MATH 3ABW Lec 24

Short Survey on other methods we didn't have time for...

- Deep Learning
  - Image Recognition using CNN's (2000s)
  - Transformers (not covered)
- Machine Teaching: Assume you can generate $\vec{x}_*$'s from a distr. cheaply but y_* is more expensive. Its
  Begin with $\mathcal{D} \xrightarrow{A} g$. Then inspect $\vec{e}$. Which $|e_i|$'s are large?
  Generate $\langle \vec{x}_*, y_* \rangle \approx \vec{x}_i$'s that produced large $|e_i|$'s
  Let $\mathcal{D} = \left\{ \begin{bmatrix} X \\ \vec{x}_* \end{bmatrix}, \begin{bmatrix} \vec{Y} \\ y_* \end{bmatrix} \right\} \xrightarrow{A} g$. Repeat until model has low $|e_i|$'s for all $\vec{x}$ locations or until you run out of resources.
- Reinforcement Learning: same as machine teaching except you generate the $\vec{x}$ entries by interacting with a dynamic system which then gives you y's.
  For instance: playing a video game. y = # of points you get or beat the level?
  x's are everything happening on the screen and the record of your "moves".
- Unsupervised Learning
  You only see X; there is no $\vec{y}$. Typical problems:
  - Clustering the units into groups $G_1, G_2, \ldots, G_K \subseteq \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\}$ (hierarchical clustering, K-means clustering) $G_1 \cup \ldots \cup G_K = X$, $G_i \cap G_j = \emptyset$ $i \neq j$
  - Anomaly detection. If the $\vec{x}_i$'s are realizations from a data generating process, which $\vec{x}_i$'s don't seem to be following the process?
  - Dimension reduction: instead of p features, can you squeeze the same amount of information out of $p' < p$ dimensions?

Principal Components Analysis (PCA) and Factor Analysis

K-means clustering. Consider data X which is pre-normalized (all col means = 0 and sd = 1).

(1) Pick the # of clusters K, and distance function d (default is Euclidean)

(2) Randomly assign K of the n units to be the "centroids" $\vec{x}_1, \vec{x}_2, \ldots \vec{x}_K$
$G_1 = \{\vec{x}_?\}$, $G_2 = \{\vec{x}_?\}, \ldots, G_K = \{\vec{x}_?\}$

(3) For each remaining $n-K$ units, compute the distance to each of the K centroids i.e. for all i

$$d_1 = \|\vec{x}_i - \vec{x}_1\|^2$$
$$d_2 = \|\vec{x}_i - \vec{x}_2\|^2$$
$$\vdots$$
$$d_K = \|\vec{x}_i - \vec{x}_K\|^2$$

(4) Now assign the i^{th} unit to the cluster which minimizes the distance to its centroid i.e. $\vec{x}_i$ is added to group G_{k^*} where $k^* = \text{argmin}\{d_k\}$

(5) Recompute centroids as

$$\vec{x}_{k^*} := \frac{1}{|G_{k^*}|} \sum_{\vec{x} \in G_{k^*}} \vec{x}$$

(6) For all units, repeat steps 3-6 until "convergence" (where no membership changes for any unit in their group)

(7) Calculate within for each cluster $D_k := \sum_{\vec{x} \in G_k} d(\vec{x}, \vec{x}_k)$ then add all distances for all clusters to get a fit metric $\sum_{k=1}^{K} D_k$

(8) Repeat steps 1-7 with different starting pts and record best starting pt

(9) Repeat steps 1-7 for K values $\in \{1, 2, \ldots, K_{max}\}$

(10) Select optimal K value visually using "the bend" as a guide ("scree plot")

Artificial Neural Networks / ANN's

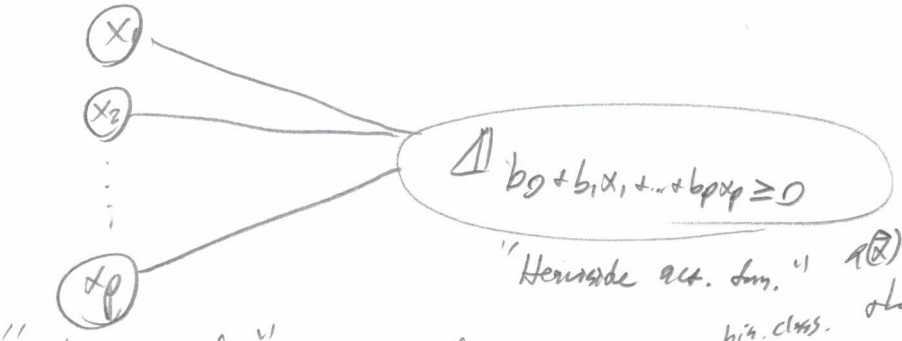
Neural Networks / Neural "Nets" / NN's

Another perspective on our models. Layer: a set of L_1 functions which take input from a vector

"activation"

For binary classification:

"Input Layer" "Output Layer" $(L=1)$



"Heaviside act. fun." $q(z)$

"Identity act. fun"

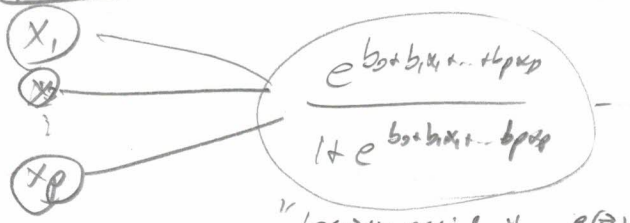
the A estimates the values of $b_0, b_1, \dots, b_p$

or for prob est in bin. class. then...

for binary classification

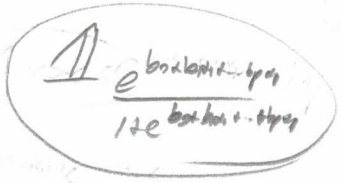
Input Layer

Output Layer



"Logistic activation" $q(z)$

or

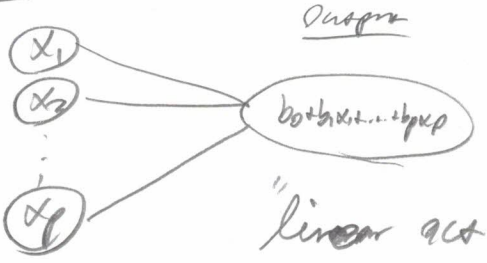


for binary classification

Input

or for regression

Output



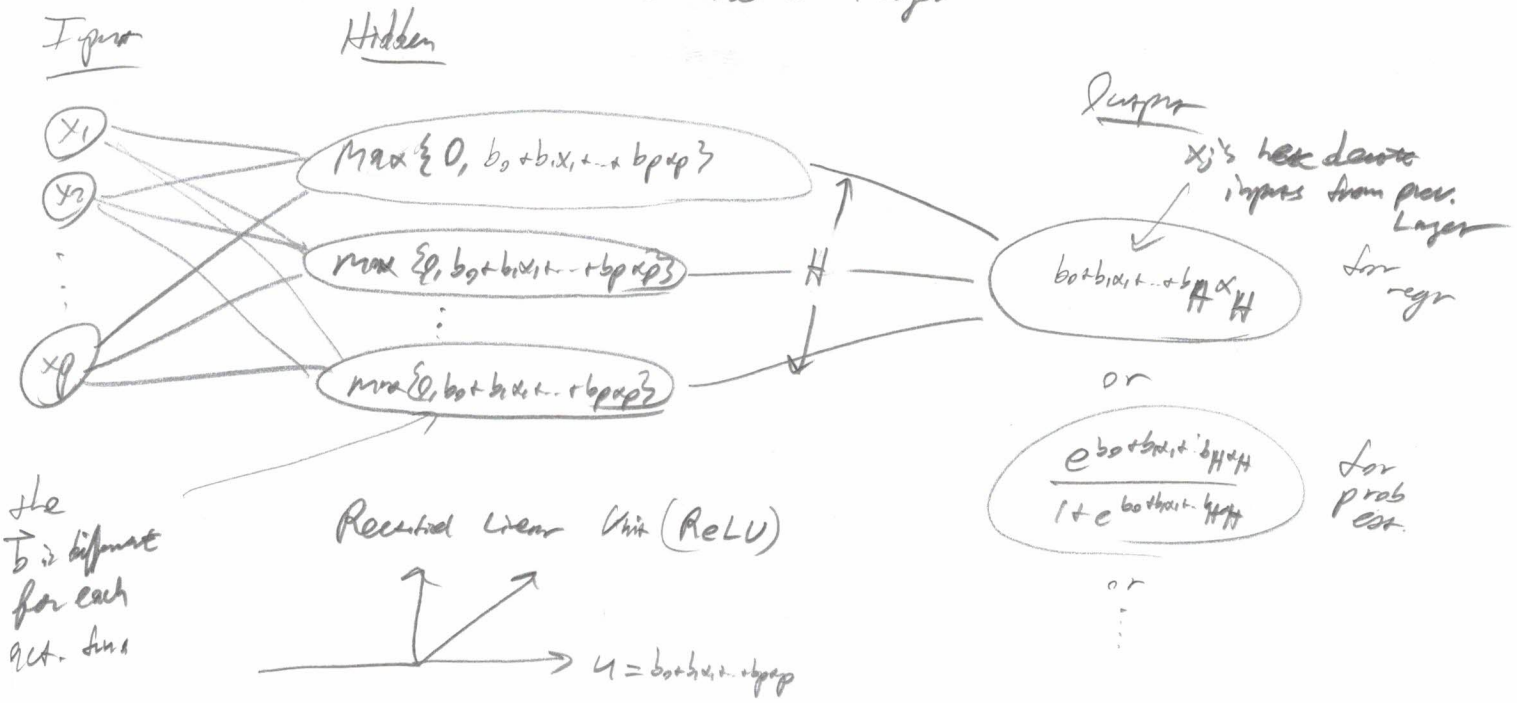
"linear act. fun"

Generally, activation functions are generalized linear functions and can be written as $q(u)$ where $u = b_0+b_1x_1+\dots+b_px_p$ (like link functions)

If so, how can NN's ever fit interactions? Just wait...

Network Topology: arrangement of Layers in a ANN.

Multi-layer Networks: has at least one hidden layer



This Hidden Layer is an "alternate representation" of the input $\vec{x}$.
It is also "fully connected" meaning each of the H act. fun's use all $x_1 \dots x_p$.

ANN's are called universal approximators i.e. they can fit any function to arbitrary precision. In 2017, they proved fully connected NN's using ReLU's with $p+1$ fully connected hidden layers and fixed depth

Width: how many hidden layers (0)
Height/depth: how many act. fun.'s per layer $\{H_1, H_2, \dots, H_L\}$

You can create a network of arbitrary width, depth and specification of activation functions. How to fit all the b's?

First specify objective function to minimize $L(\vec{y}, \hat{\vec{y}})$ or $L(\vec{y}, \hat{\vec{p}})$

Regression: SSE, Prob Est. log loss: $-\sum y_i \ln(\hat{p}_i) + (1-y_i)\ln(1-\hat{p}_i)$

Generally, NN's are tuned to do prob. est. since this obj. func. is differentiable.

② Now, pick a random starting position for all w's. E.g. $\overset{\text{iid draws from}}{\mathcal{U}(-1,1)}$

① "Feed the data forward": do all calculations for all $i=1,\dots,n$ for every activation function in all hidden layers and outer layer. Here, weights are fixed.

③ Compute the gradients of all activation functions. How?

The entire network can be expressed as a function $g(\vec{x})$ (collection of H_{D-1} functions)

even though it's complicated

$= a_D(\vec{w}_D \vec{a}_{D-1}(\vec{w}_{D-1}\ \vec{a}_{D-2}(\dots$

$a_1(\vec{w},\vec{x})\dots))$ ↑ (collection of H_{D-2} functions)

We seek to minimize $L := \sum_{i=1}^{n} L(y_i, g(\vec{x}_i))$

So we compute $\vec{\nabla}(L) := \begin{bmatrix} \frac{\partial}{\partial w_{D,1}}(L) & \dots \\ \frac{\partial}{\partial w_{D,2}}(L) & \frac{\partial}{\partial w_{D,H_D}}(L) \end{bmatrix}$

And then we set $\vec{w}_{D,t} = \vec{w}_{D,t-1} + \eta \vec{\nabla} L$ i.e. gradient descent

There, we started the last layer D first!

Now we go to the second to last layer $D-1$.

We can take the gradient here as well using the chain rule, then do gradient descent to update the weights

⋮

until the first layer. This is why it's called back-propagation (due to the chain rule)

④ Repeat steps 1,2 until convergence or max # of iterations

As D out, the Hdims get larger, the # of parameters increases
⇒ risk of overfitting. To mitigate overfitting,

(1) you can use lasso / ridge penalty at each step of back-prop. to push weights smaller

(2) dropout: randomly set weights to zero during training to force more in other weights.

(3) randomly add noise to layer inputs during training

(4) keep a select set of data and monitor its performance over iterations

⇒ there are tons of hyperparameters! D, $H_1, \ldots H_D$, a function!, objective function, η, regularization hyperparams!

So far we've explored a flexible basis set (such as trees)
⇒ deep neural nets should perform about equally to RF and xgboost. I've found it's rarely the case
I think because the # of hyperparams is so large
⇒ for "tabular" data where $\mathcal{D} = \langle X, y \rangle$

↑ a "table", a data frame, or spreadsheet

⇒ RF or xgboost is the way to go!

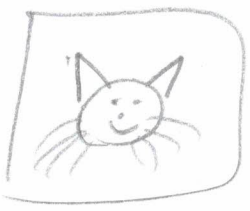
So why use NN's? Non tabular data e.g. images, text, sound, video

eg classify an image

Dog

vs.

Cat



$y=1$ $y=0$

Why do NN's do well here? Each layer is able to represent the data in a different way. If you can "represent" dog ears vs. cat ears $\Rightarrow$ simple classification!

How do we do this? We first of all need to handle image input RGB $l \times w \times 3$ arrays with pixel values $\{0, 1, \ldots, 255\}$

image color usually...