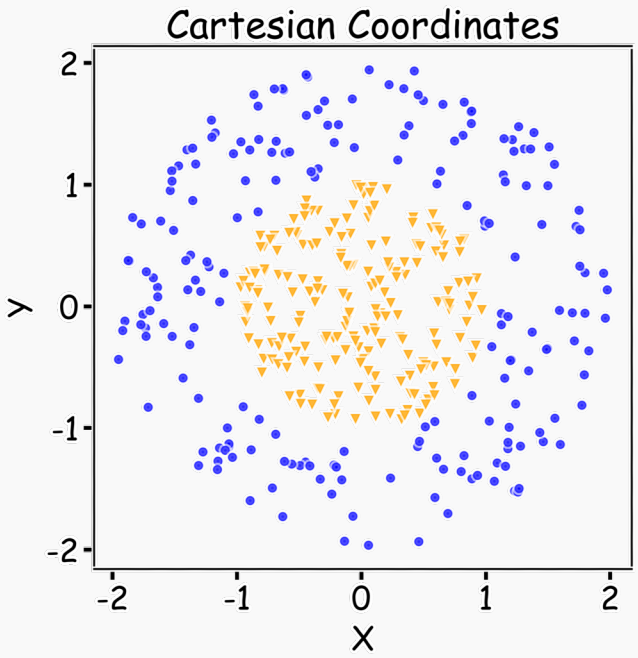


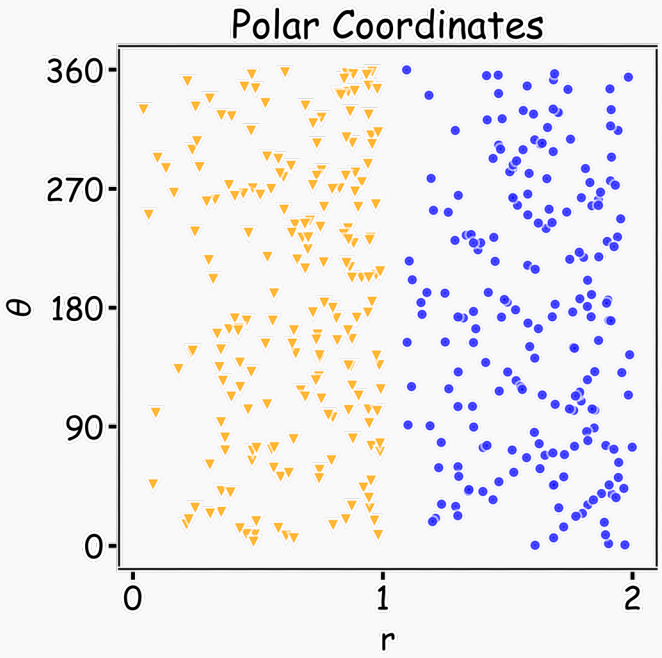
Anatomy of artificial neural network (ANN)



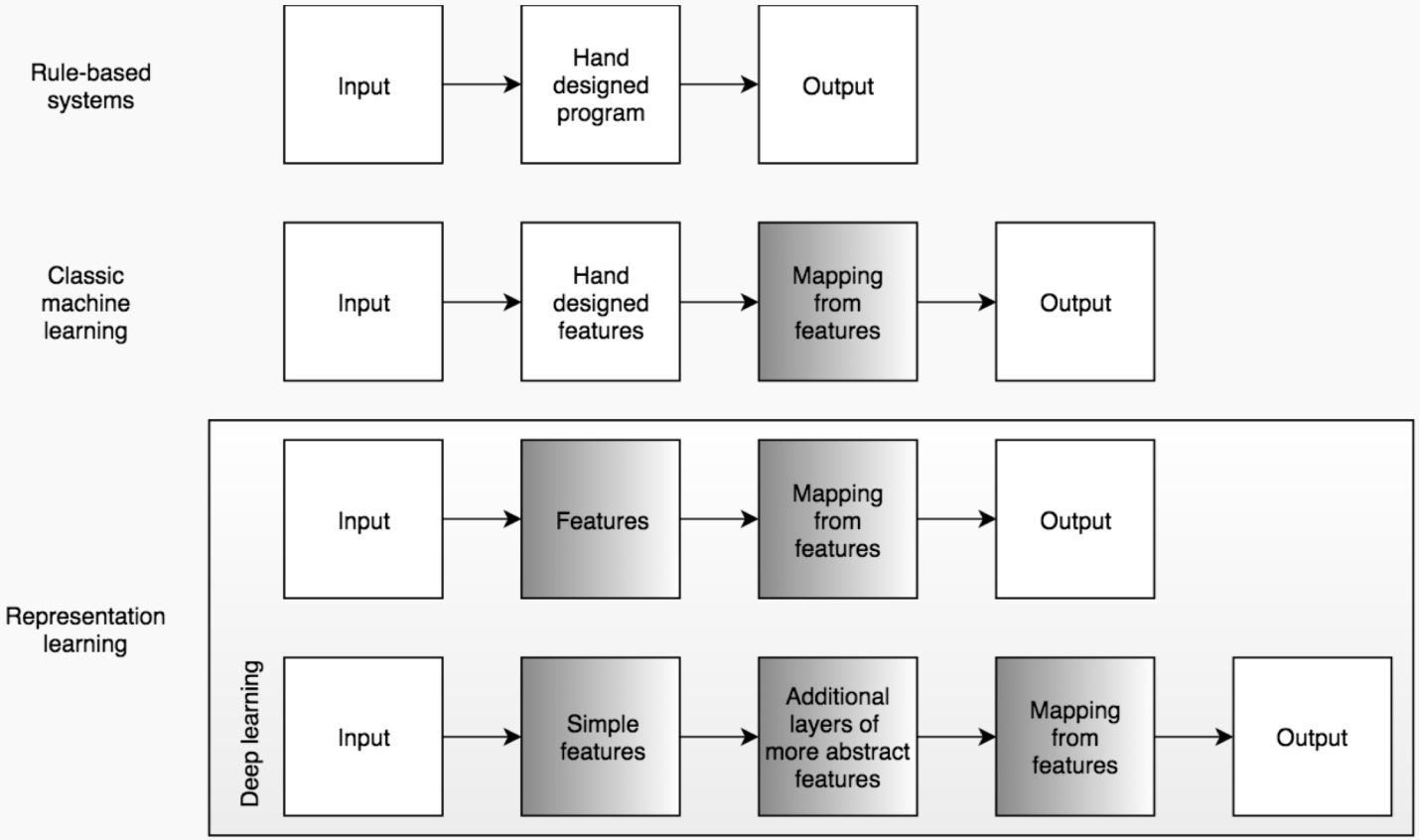
Why layers? Representation

Representation matters!





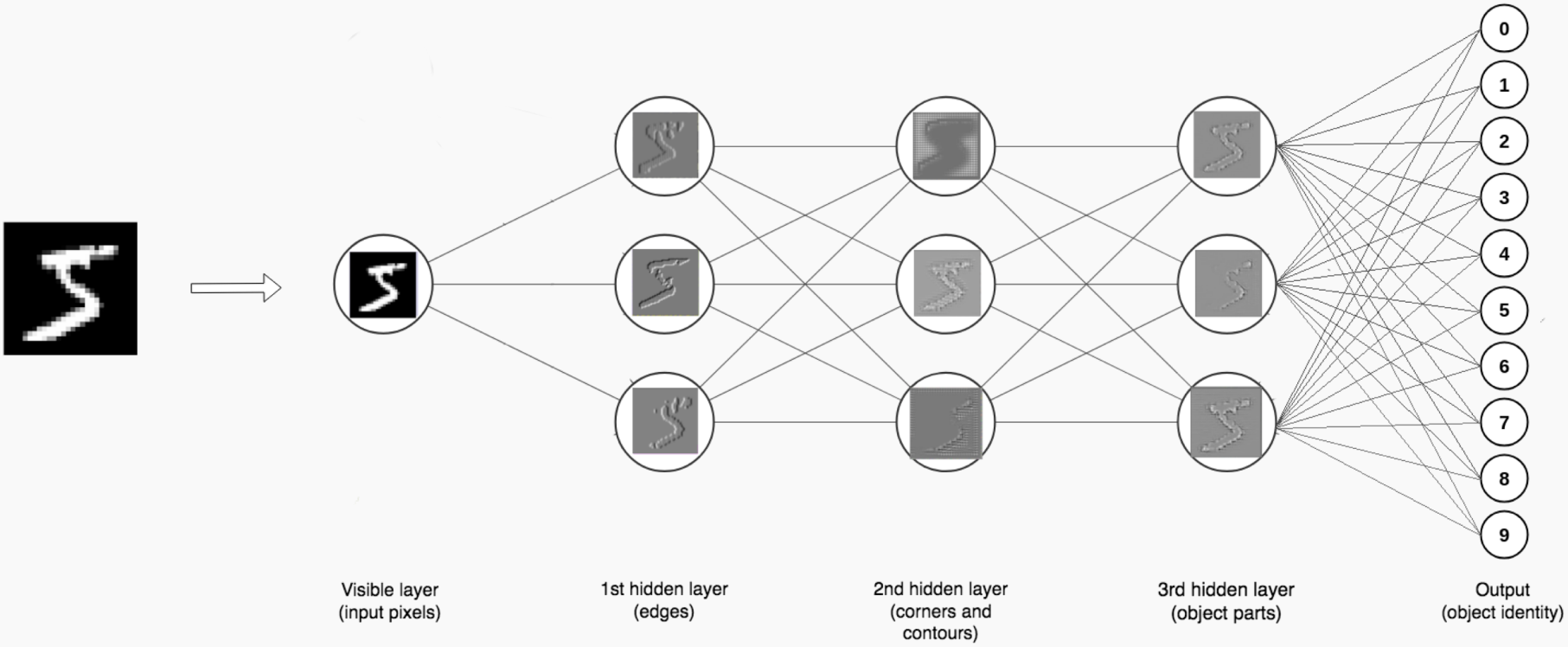
Learning Multiple Components



Neural Networks

Hand-written digit recognition: MNIST data

Depth = Repeated Compositions



Beyond Linear Models

Linear models:

- Can be fit efficiently (via convex optimization)
- Limited model capacity

Alternative:

$$f(x) = w^T \phi(x)$$

Where ϕ is a *non-linear transform*

Traditional ML

Manually engineer ϕ

- Domain specific, enormous human effort

Generic transform

- Maps to a higher-dimensional space
- Kernel methods: e.g. RBF kernels
- Over fitting: does not generalize well to test set
- Cannot encode enough prior information

Deep Learning

- Directly learn ϕ

$$f(x;\theta) = W^T \phi(x;\theta)$$

- $\phi(x;\theta)$ is an automatically-learned **representation** of x
- For **deep networks**, ϕ is the function learned by the **hidden layers** of the network
- θ are the learned weights

- Non-convex optimization
- Can encode prior beliefs, generalizes well

Design Choices: Activation Function

Activation function

$$h = f(W^T X + b)$$

The activation function should:

- Provide non-linearity
- Ensure gradients remain large through hidden unit

Common choices are

- Sigmoid
- Relu, leaky ReLU, Generalized ReLU, MaxOut
- softplus
- tanh
- swish

Activation function

$$h = f(W^T X + b)$$

The activation function should:

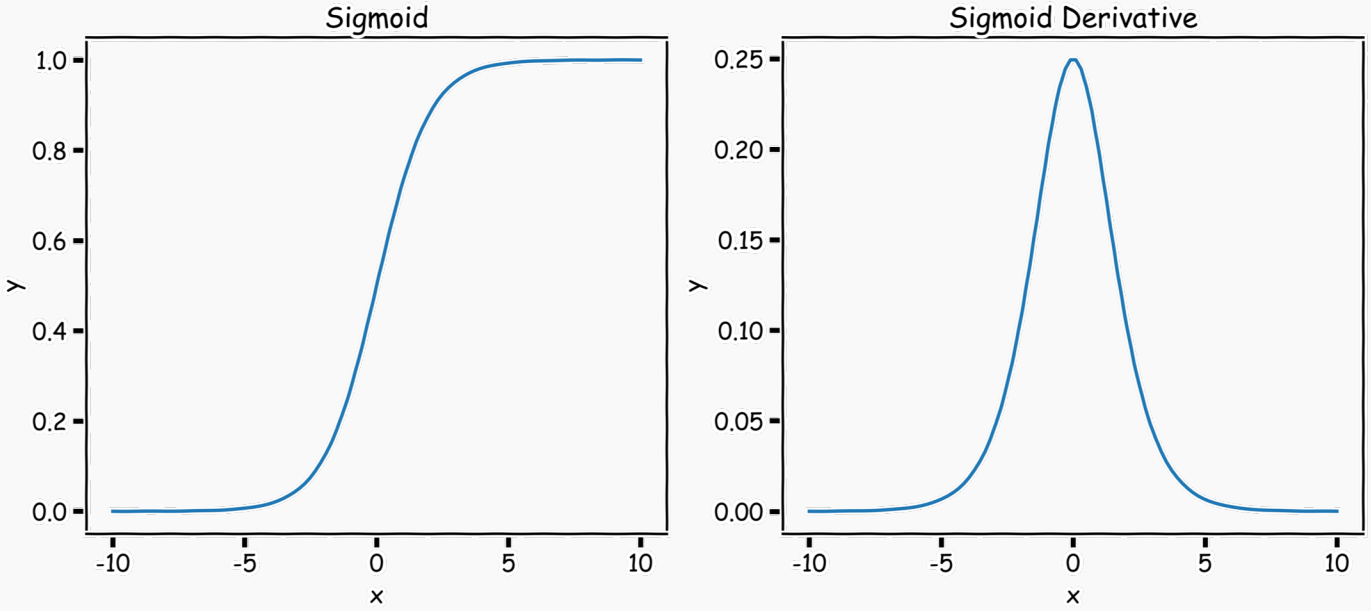
- Provide **non-linearity**
- Ensure gradients remain large through hidden unit

Common choices are

- sigmoid
- tanh
- ReLU, leaky ReLU, Generalized ReLU, MaxOut
- softplus
- swish

Sigmoid (aka Logistic)

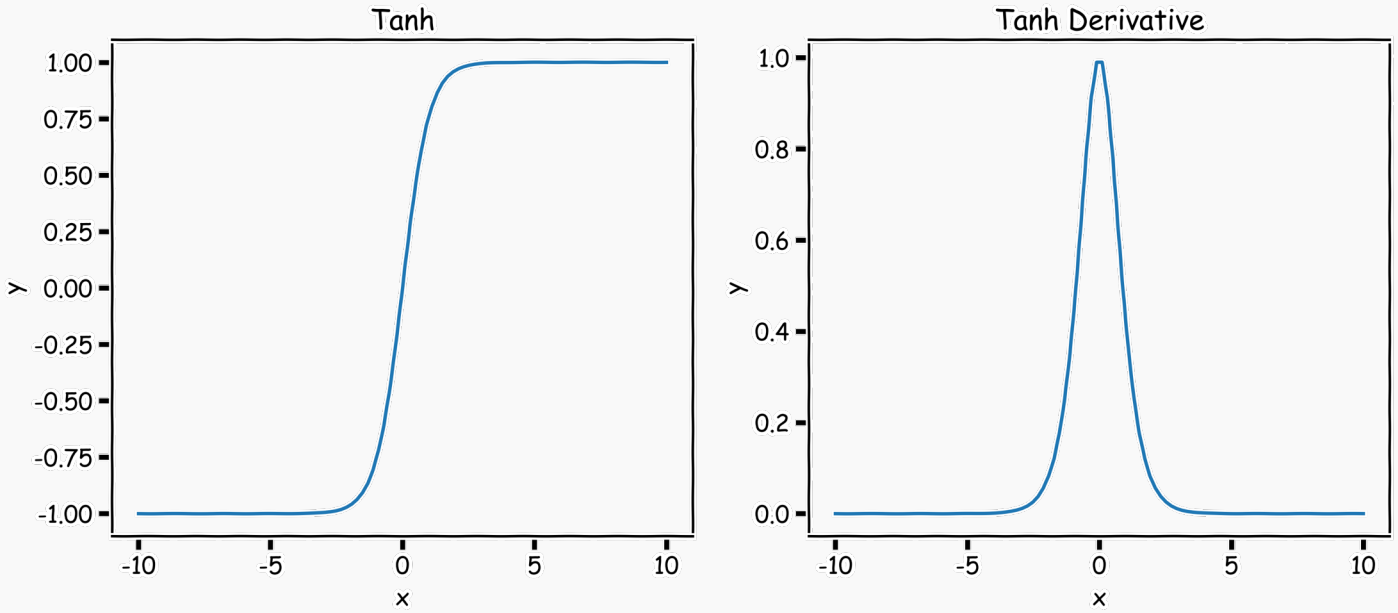
$$y = \frac{1}{1 + e^{-x}}$$



Derivative is **zero** for much of the domain. This leads to "vanishing gradients" in backpropagation.

Hyperbolic Tangent (Tanh)

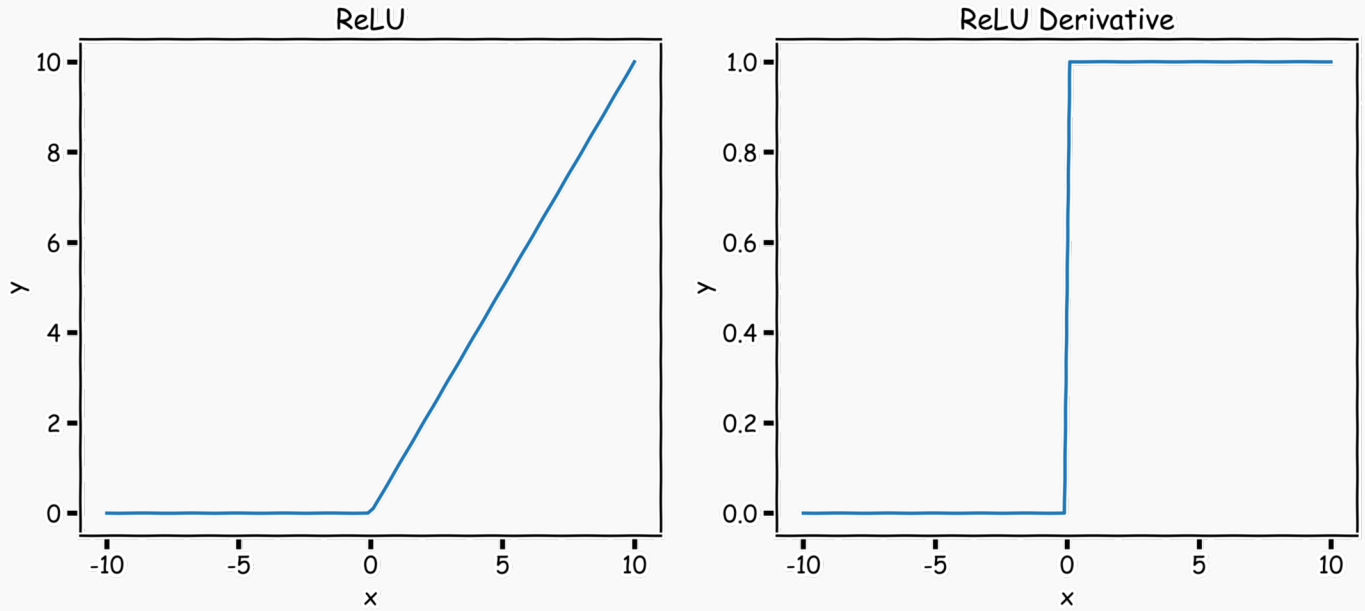
$$y = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Same problem of "vanishing gradients" as sigmoid.

Rectified Linear Unit (ReLU)

$$y = \max(0, x)$$

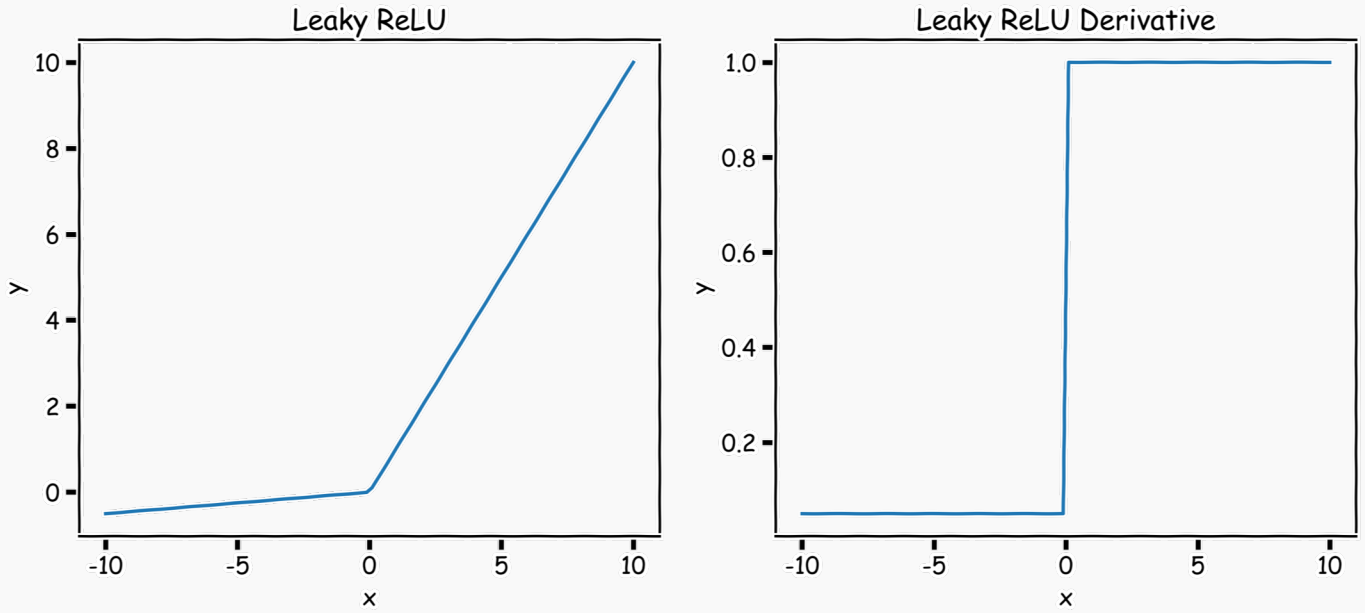


Two major advantages:

1. No vanishing gradient when x > 0
2. Provides sparsity (regularization) since y = 0 when x < 0

Leaky ReLU

$$y = \max(0, x) + \alpha \min(0,1)$$
where α takes a small value

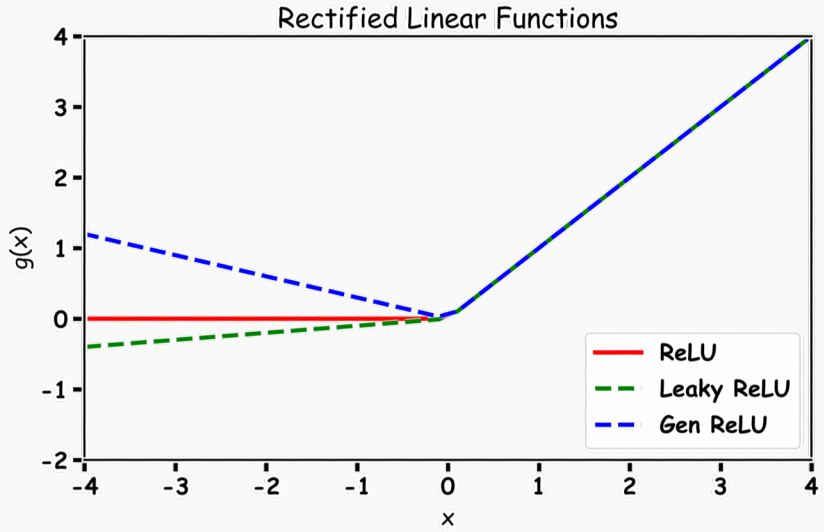


- Tries to fix "dying ReLU" problem: derivative is non-zero everywhere.
- Some people report success with this form of activation function, but the results are not always consistent

Generalized ReLU

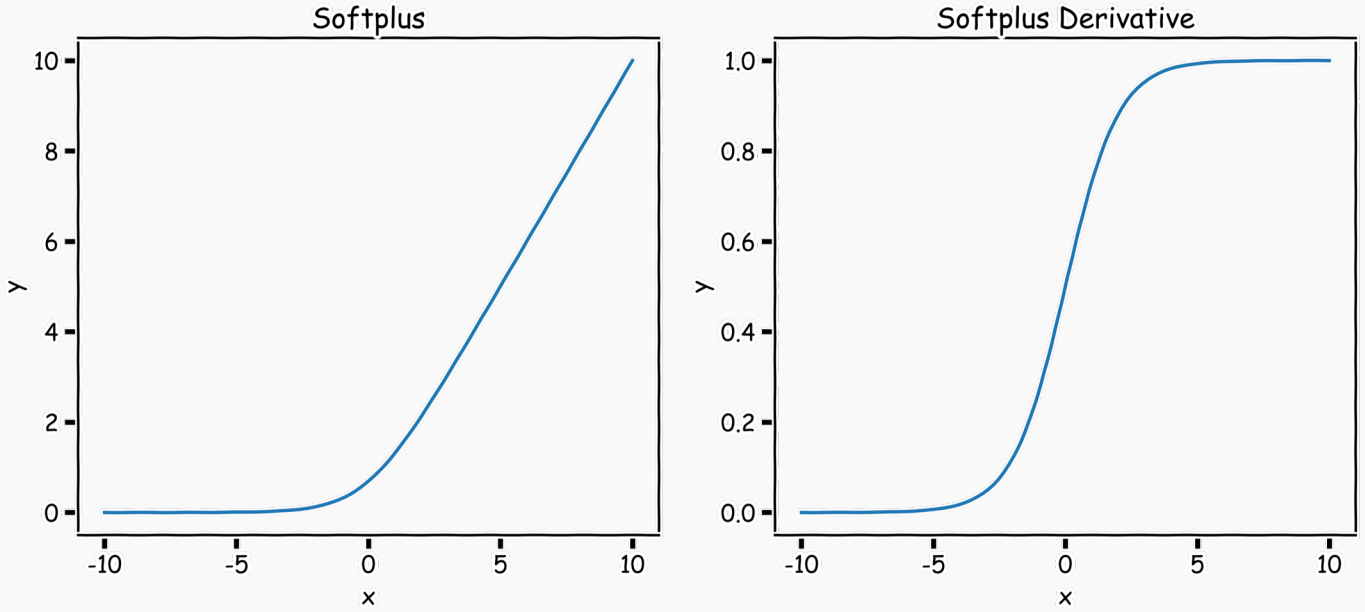
Generalization: For $\alpha_i > 0$

$$g(x_i, \alpha) = \max\{a, x_i\} + \alpha \min\{0, x_i\}$$



softplus

$$y = \log(1 + e^x)$$

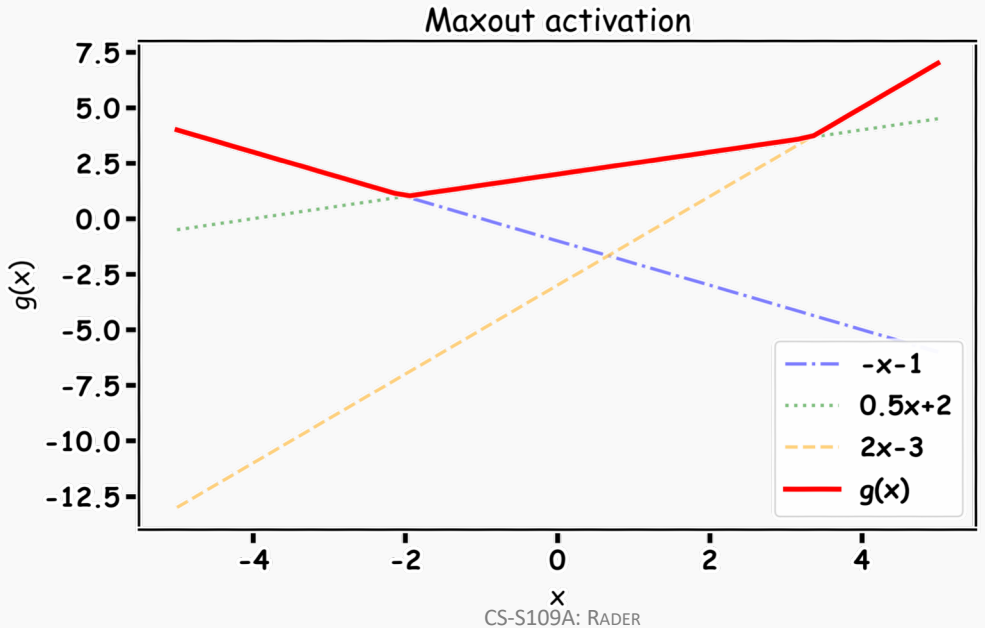


The logistic sigmoid function is a smooth approximation of the derivative of the rectifier

Maxout

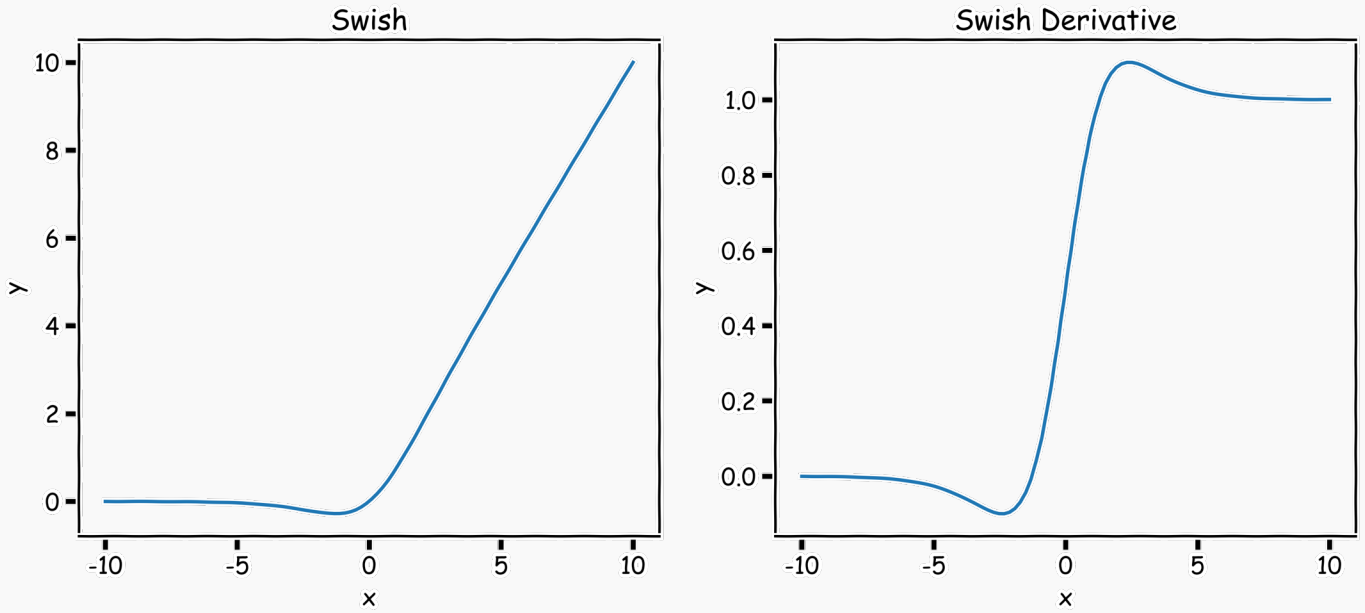
Max of k linear functions. Directly learn the activation function.

$$g(x) = \max_{i \in \{1,\ldots,k\}} \alpha_i x_i + \beta$$



Swish: A Self-Gated Activation Function

$$g(x) = x\,\sigma(x)$$



Currently, the most successful and widely-used activation function is the ReLU. Swish tends to work better than ReLU on deeper models across a number of challenging datasets.

Design Choices: Loss Function

Loss Function

Likelihood for a given point:

$$p(y_i|W;x_i)$$

Assume independency, likelihood for all measurements:

$$L(W;X,Y) = p(Y|W;X) = \prod_i p(y_i|W;x_i)$$

Maximize the likelihood, or equivalently maximize the log-likelihood:

$$\log L(W;X,Y) = \sum_i \log p(y_i|W;x_i)$$

Turn this into a loss function:

$$\mathcal{L}(W;X,Y) = -\log L(W;X,Y)$$

Loss Function

Do not need to design separate loss functions if we follow this simple procedure

Examples:

- Distribution is **Normal** then likelihood is:

$$p(y_i|W;x_i) = \frac{1}{\sqrt{\{2\pi^2\sigma\}}} e^{-\frac{(y_i-\hat{y}_i)^2}{2\sigma\hat{}2}}$$

MSE

$$\mathcal{L}(W;X,Y) = \sum_i (y_i - \hat{y}_i)^2$$

- Distribution is **Bernouli** then likelihood is:

$$p(y_i|W;x_i) = p_i^{y_i}(1-p_i)^{1-y_i}$$

$$\mathcal{L}(W;X,Y) = -\sum_i [y_i \log p_i + (1-y_i)\log(1-p_i)]$$

Cross-Entropy

Design Choices: Output Units

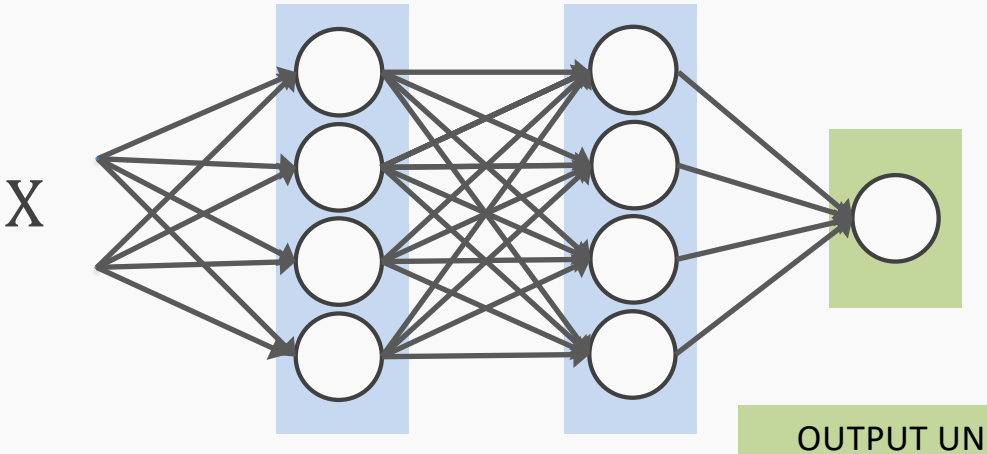
Output Units

Output Type	Output Distribution	Output layer	Loss Function
Binary			

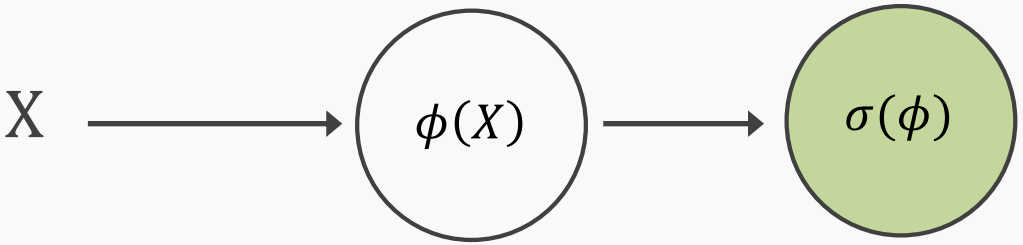
Output Units

Output Type	Output Distribution	Output layer	Loss Function
Binary	Bernoulli	?	Binary Cross Entropy

Output unit for binary classification



$\hat{Y} = \mathrm{P}(\mathrm{y} = 1)$



$\hat{Y} = \mathrm{P}(\mathrm{y} = 1)$

$$X \Longrightarrow \phi(X) \Longrightarrow P(y = 1) = \frac{1}{1 + e^{-\phi(X)}}$$

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy

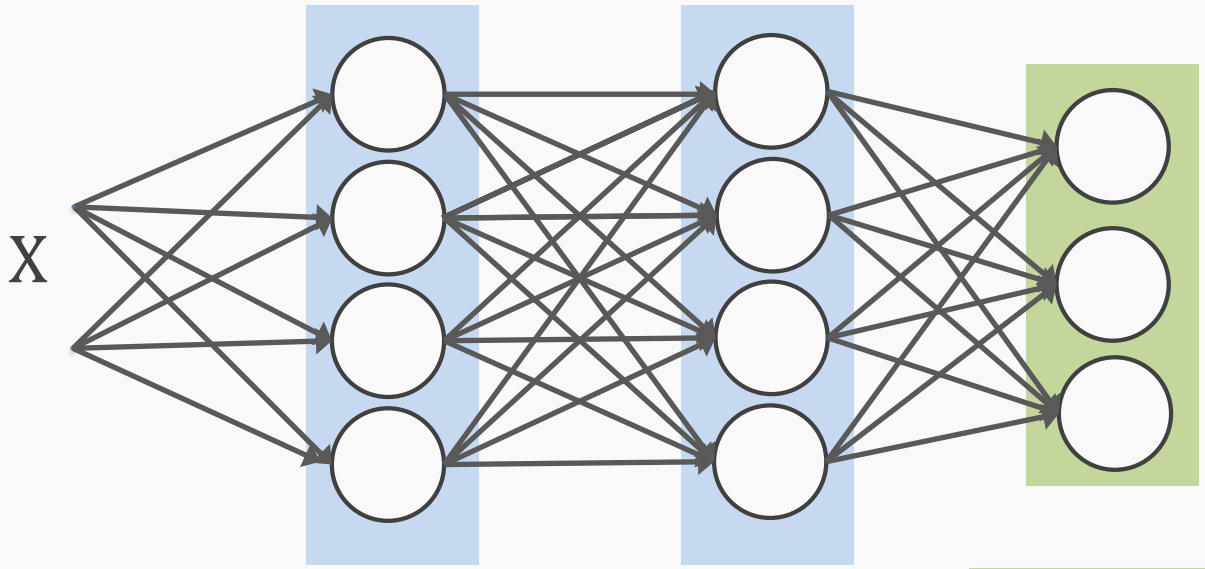
Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete			

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinouli	?	Cross Entropy

Output unit for multi-class classification

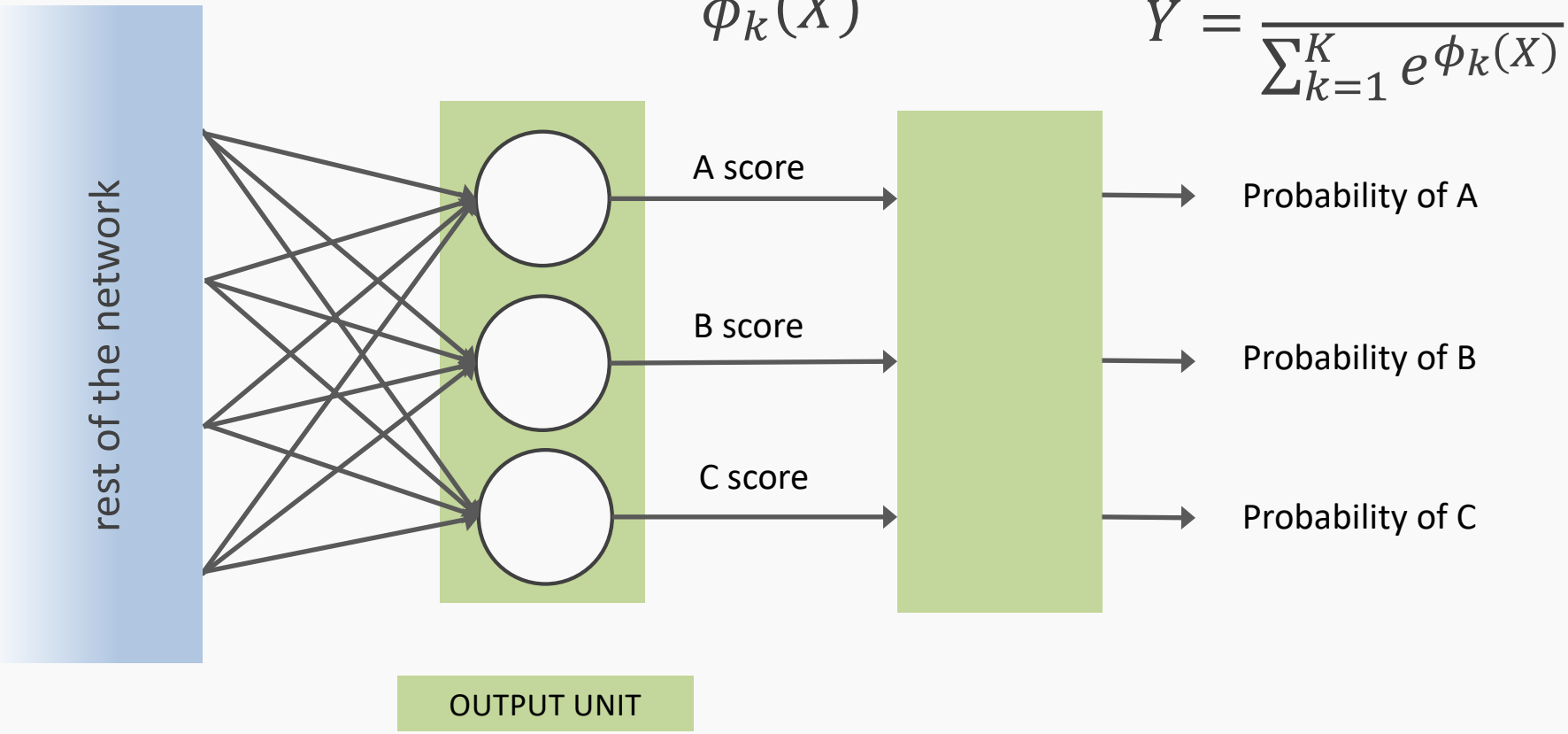


$$\hat{Y} = [P_1, P_2, P_3]$$

SoftMax

$$\phi_k(X)$$

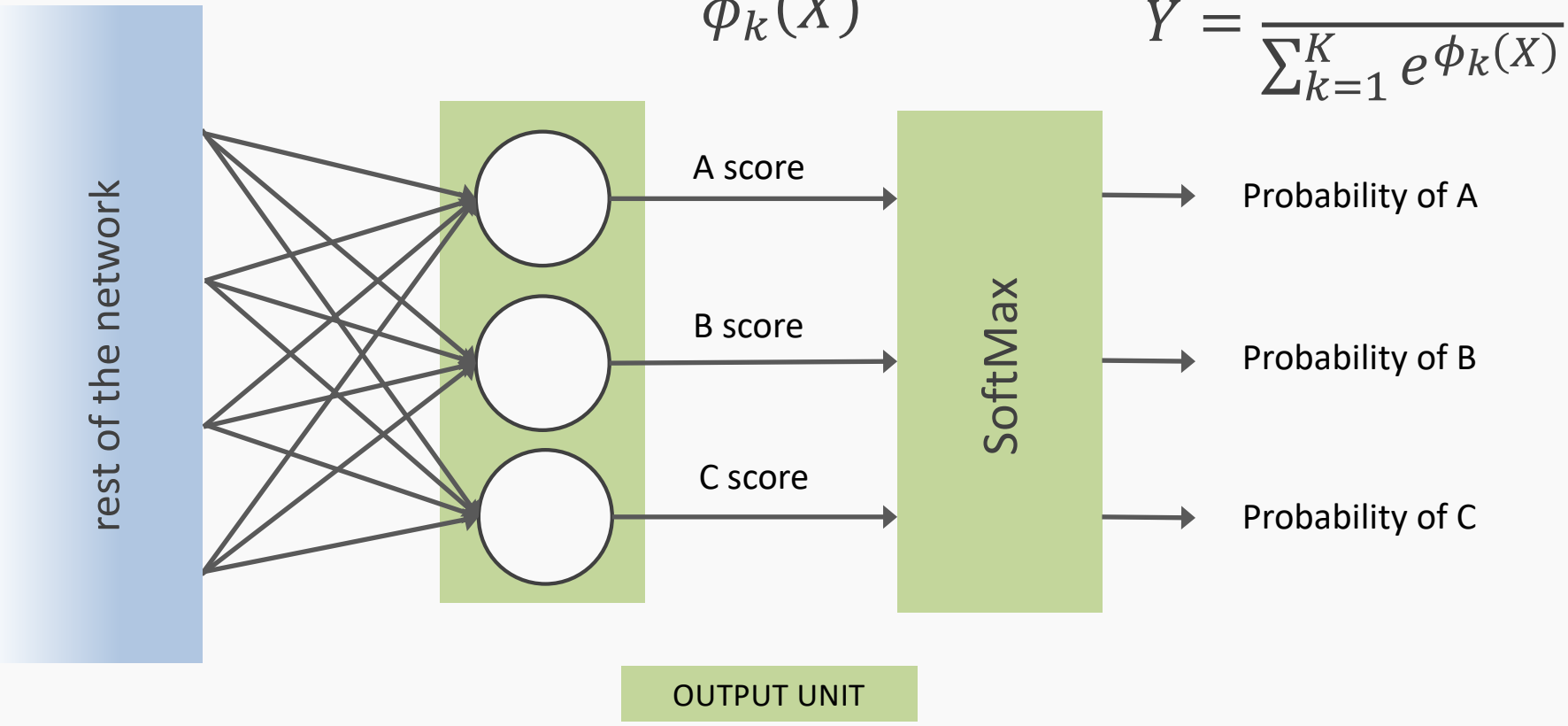
$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$$



SoftMax

$$\phi_k(X)$$

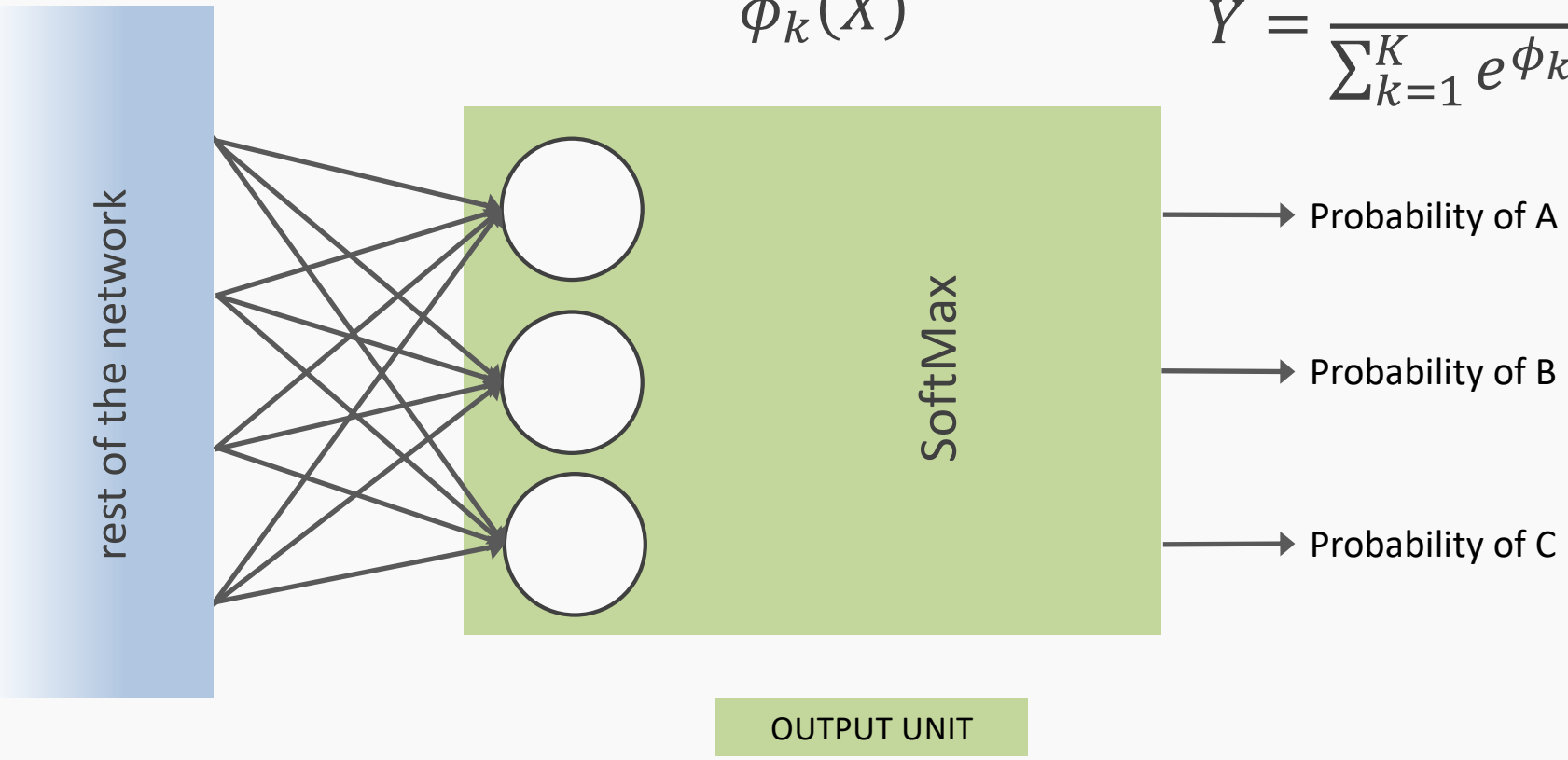
$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$$



SoftMax

$$\phi_k(X)$$

$$\hat{Y} = \frac{e^{\phi_k(X)}}{\sum_{k=1}^{K} e^{\phi_k(X)}}$$



Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous			

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian		

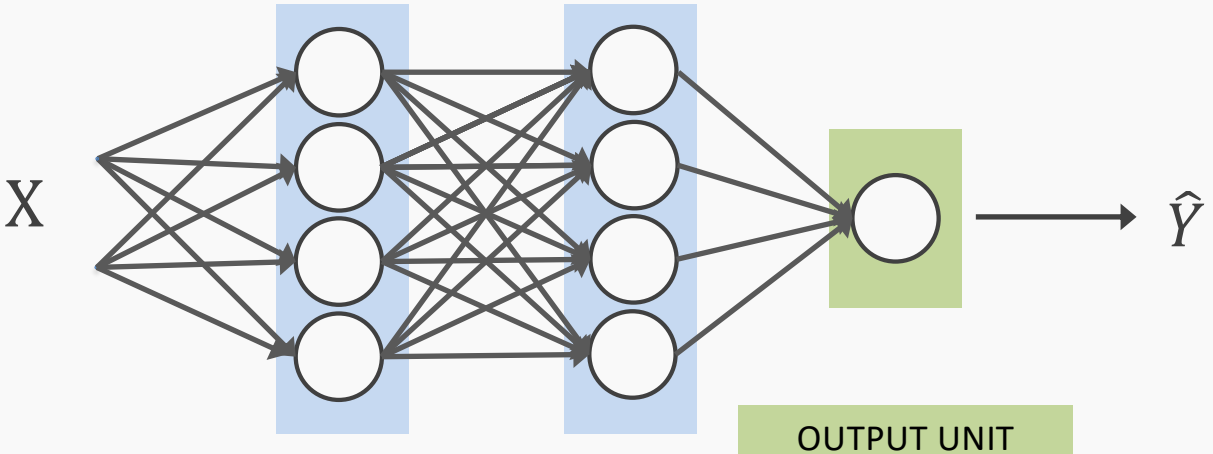
Output Units

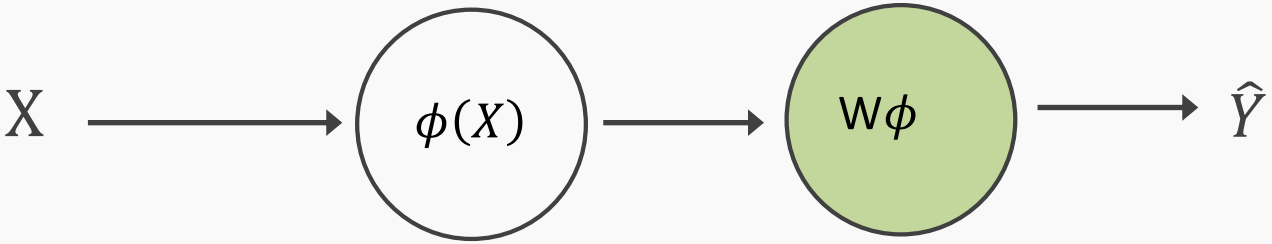
Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian		MSE

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	?	MSE

Output unit for regression





$$X \Longrightarrow \phi(X) \Longrightarrow \hat{Y} = W\phi(X)$$

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE

Output Units

Output Type	Output Distribution	Output layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	

Output Units

Output Type	Output Distribution	Output layer	Loss Function
Binary	Bernoulli	Sigmoid	Binary Cross Entropy
Discrete	Multinoulli	Softmax	Cross Entropy
Continuous	Gaussian	Linear	MSE
Continuous	Arbitrary	-	GANS

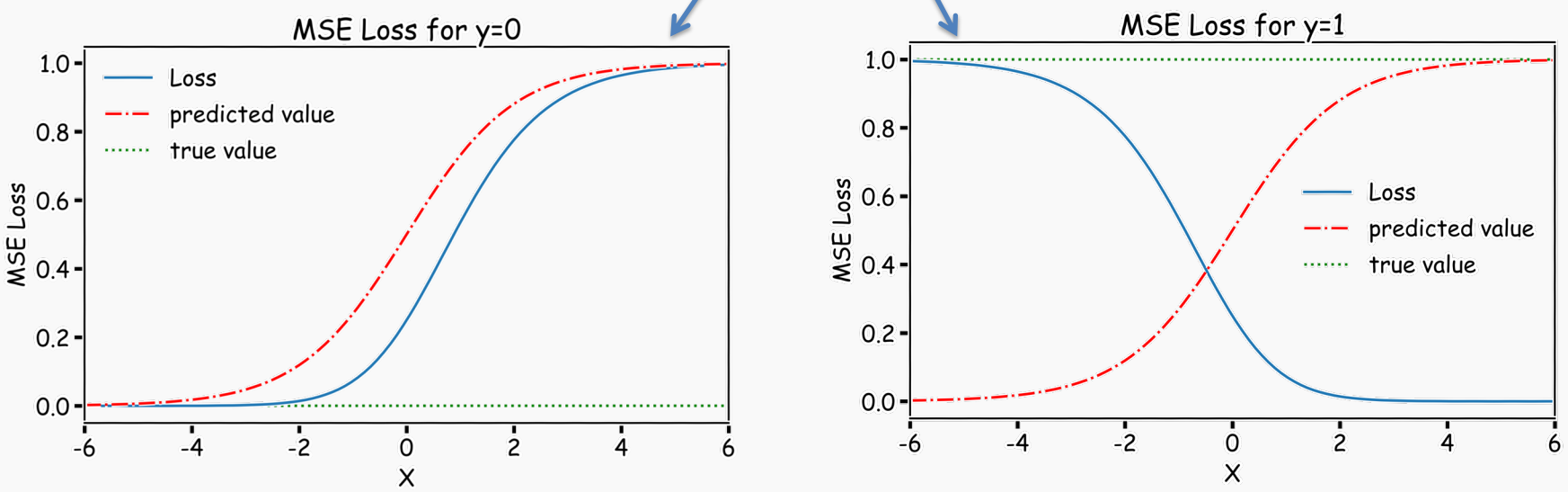
Lectures 18-19 in CS109B

Loss Function

Example: sigmoid output + squared loss

$$L_{sq} = (y - \hat{y})^2 = \big(y - \sigma(x)\big)^2$$

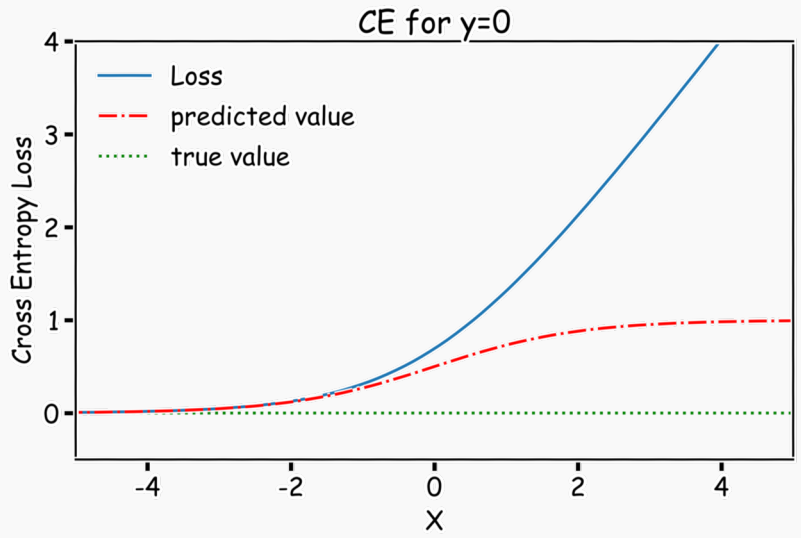
Flat surfaces

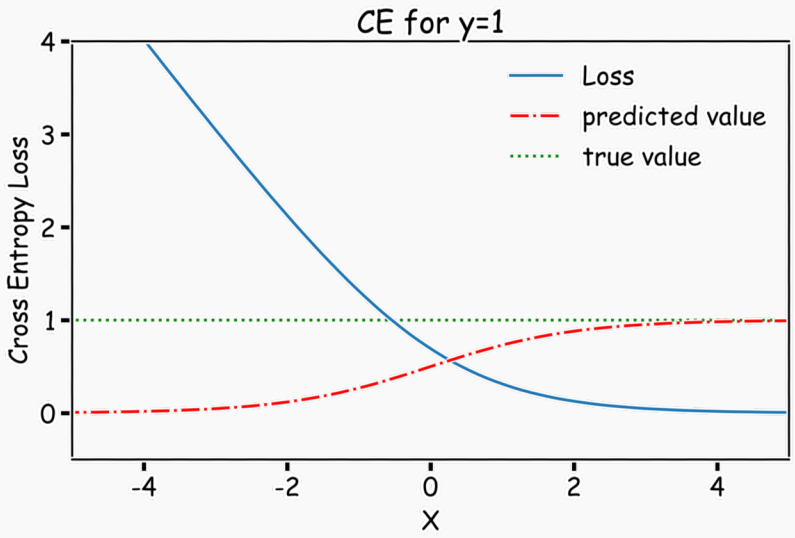


Cost Function

Example: sigmoid output + cross-entropy loss

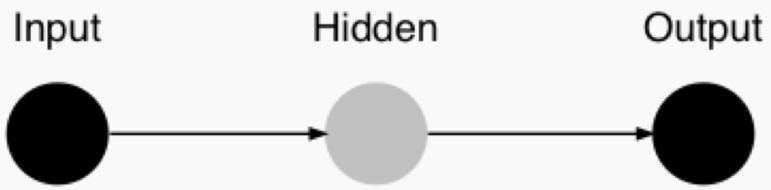
$$L_{ce}(y, \hat{y}) = -\{\, y \log \hat{y} + (1 - y) \log(1 - \hat{y})\}$$

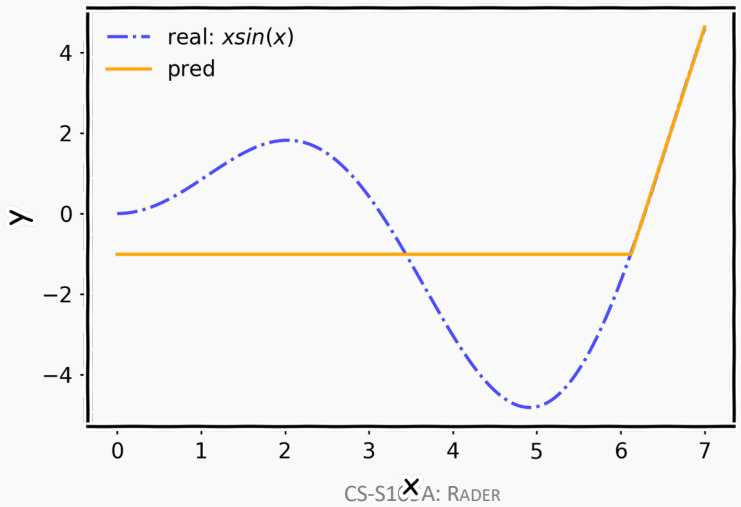




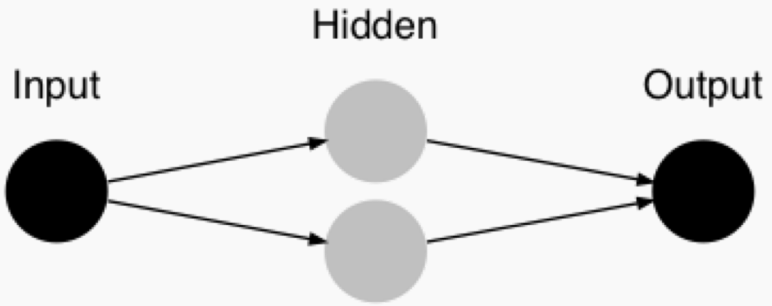
Design Choices: Architecture

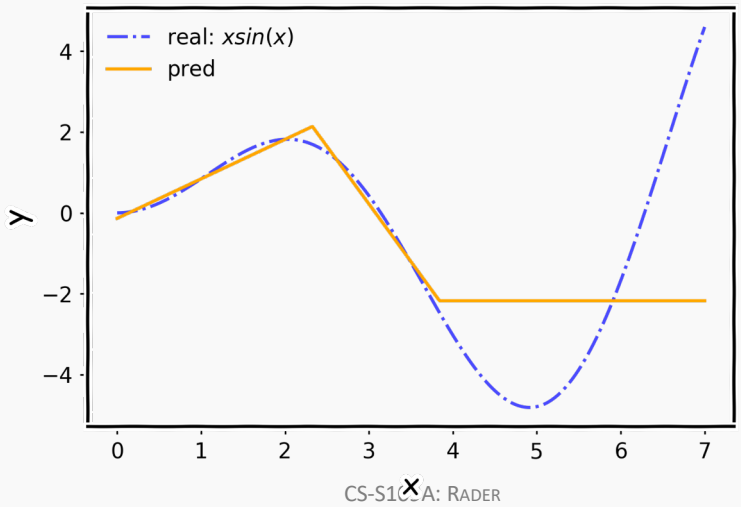
NN in action



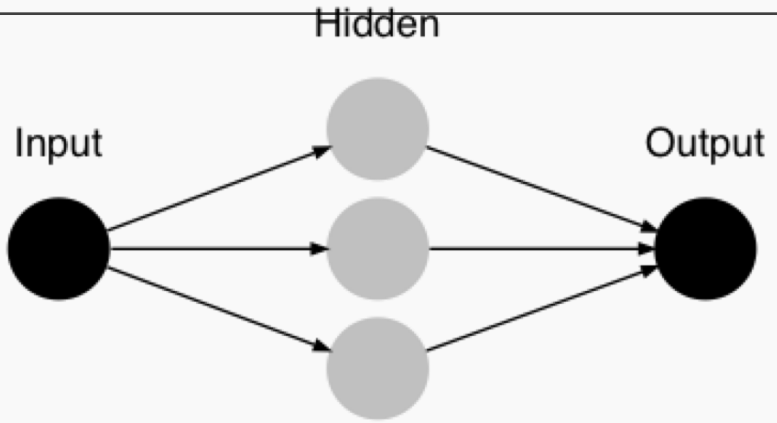


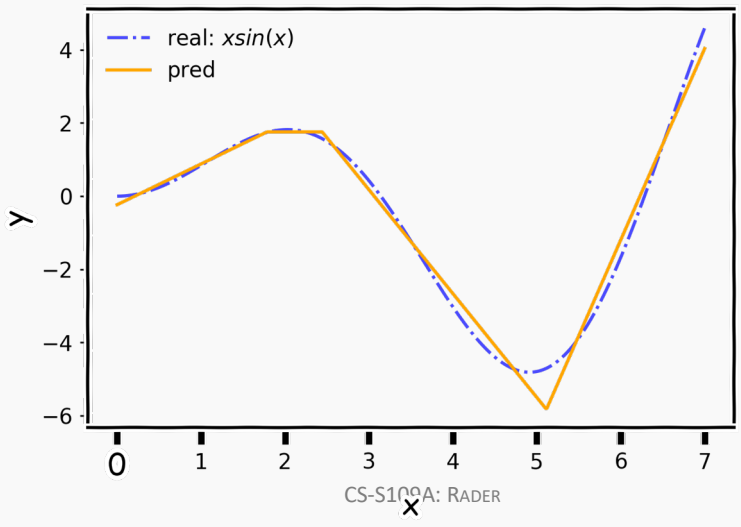
NN in action



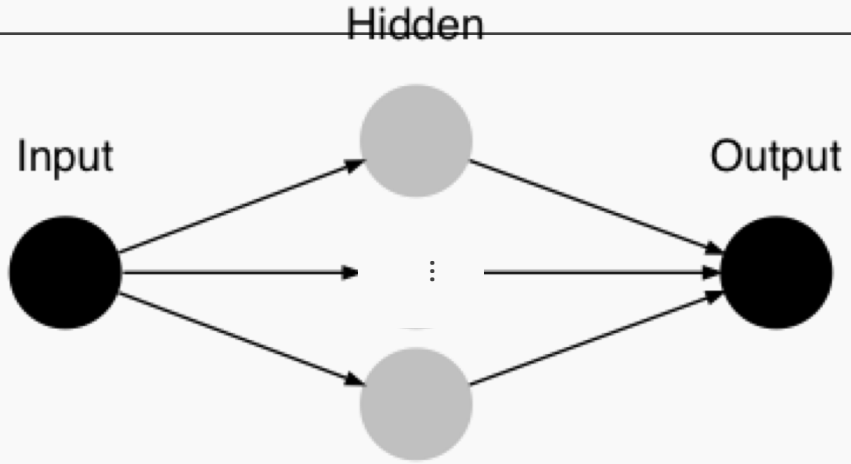


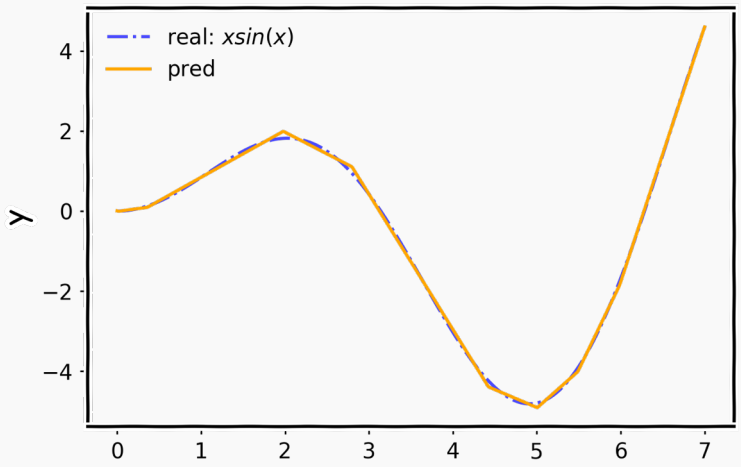
NN in action



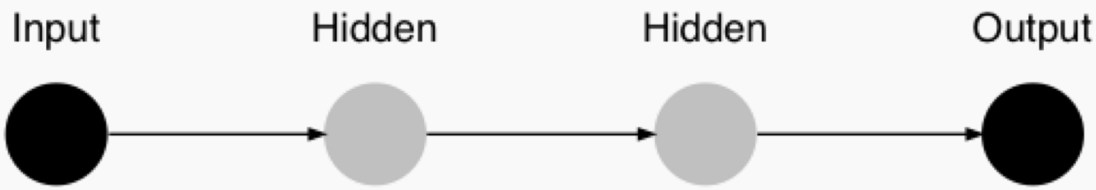


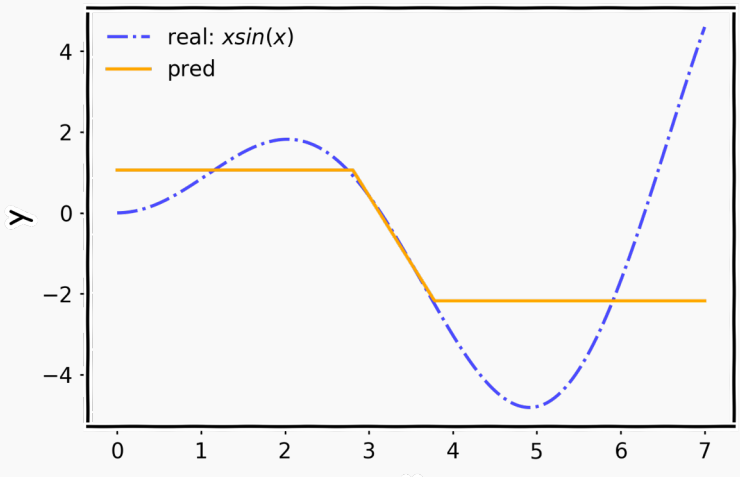
NN in action



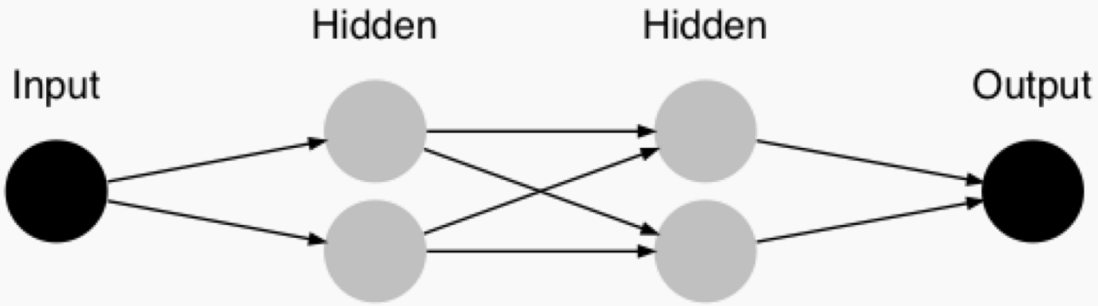


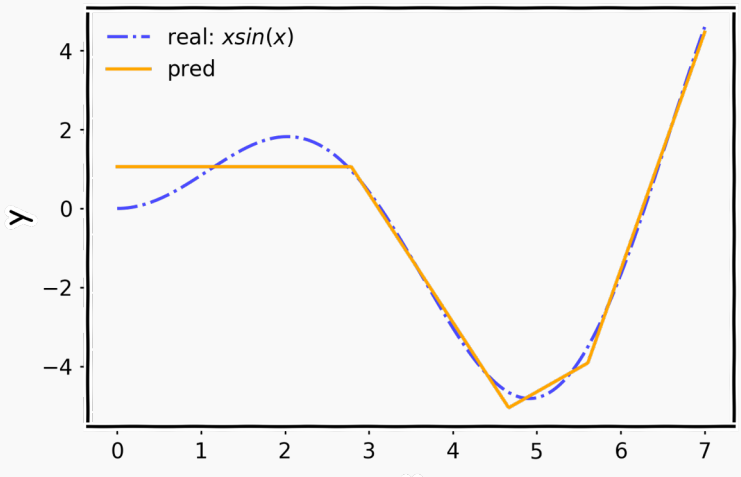
NN in action


Input
Hidden
Hidden
Output

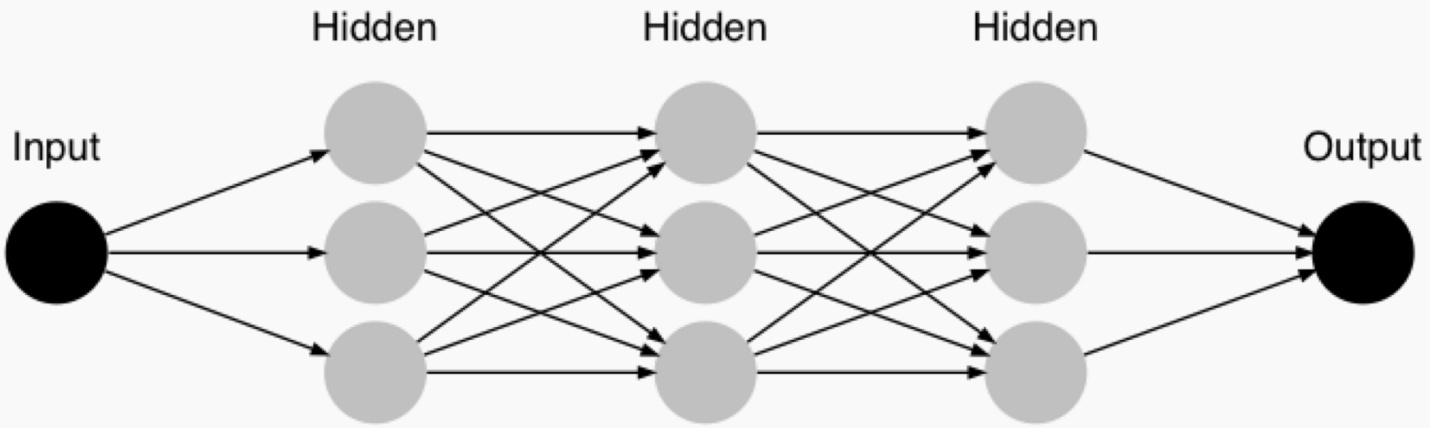

real: xsin(x)
pred
4
2
0
−2
−4
y
0
1
2
3
4
5
6
7
x

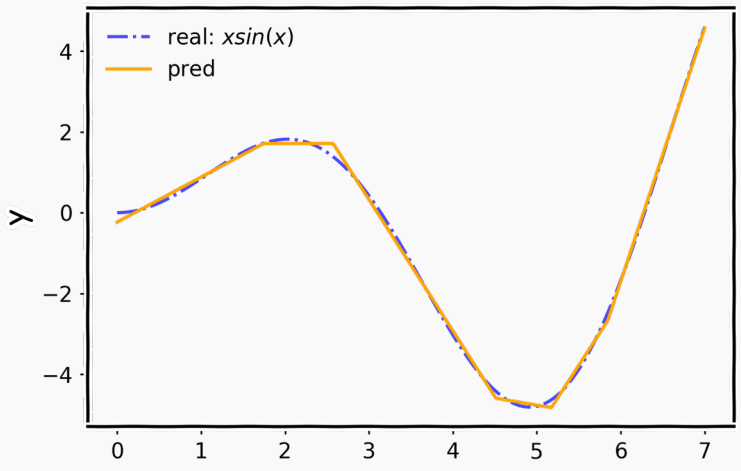
NN in action


Input
Hidden
Hidden
Output


real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

NN in action


Hidden
Hidden
Hidden
Input
Output


real: xsin(x)
pred
y
x
4
2
0
−2
−4
0
1
2
3
4
5
6
7

Universal Approximation Theorem

Think of a Neural Network as function approximation.

$$Y = f(x) + \epsilon$$

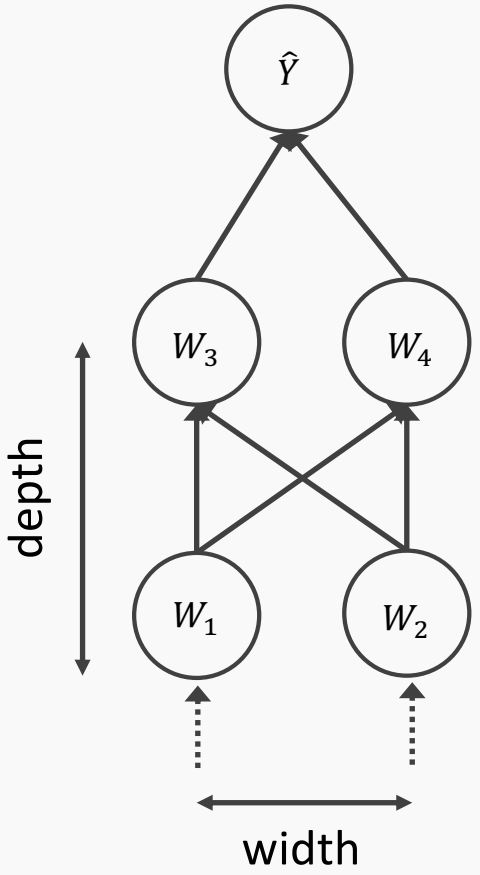
$$Y = \hat{f}(x) + \epsilon$$

$$\text{NN:} \Longrightarrow \hat{f}(x)$$

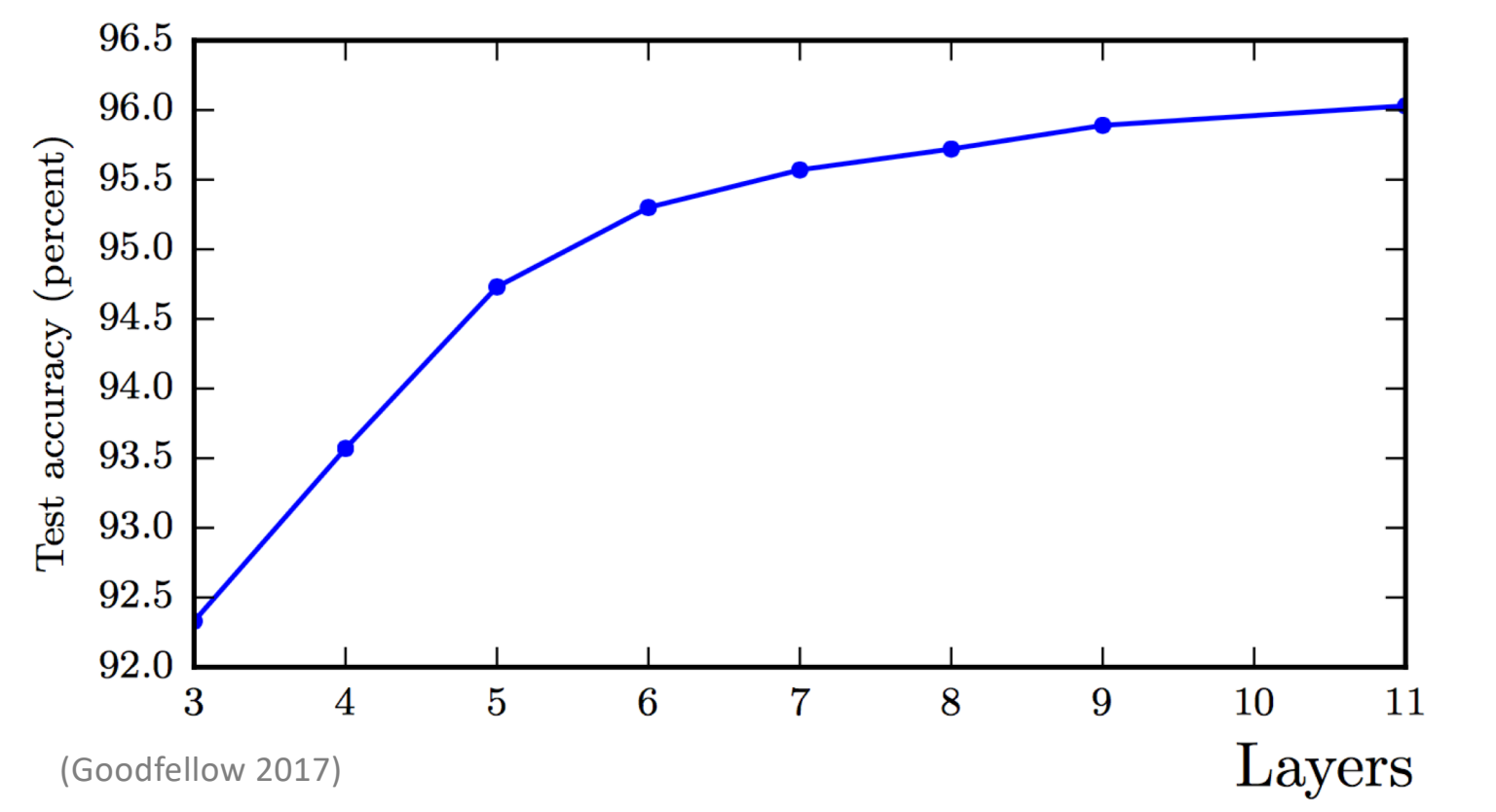
One hidden layer is enough to *represent* an approximation of any function to an arbitrary degree of accuracy

So why deeper?

- Shallow net may need (exponentially) more width
- Shallow net may overfit more



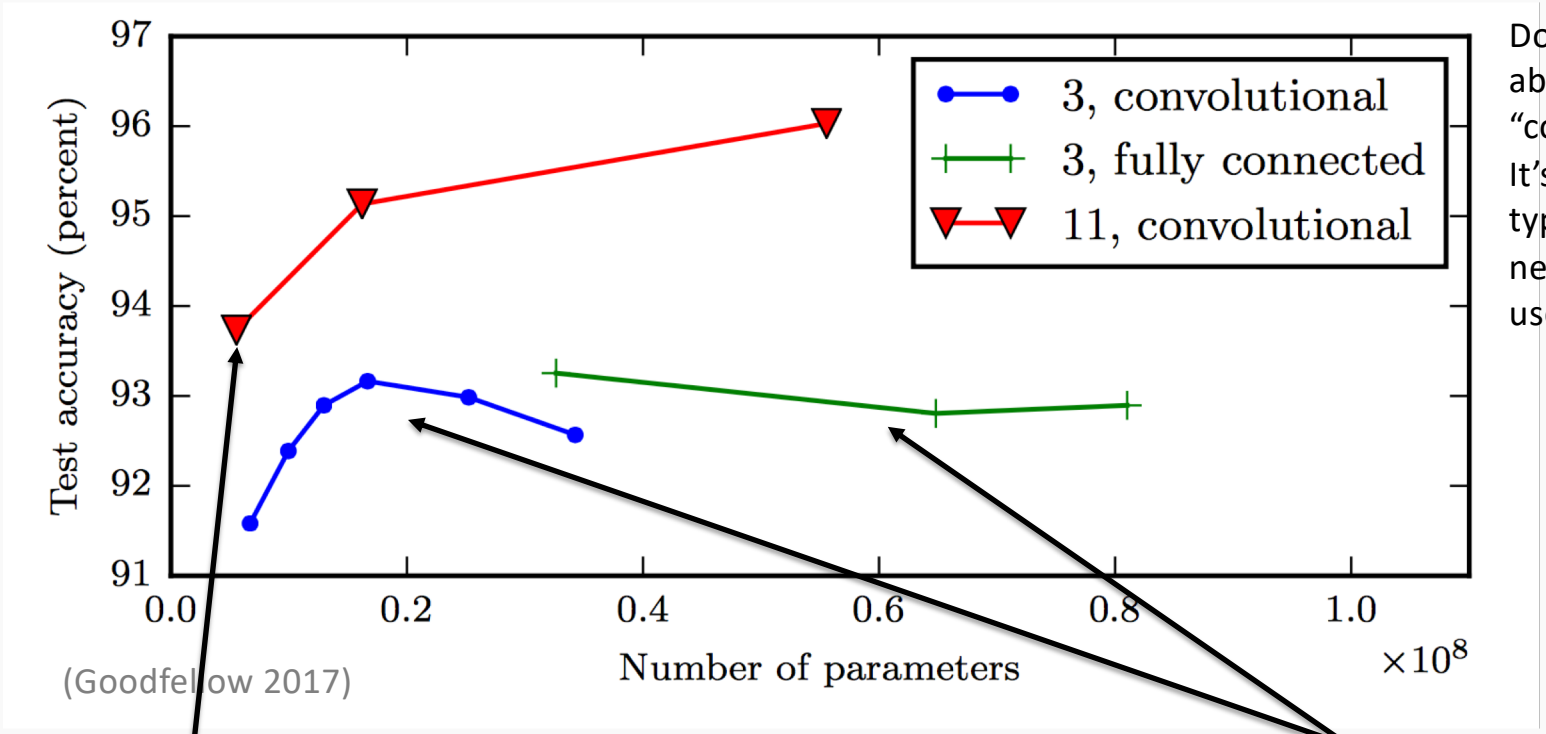
Better Generalization with Depth



(Goodfellow 2017)

Shallow Nets Overfit More

Depth helps, and it's not just because of more parameters

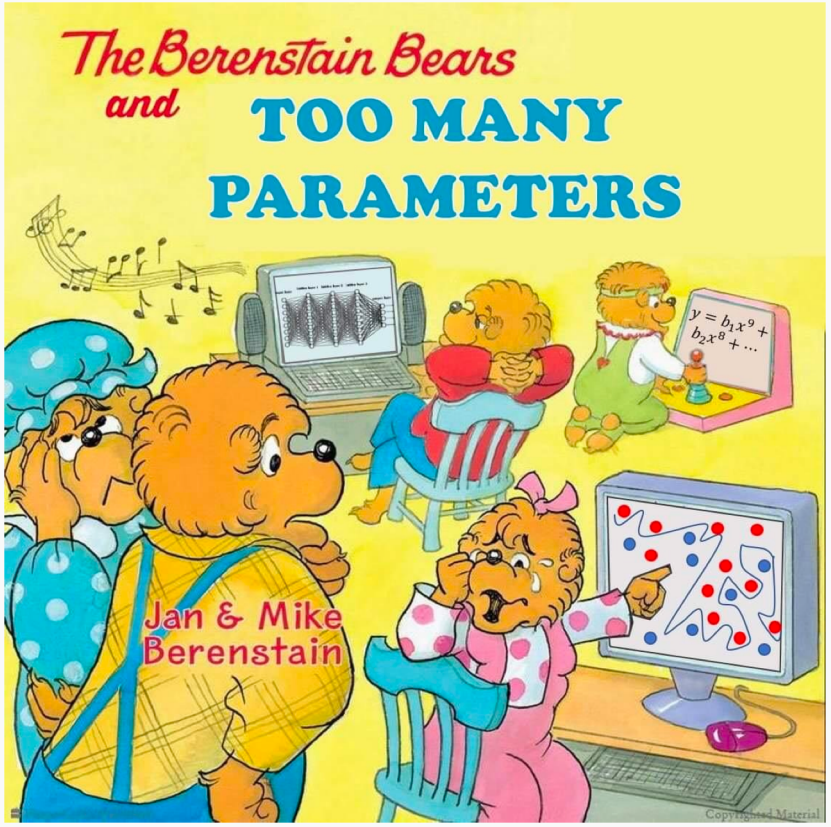


(Goodfellow 2017)

Don't worry about this word "convolutional". It's just a special type of neural network, often used for images.

The **11-layer net** generalizes better on the test set when controlling for number of parameters.

The 3-layer nets perform worse on the test set, even with similar number of total parameters.


The Berenstain Bears
and
TOO MANY
PARAMETERS
$y = b_1x^9 + b_2x^8 + \ldots$
Jan & Mike
Berenstain
Copyrighted Material