Machine Learning

Ensembles+NN

Dan Goldwasser

dgoldwas@purdue.edu

Ensemble Learning Algorithms

- In this lecture we will talk about two ensembles
  - Boosting
  - Bagging

- Both combine classifiers of the **same type**, multiple times by modifying the training set

- **Big idea**: *The methods will be used to help control the **bias** and **variance** in a different way*

Theoretical Motivation

- **"Strong" PAC algorithm *(for class H):***
  - for *any distribution*
  - $\forall$ ε, δ> 0
  - Given polynomially many random examples
  - Finds hypothesis with error ≤ ε with probability ≥ (1-δ)

- **"Weak" PAC algorithm**
  - Same, but only for ε ≥ ½ - γ

Not trivial: for any distribution you can find a hypothesis that performs better than chance (for any training set)

- [Kearns & Valiant '88]:
  - *Does weak learnability imply strong learnability?*
    - *I.e., "can we boost a better-than-chance learner to be as good as we want it to be?"*

A Formal View of Boosting

- Given training set $(x_1, x_1), \ldots, (x_m, y_m)$
- $y_i \in \{-1, +1\}$ is the correct label of instance $x_i \in X$
- For t = 1, … , T
  - Construct a distribution D_t on {1,… m}
  - Find weak hypothesis (*"rule of thumb"*) $h_t : X \to \{-1, +1\}$

    with small error ϵ_t on D_t: $\epsilon_t = P_{D_t}[h_t(x_i) \neq y_i]$

Error is measured according to the distribution at step t!

- **Output**: final hypothesis H_{final}

Source: A Tutorial on Boosting Yoav Freund, Rob Schapire

How can we construct the distribution?

Constructing D_t on {1,..,m}:

$$D_1(i) = \frac{1}{m}$$

Given D_t and h_t :

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \cdot \begin{cases} e^{-\alpha_t} & \text{if} \quad y_i = h_t(x_i) \\ e^{\alpha_t} & \text{if} \quad y_i \neq h_t(x_i) \end{cases}$$

$$= \frac{D_t(i)}{Z_t} \cdot e^{(-\alpha_t y_i h_t(x_i))}$$

Note that the weight of each hypothesis correlates with its error

Where:

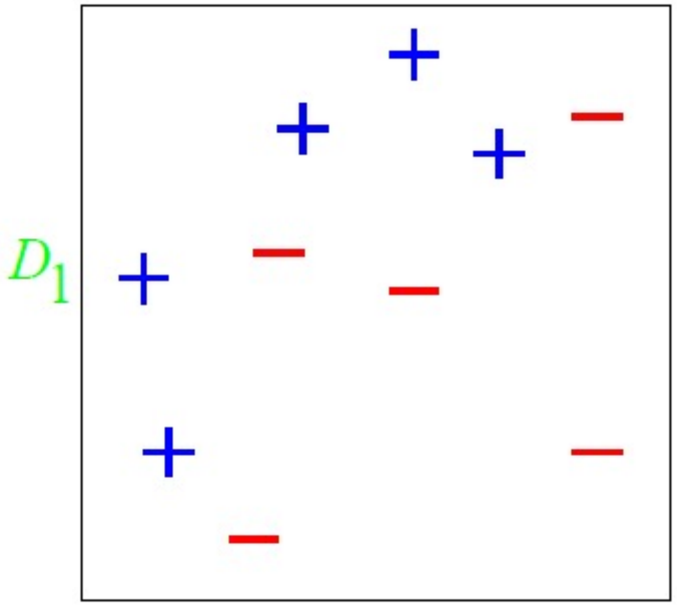
$$Z_t = \text{Normalization constant}$$

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1-\epsilon_t}{\epsilon_t}\right) > 0$$

Final Hypothesis:

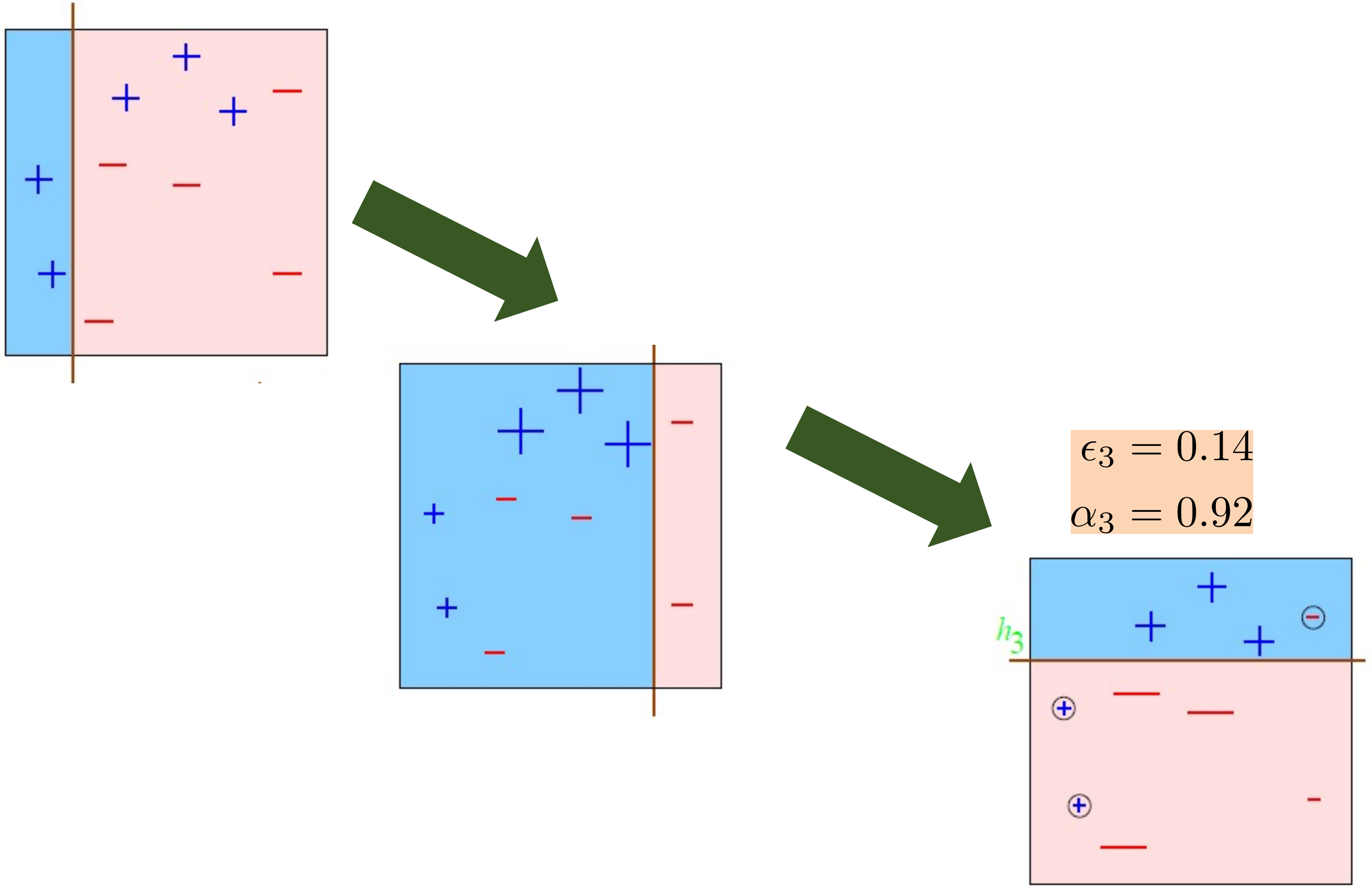
$$H_{\text{final}}(x) = \text{sign}\left(\sum_t \alpha_t h_t(x)\right)$$

A Toy Example



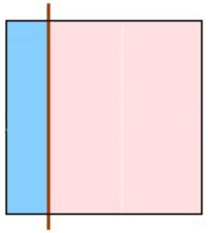
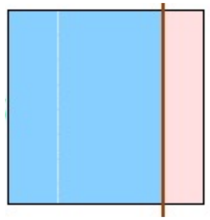
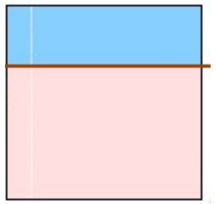
Weak learner: axis parallel lines

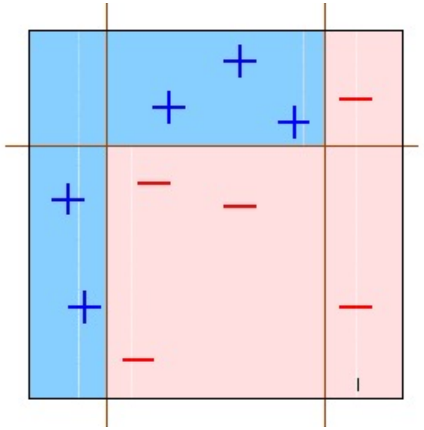
A Toy Example: Round 3



A Toy Example: **Final Hypothesis**

$$H_{Final} =$$

(0.42  + 0.65  + 0.92 )

= 

Note: It is possible that the combined hypothesis makes no mistakes on the training data, but boosting can still learn, by adding more weak hypotheses.

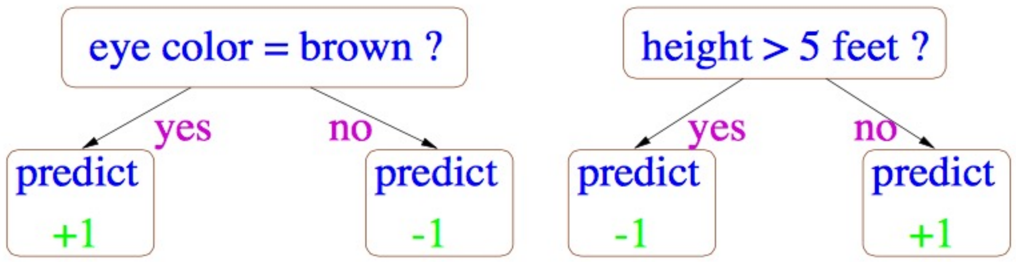
AdaBoost in practice

- **Initialization**:
  - *Weigh all training samples equally*
- **Iteration Step:**
  - Train model on (weighted) train set
  - Compute error of model on train set
  - Increase weights on training cases model gets wrong
    - Focus the next classifier on “difficult” examples
- *Slow convergence*
  - *Typically requires 100’s to 1000’s of iterations*
- **Return final model:**
  - Carefully weighted prediction of each model

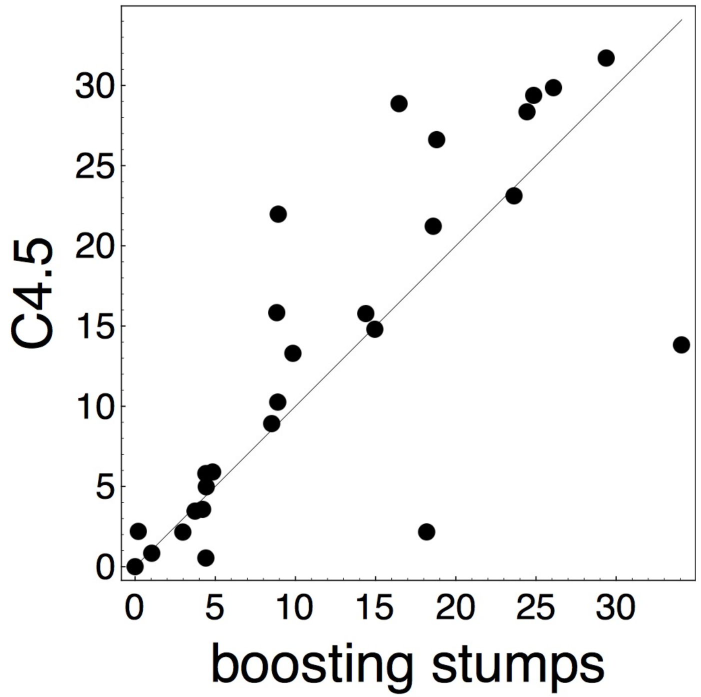
Key idea: change the distribution during training

9

AdaBoost in practice: **Boosting Decision Stumps**



Decision Stumps:
decision rule splitting on one attribute

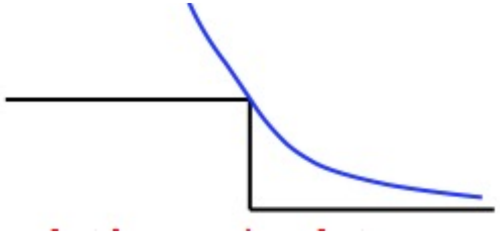


AdaBoost in practice: Loss minimization

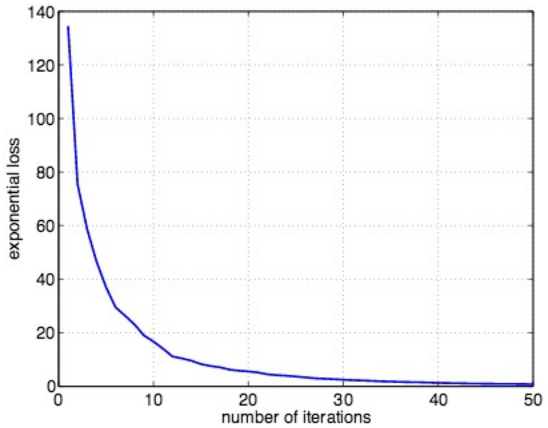
- *AdaBoost corresponds to minimizing the Exponential loss function*

$$C_{ada} = \sum_i \exp[-y^{(i)} f(x^i)]$$

- *Convex and smooth surrogate loss function*

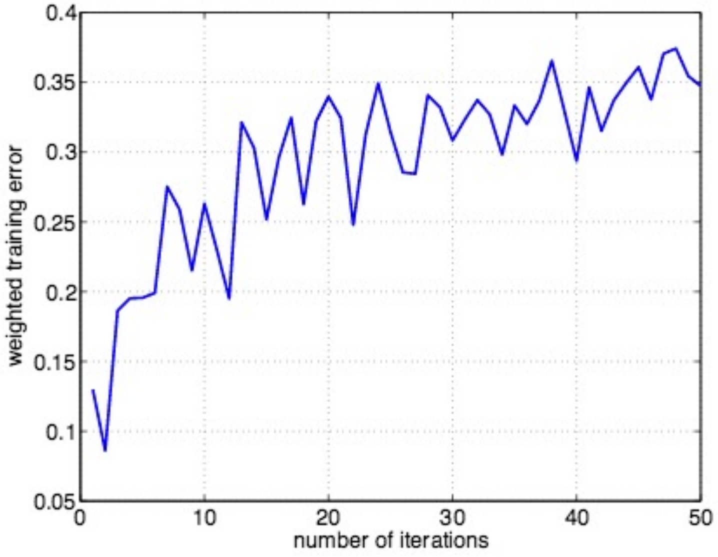


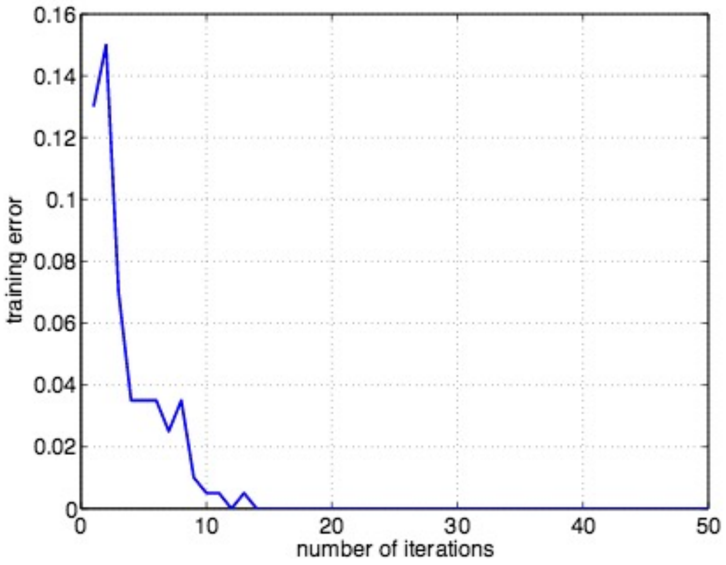
- At each iteration, it will have a lower exponential loss



AdaBoost in practice

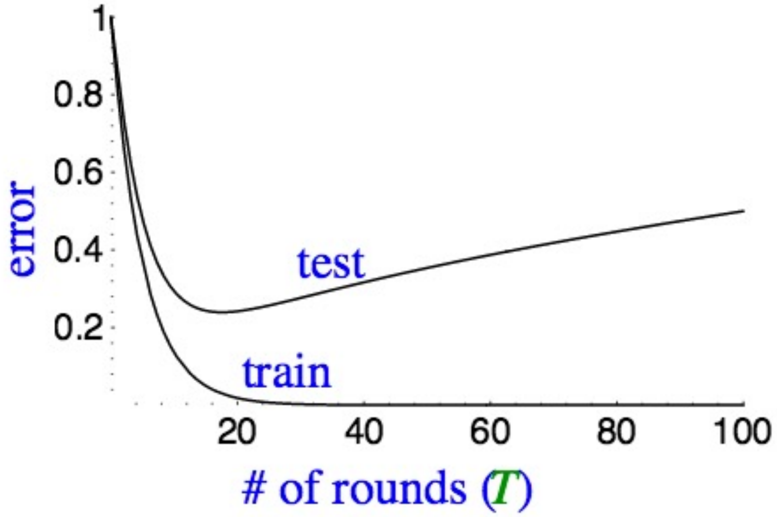
- *Note the difference between the* ***individual classifiers*** *(that tend to get worse over time), and the* ***combined hypothesis*** *(that improves over time)*





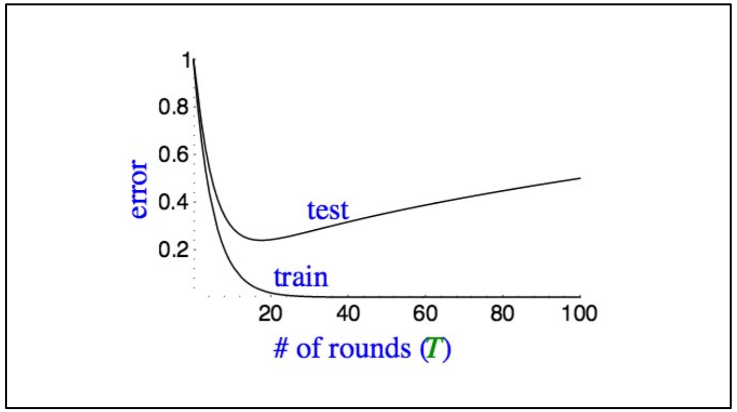
AdaBoost at Test time

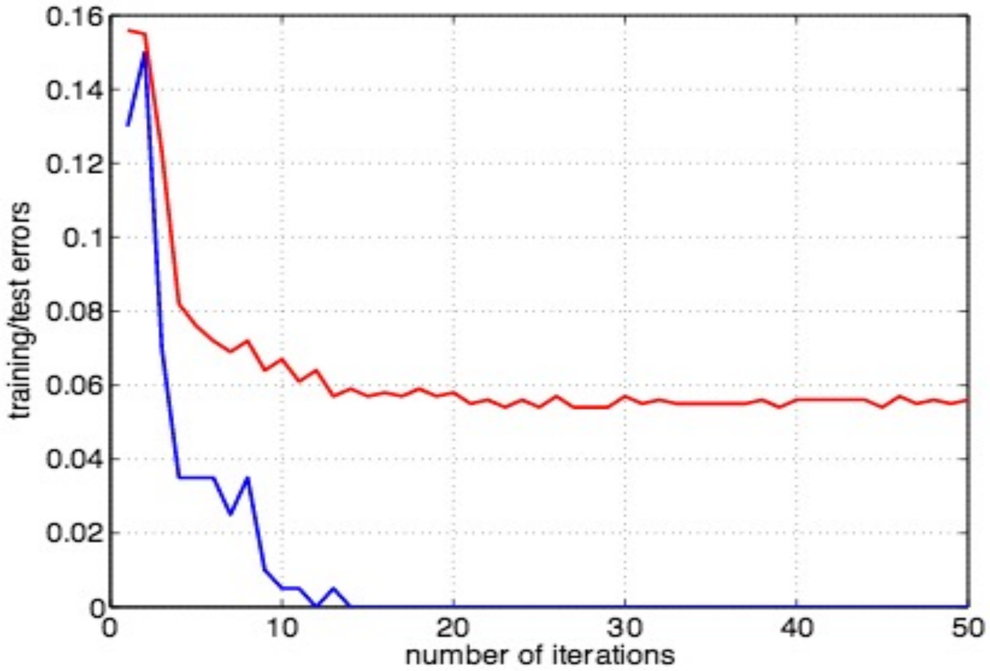
- The training error will reach zero (or keep decreasing) as we make the hypothesis more complex
- As a result, the test error will start increasing as the final hypothesis becomes overly complex



AdaBoost in practice

Interestingly, it does not happen!





Two observations:

(1) Adaboost does not over fit easily

(2)The *test error* can continue to decrease even when we already have *zero training error*

Boosting Summary

- Combine weak learners into a powerful learner
- Reduce bias without increasing variance!
- **Strong points**:
  - Effective and easy to implement
  - You can plug in your favorite learner
  - Only hyperparameter is T
- **Caveats**
  - Will overfit if the weak learners are too complex
  - Will underfit if the weak learners are too weak
    - $\gamma_t \rightarrow 0$ too quickly

Bagging

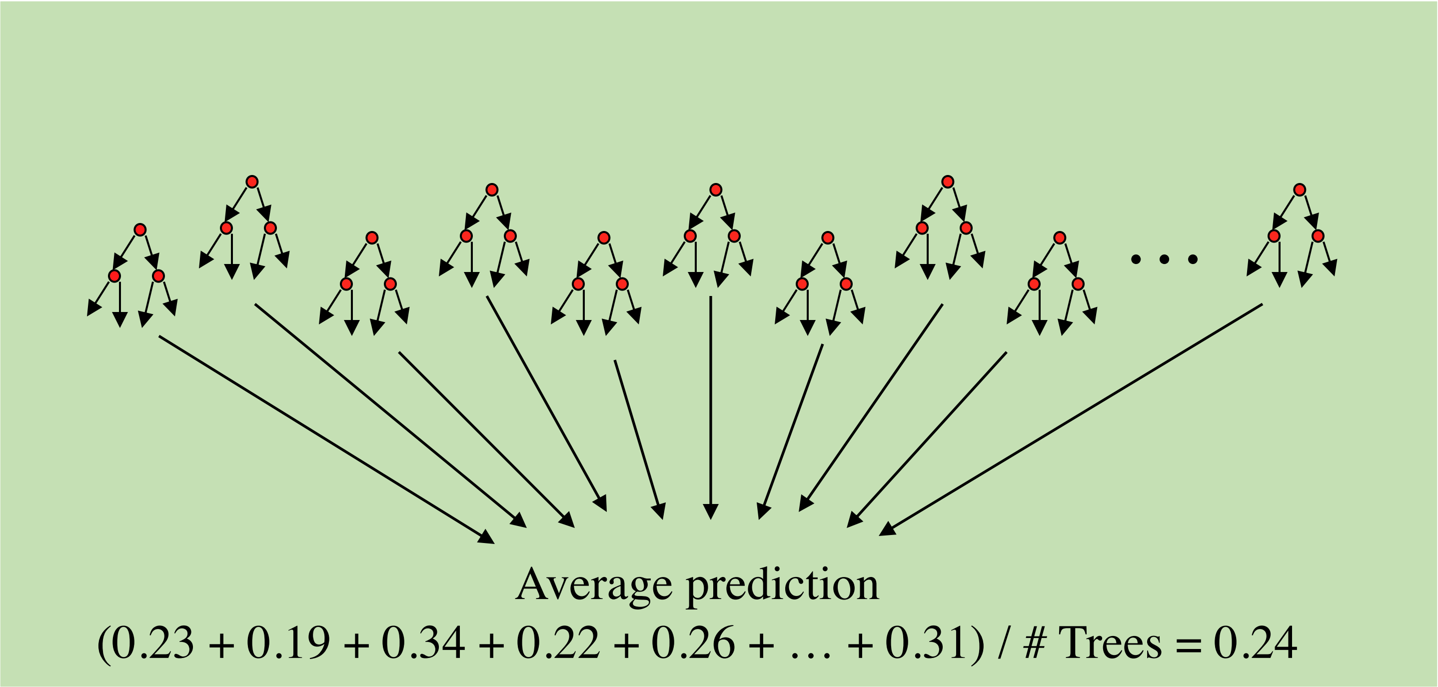
- **Bagging Intuition:**
  - Overfitting occurs when the model start memorizing the data (no generalization)
  - Variations in the data will results in different models
  - Bagging: **bootstrapped Aggregation**
    - Sample smaller sets of the data, each specific learner cannot memorize the entire dataset
  - Appropriate for: *Low bias-High Variance learners* (e.g., expressive learners)
    - Helps reduce variance!

Bagging

- Bagging predictors generates multiple versions of a predictor and uses these to get an **aggregated predictor**
- The aggregation **averages over the versions** when predicting a numerical value and a **majority vote** when predicting a class.
- The multiple versions are formed by making bootstrap replicates of the learning set and using these as new learning sets.
    - *That is, use samples of the data, with repetition*
- The vital element is the **instability of the prediction** method. If perturbing the learning set can cause significant changes in the predictor constructed then bagging can improve accuracy.

Example: Bagged Decision Trees

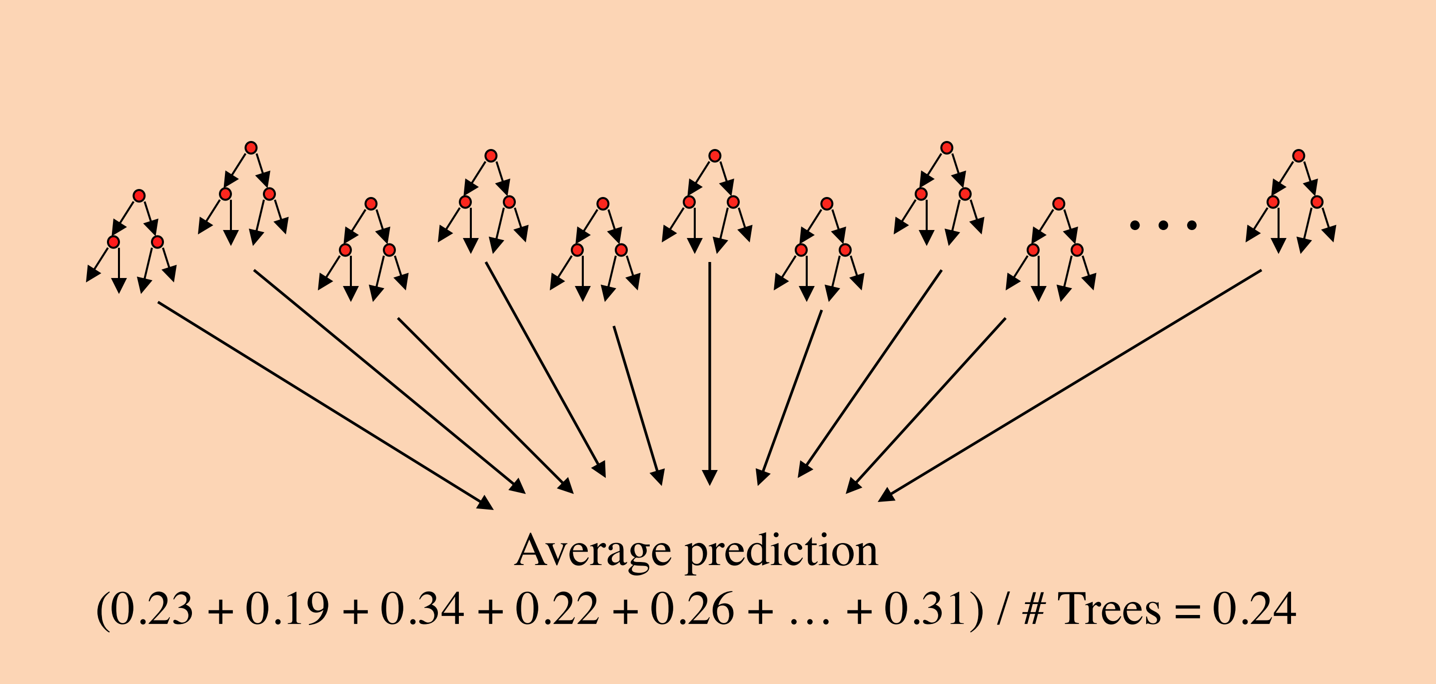
- Draw 100 bootstrap samples of data
- Train trees on each sample → 100 trees
- Average prediction of trees on test examples (or majority vote)



Random Forests (*Bagged Trees++*)

Key idea: sample data (bagging) + sample features

- Draw ***1000+*** bootstrap samples of data
- ***Draw sample of available attributes at each split***
- Train trees on each sample/attribute set → ***1000+*** trees
- Average prediction of trees on out-of-bag samples



Goals for today

- *Neural networks introduction*
  - *What can and can't be learned by linear models?*
  - *NN intuition: engineering vs.* ***learning*** *a new representation for the data*
- *We will start with a basic intro to NN*
  - *Representation, training*
- *..and we'll move to NLP specific architectures*
- ***Big Questions:***
  - *How/whether to account for linguistic structure*
  - *How to model the interactions between words/sentences*
  - *How to do "neural reasoning" in complex problems?*

Dealing with Structures

- Structured prediction – dealing with multiple decisions at the same time. Modeling the interactions between decisions is the key challenge.

Conceptually two considerations:

- Entity **boundary (Begin, Inside ,Outside)**
- Entity **type (PER,ORG,LOC)**

Begin$_p$

Inside$_p$

Barak Obama visited Mount Sinai hospital in New York

PER

ORG

LOC

Many of the predictions are context dependent: e.g., Mount Sinai is a LOC, while Mount Sinai hospital is an ORG.

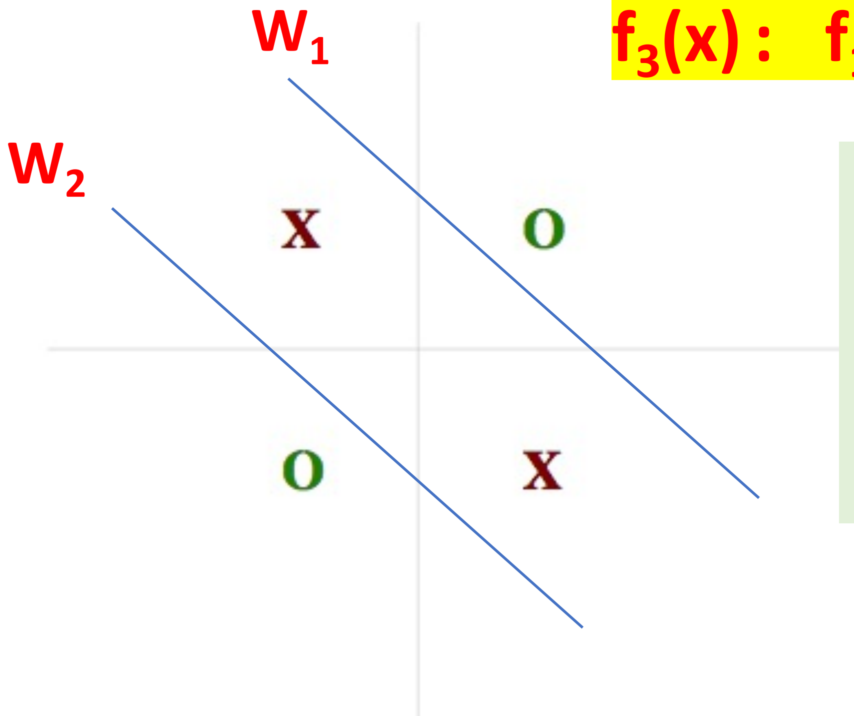
How can you capture it using the machinery we currently have? At what cost?

Linear vs. Non Linear Classification

- Up to this point we focus on linear classifiers.
- They depend on *manually finding expressive* features which define simple learning problems.
- Simple solutions often break.
  - BoW is hard to beat, and works great for simple problems
    - Harder cases: nuanced problems, domain differences
    - Tends to blowup the feature space.
- Non-Linear classifiers – complex decision boundaries
  - Decision trees, **neural-nets**,..
  - NN dynamically learn a feature representation

Limitations of Linear Models

$f_3(x)$: $f_1(x)$ OR $f_2(x)$



Can you find a way to combine linear models to solve this problem?
(i.e., chain their predictions)

Learn *a model for* ***representing*** *the data, that would simplify the problem,* **now build a simple classifier over it.**

Big idea behind neural net – *jointly learn the representation to make classification easier with the classification problem.*

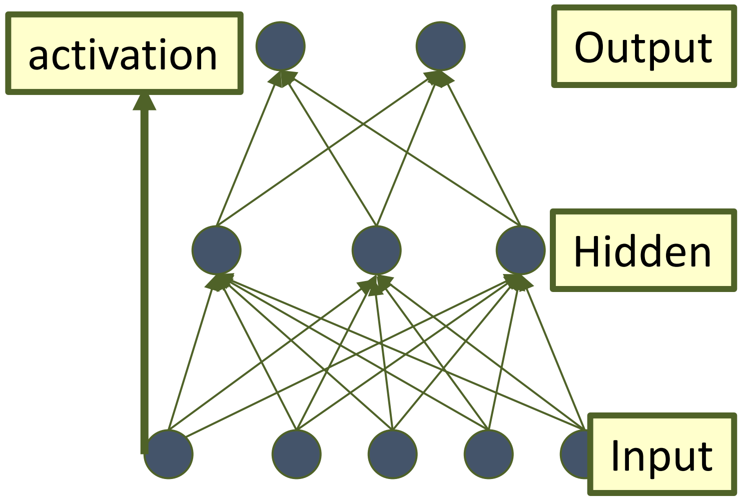
Neural Network

- Simply put, NN's are functions **f: X→Y**
  - *f is a **non-linear** function*
  - *X is a **vector** of continuous or discrete variables*
  - *Y is a **vector** of continuous or discrete variables*
- **Very expressive classifier**
  - In fact, NN can be used to represent any function
- The function f is represented using a network of logistic units

Multi Layer Neural Networks

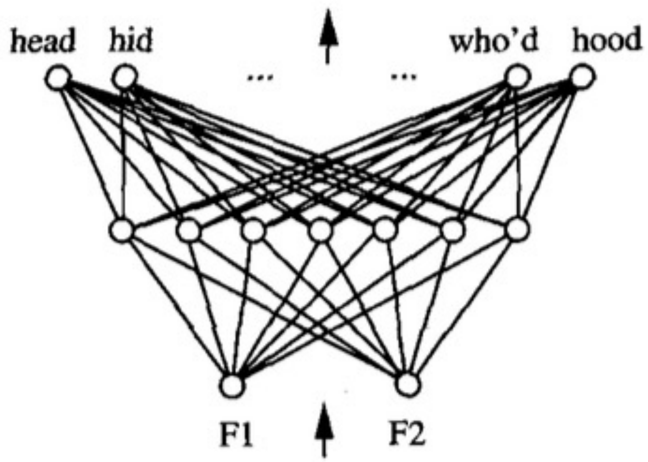
- Multi-layer network were designed to overcome the computational (**expressivity**) limitation of a single threshold element.

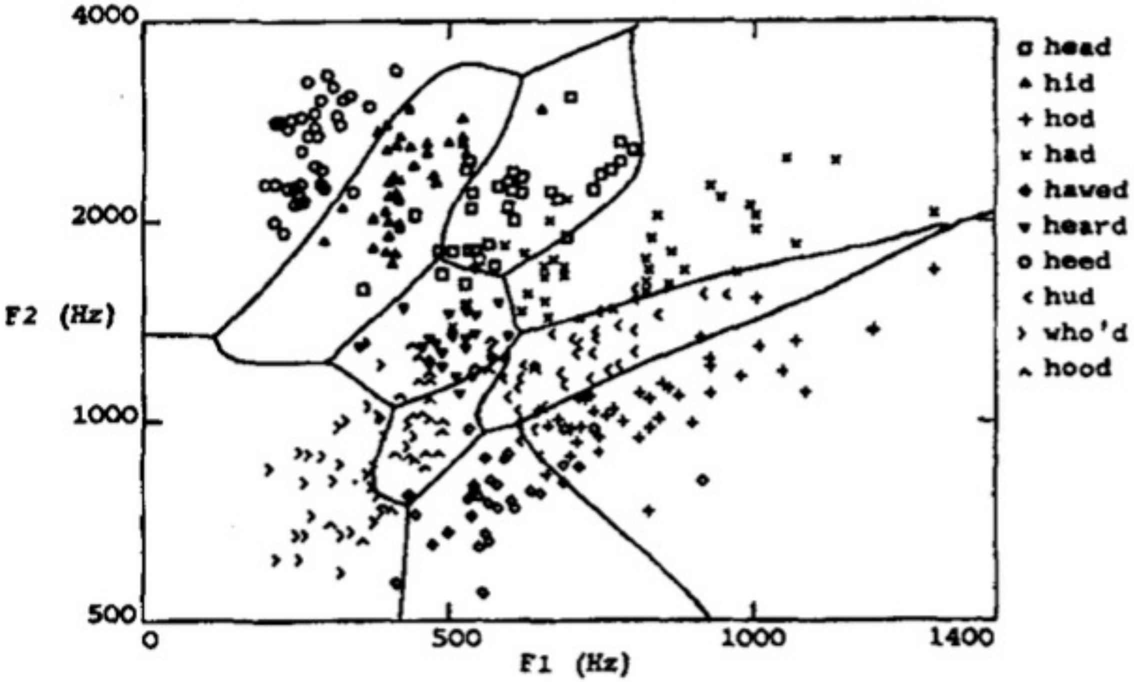
- The idea is to **stack** several layers of threshold elements, *each layer using the output of the previous layer as input*



Multi-layer networks **can represent arbitrary functions**, but building effective learning methods for such network was/is difficult.

Example: NN for speech vowel recognition





ALVINN: autonomous land vehicle in a NN

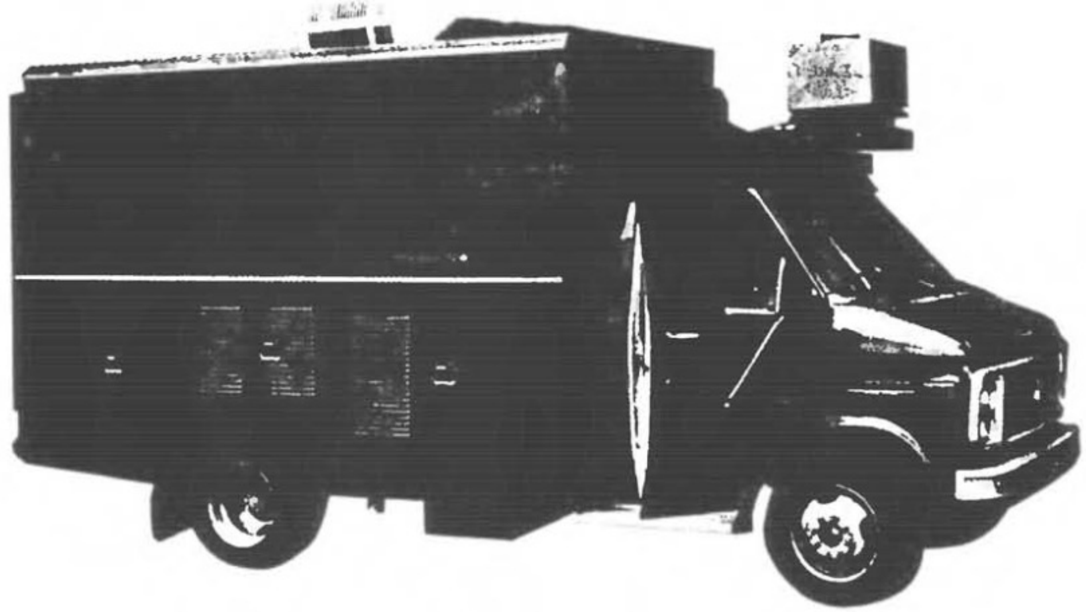
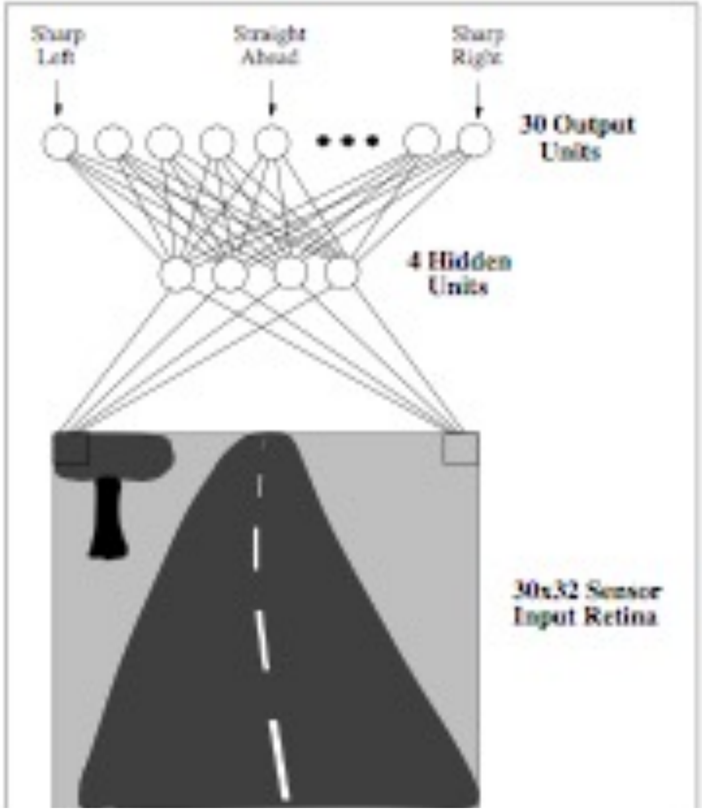
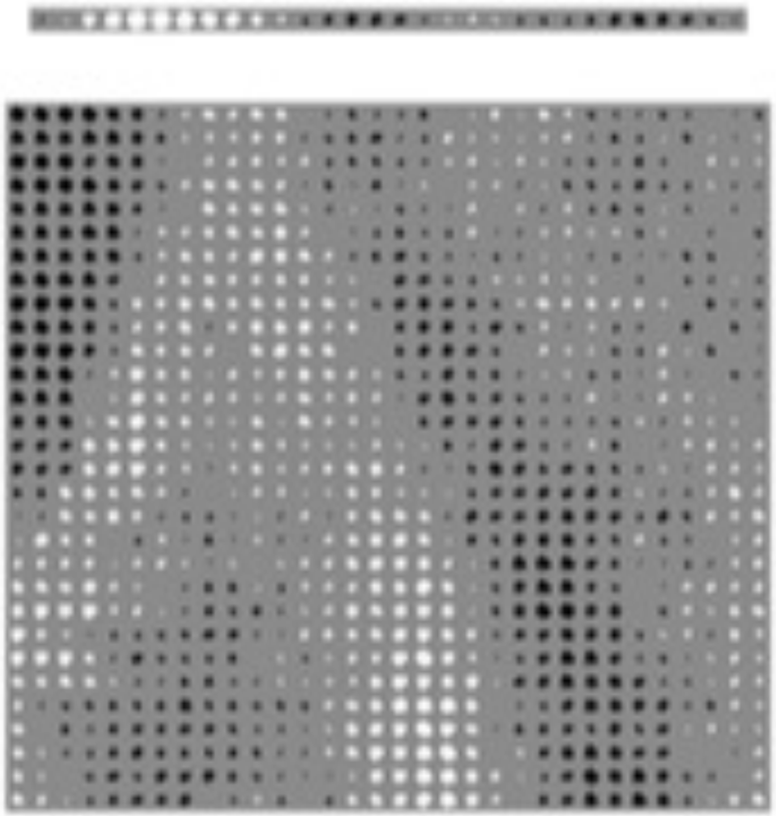


Figure 3: NAVLAB, the CMU autonomous navigation test vehicle.

Pomerleau ‘89

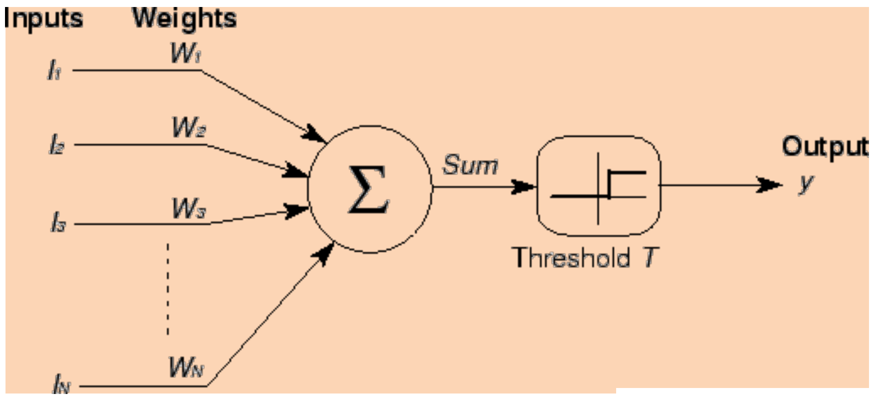
ALVINN: autonomous land vehicle in a NN

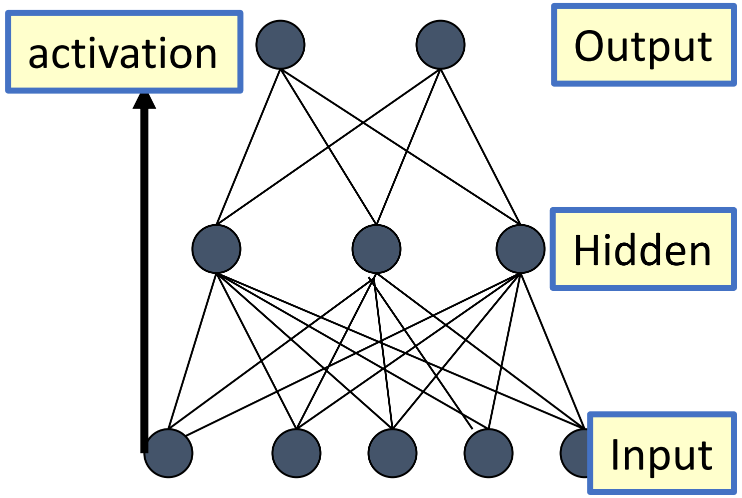




Basic Units in Multi-Layer NN

- Basic element: **linear unit**
  - But, we would like to represent nonlinear functions
    - Multiple layers of linear functions are still linear functions
  - Threshold units are not smooth (we would like to use gradient-based algorithms)



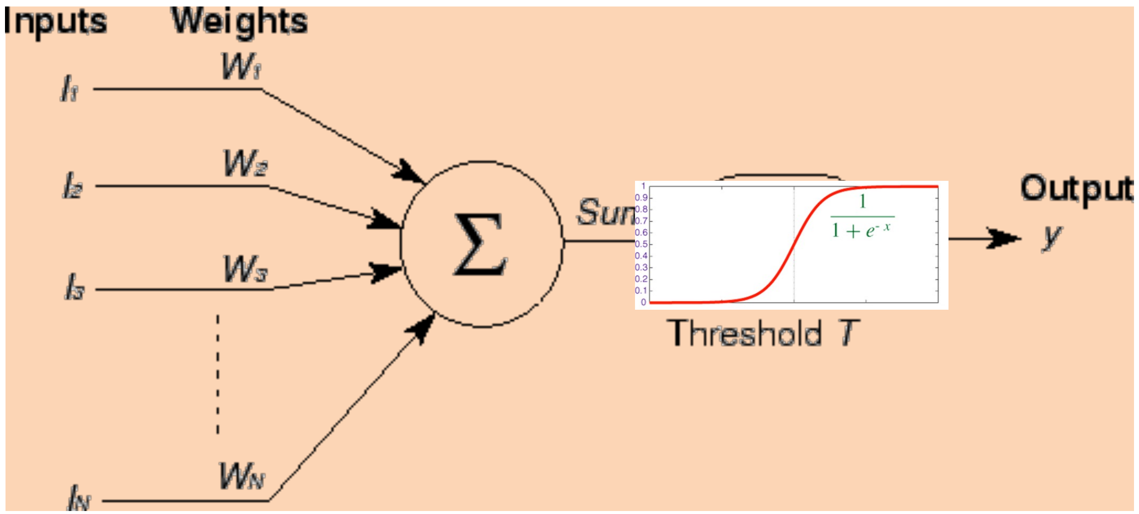


Basic Units in Multi-Layer NN

- Basic element: **sigmoid unit**
  - Input to a unit *j* is defined as: $\Sigma w_{ij} x_i$
  - Output is defined as : $\sigma(\Sigma w_{ij} x_i)$
    - σ is simply the logistic function:

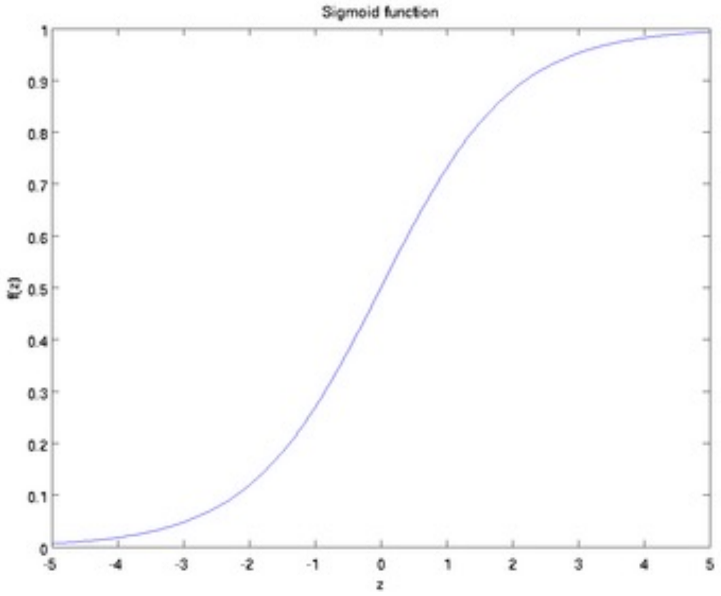
$$\frac{1}{1+e^{-x}}$$

Note: similar to previous algorithms, We encode the bias/threshold, as a "fake" Feature that is always active

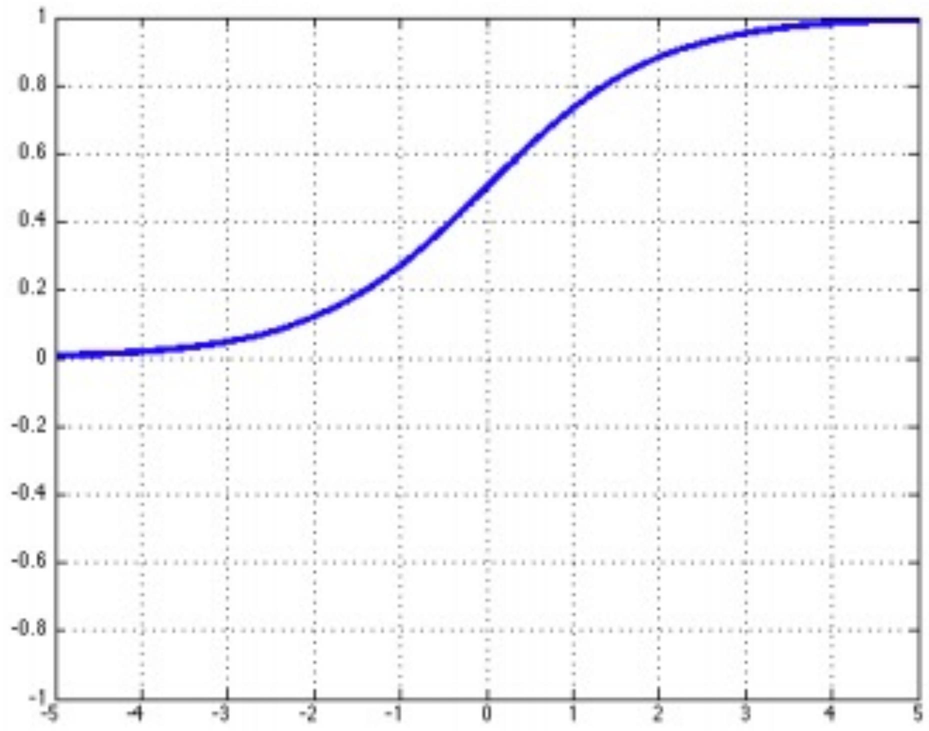


Basic Units in Multi-Layer NN

- Basic element: **sigmoid unit**
  - You can also replace the logistic function with other smooth activation functions



Sigmoid

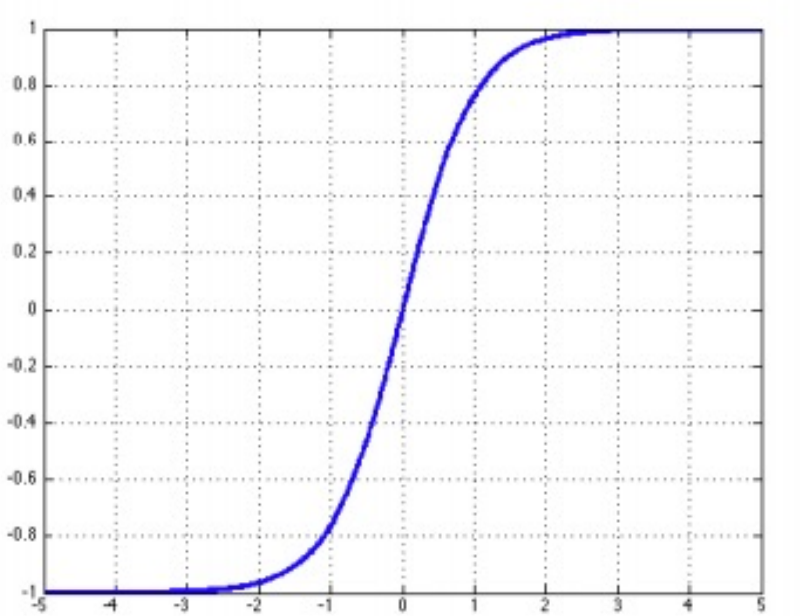


$$\sigma(z) = \frac{1}{1 + \exp(-z)}$$

Images taken from web.stanford.edu/class/cs224n

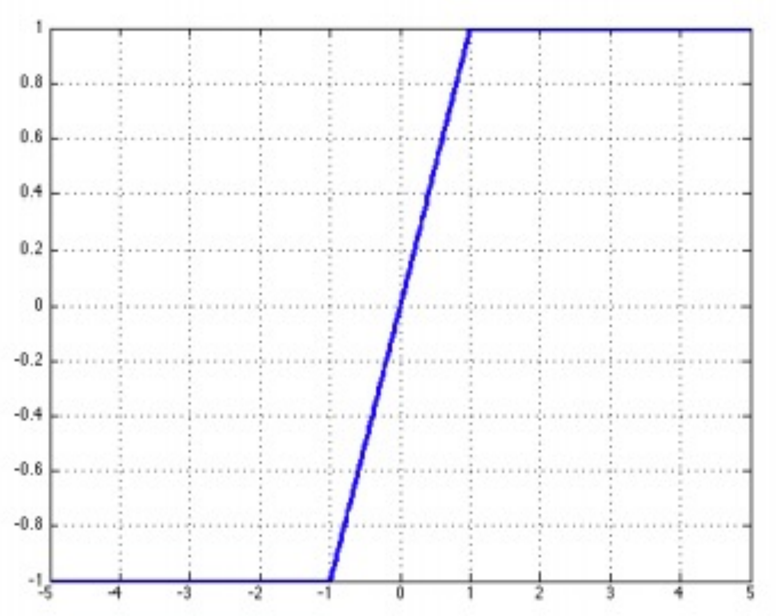
Tanh

• sd



$$\tanh(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)} = 2\sigma(2z) - 1$$

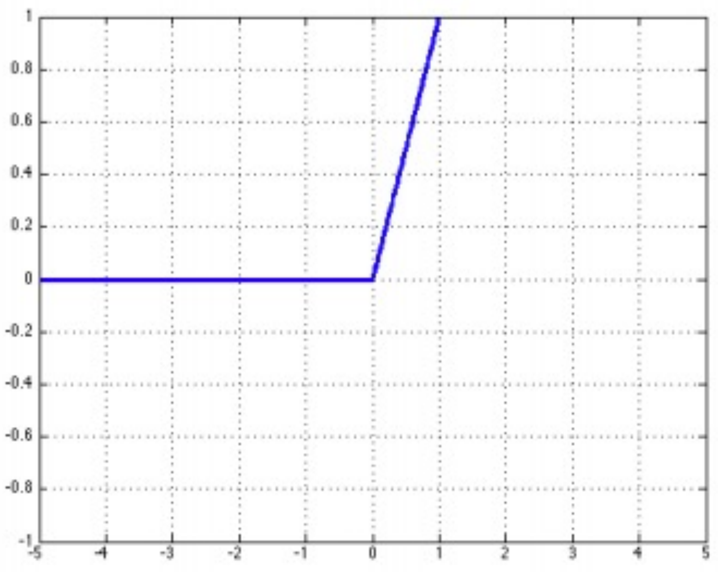
where $\tanh(z) \in (-1, 1)$



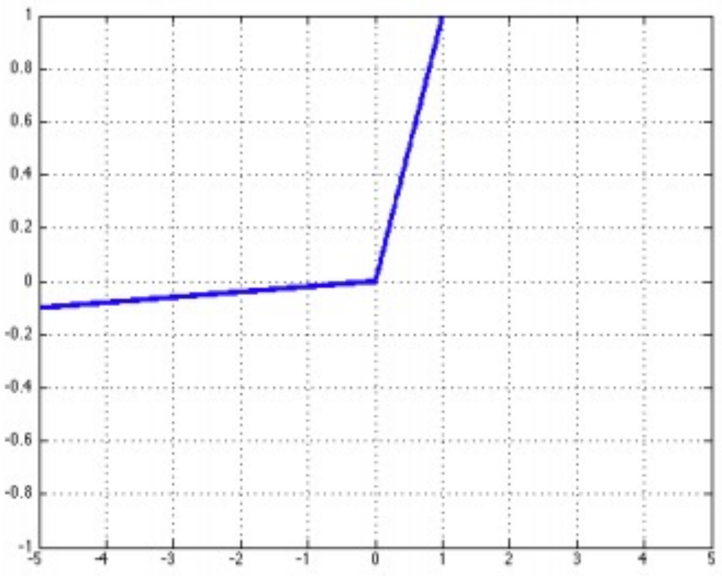
$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \le z \le 1 \\ 1 & : z > 1 \end{cases}$$

Images taken from web.stanford.edu/class/cs224n

Rectified Linear Units (RelU)



$$\mathrm{rect}(z) = \max(z, 0)$$



$$\mathrm{leaky}(z) = \max(z, k \cdot z)$$

$$\text{where } 0 < k < 1$$

Images taken from web.stanford.edu/class/cs224n

Multi Layer NN

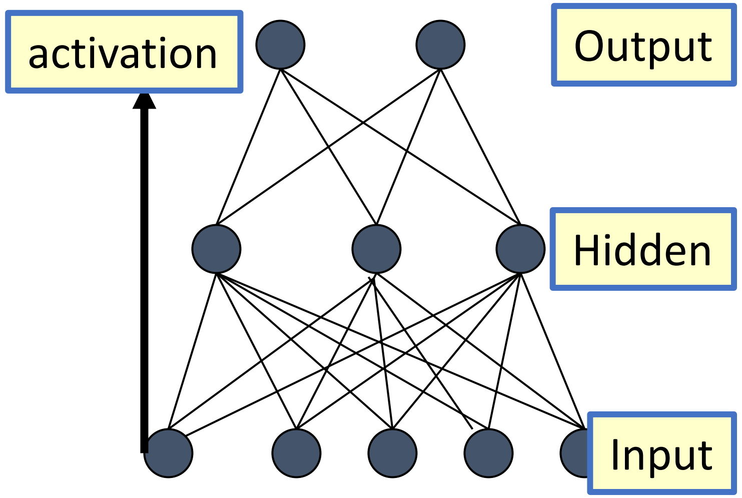
- Another approach for increasing expressivity:
  Stacking multiple units to form a network
- Compute the output of the network using a ‘feed-forward’ computation
- Learn the parameters of the network using the backpropagation algorithm
- Any Boolean function can be represented using a two layer network
- Any bounded continuous function can be approximated using a two layer network

Multi Layer NN: forward computation

- Observe an input vector x
- *Push x through the network:*
  - For each **hidden unit** compute the activation value

$$h_j = \sigma(t_j) = \sigma(\sum_i w^1_{ji}\, x_i)$$

  - For each **output value**, compute the activation value coming from the hidden units



  - **Prediction:** $$\hat{y}_k = \sigma(s_k) = \sigma(\sum_j w^2_{kj} h_j)$$
    - **Categories**: winner take all
    - **Vector**: take all output values
    - **Binary outputs**: Round to nearest 0-1 value