

Neural Networks

Siva Balakrishnan
Data Mining: 36-462/36-662

April 18th, 2019

Outline for Today

- ▶ Recap
- ▶ Feedforward Neural Nets Basics

Recap: Dealing with Data

Lots of considerations in analyzing data:

1. Create train, validation, test folds }
- { 2. Useful to understand your data (make plots, make conditional plots, etc.)
3. Understand the task you want to accomplish with data, and constraints (test time budget, train time budget)
4. Fix/assess outliers and missing-data ←
- { 5. Think carefully about featurization }

Recap: Possible Actions

Once you fit a model there are several possible next steps:

- ▶ Get more data
- ▶ Make more/better features
- ▶ Use a more flexible model
- ▶ Use a more regularized/less flexible model



Need to use train and validation errors, and possibly a more detailed error analysis (look at individual points) to decide what the next steps are.

Recap: Model Tuning

1. Train-validation-test splits ←

2. Featurization ←

→ 3. Quick and dirty (fit a couple of models). Understand base rates (fit naive predictors), understand what error metric you should be using, know that you can trade-off things like precision and recall in many predictors.

→ 4. Diagnose bias/variance problems (use sample-size curves, model-complexity curves, regularization curves, compare different models)

{ 5. Fix bias/variance problems (different set of fixes in each case). Iterate 2,4,5. Think carefully about how to not get bogged down by the tyranny of tuning parameters (use smaller data sets, be parsimonious in choices to try out).

6. Error analysis (diagnose points on which you are predicting poorly, are they outliers? Can you design useful features for them?).

7. Maybe you need more training data but this should be last resort.

Today: Neural Networks

The intent of today's lecture (and some of next week) is to introduce you to ideas from deep learning.

This is generally a large and exciting topic (there are several classes around campus that are entirely devoted to it).



The hope is that you'll become comfortable with some of the ideas, goals, and language, so that you can

- ▶ Recognize when it is potentially appropriate
- ▶ Discuss the basic ideas
- ▶ More easily learn about it when you need to

Getting started

Start by considering linear regression. We observe a bunch of features $x \in \mathbb{R}^p$ and outcomes $y \in \mathbb{R}$. We model them by a linear model:

$$\hat{y} = f(x) = x^T \hat{\beta}$$

We find β by solving the following optimization problem:

$$\operatorname{argmin}_{\beta} \|Y - X\beta\|_2^2 = \operatorname{argmin}_{\beta} \sum_{i=1}^n (y_i - x_i^T \beta)^2$$

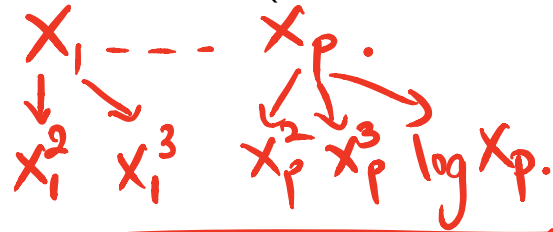
This works great when our regression function $\mathbb{E}(Y|X = x)$ is approximately linear. What if it's not?

Beyond linear regression

We've seen some approaches: we can move to a more complicated f function: random forests, bagging, SVM.

Suppose we want to improve linear regression instead. For many years, this problem was approached by engineering functions $\phi_j(x)$ to provide custom features, and then fitting a linear model (this was also how we thought of kernel SVMs):

$$y = \phi(x)^T \beta$$



In image recognition, edge detection, corner detection, keypoints (SIFT).

⌋ This is incredibly expensive! It led to improvements, but at great cost. It's also very difficult to scale to new problems.

Maybe I can do it for handwriting and voice dictation, but what about identifying facial expressions, translating languages, colorizing images, locating cats, image captioning, etc.

Beyond linear regression

Neural networks try to construct these ϕ from the data.

$$y = \underline{\phi(x; \theta)^T} \beta = \sum_{j=1}^{p'} \phi_j(x; \theta) \beta_j$$

$\theta \rightarrow$ "feature engineering"
 $\beta \rightarrow$ usual regression parameters.

If we can estimate these $\phi_j(x; \theta)$ well, then we've "replaced" feature engineering!

What should we use for $\phi_j(x; \theta)$? We like linear functions, what if we define $\phi_j(x; \theta) = x^T \theta_j$?

Why is this a bad idea?

$$x_1 \rightarrow (\theta_{11} + \theta_{21} + \theta_{31} \dots) \beta_1 \begin{bmatrix} \theta_1^T x \\ \theta_2^T x \\ \vdots \end{bmatrix} \leftarrow \text{learn linear classifier/reg. on these.}$$

$$\beta^T \begin{bmatrix} \theta_1^T x \\ \theta_2^T x \\ \vdots \end{bmatrix} = \underline{\gamma^T x}$$

Beyond linear regression

Ok, so we can't use a linear function. What's non-linear but similar to linear regression?

$$\phi_j(x; \theta) =$$


$$g(x^T \theta_j).$$

non-linear

"simple non-linearities"

Common choices for g :

Sigmoid non-linearity:

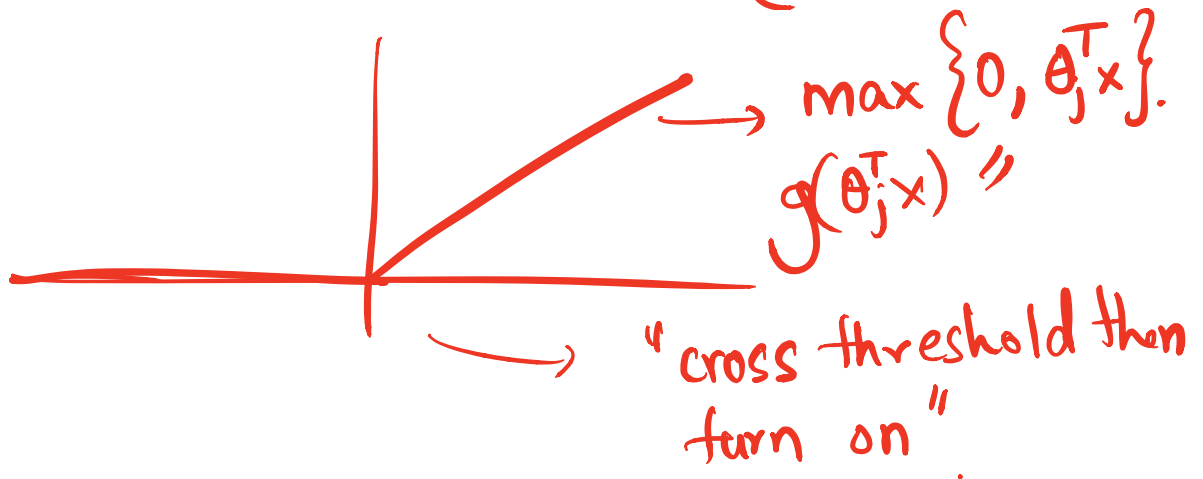
$$g(x) = \frac{1}{1 + \exp(-x)}$$


$$\sum_{j=1}^p g(x^T \theta_j) \beta_j.$$

↳ no longer linear function of x .

↳ why not $g(x)$ directly?

Rectified Linear Unit (ReLU).



So far

So now we have a model of the form

$$y = \sum_{j=1}^{p'} w_j \phi(x; \theta_j, c_j)$$

$$= \sum_{j=1}^{p'} w_j \max\{0, x^T \theta_j + c_j\}$$

ReLU non-linearity.

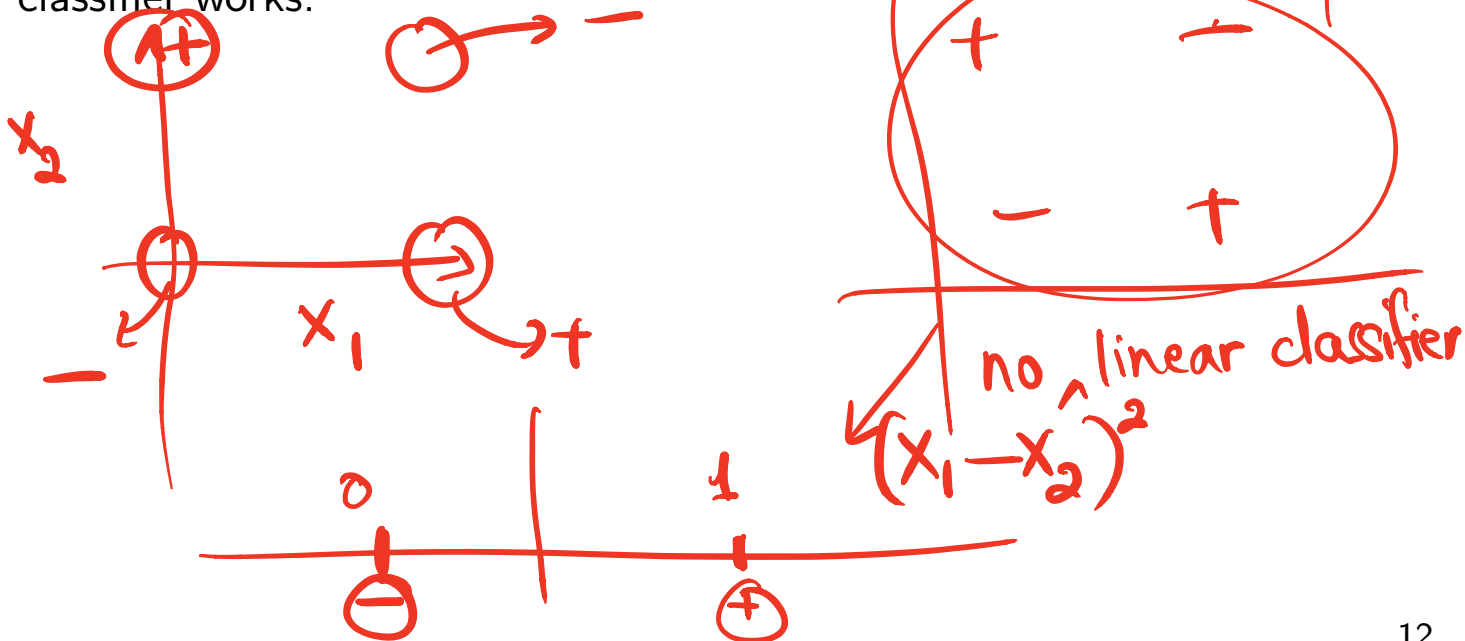
These $\max\{0, x^T \theta_j + c_j\}$ provide simple features of x to use in your final regression model.

A very simple example

Suppose we want to learn the XOR function, i.e. we have two binary features X_1, X_2 and:

$$y = \begin{cases} 0 & \text{if } (X_1 + X_2) = 0 \text{ or } 2, \\ 1 & \text{otherwise.} \end{cases}$$

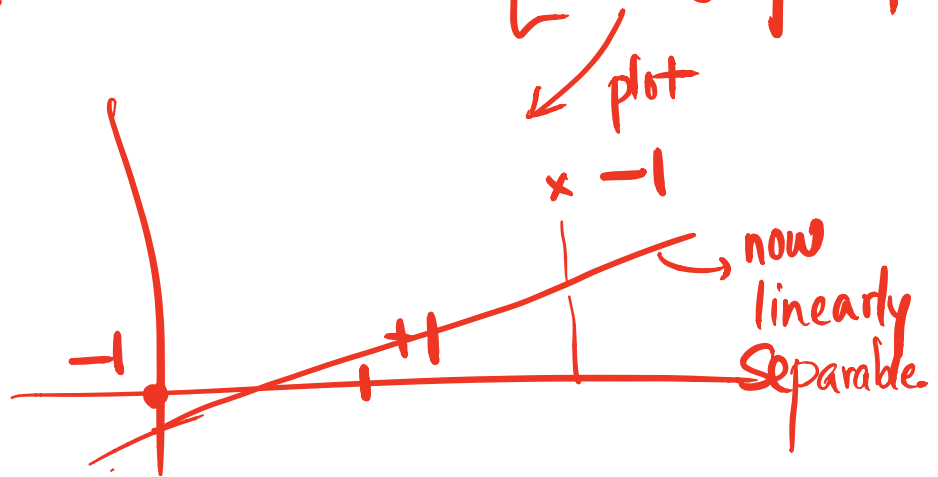
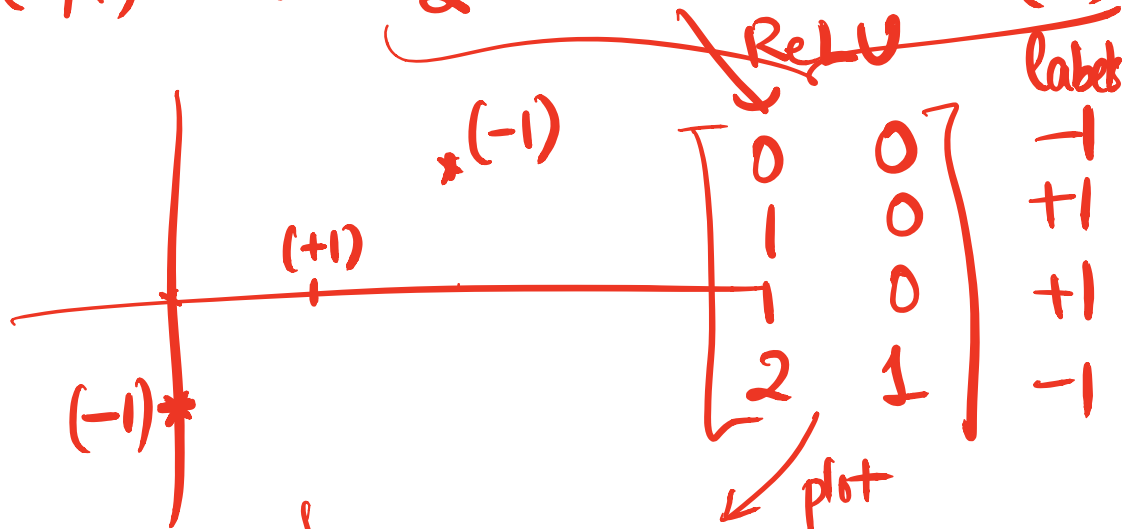
No good linear separator. Easy to engineer features so that a linear classifier works:



$$w_1 = (1, 1) \quad c_1 = 0$$

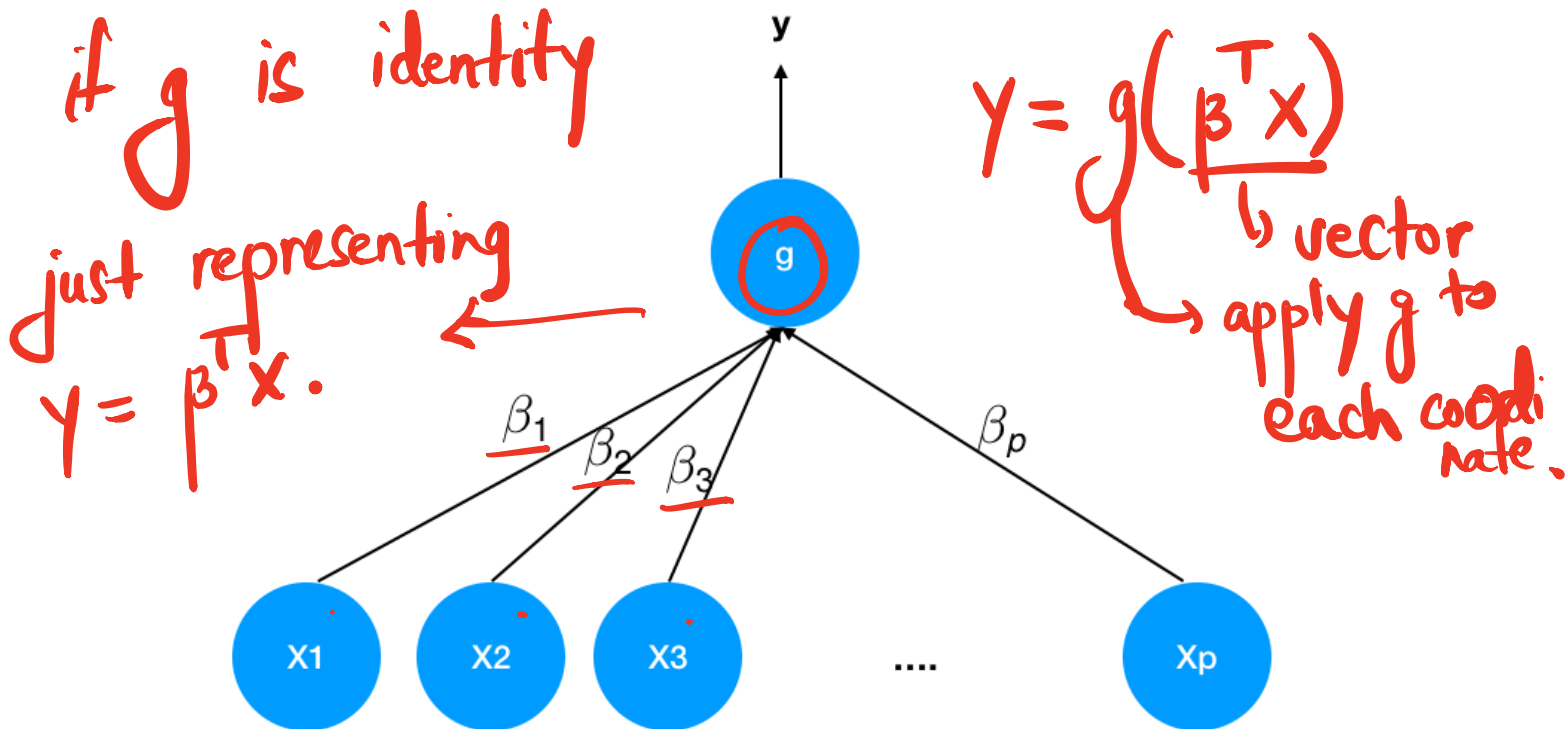
$$w_2 = (1, 1) \quad c_2 = -1$$

	$w_1^T x + c_1$	$w_2^T x + c_2$	Y
$(0, 0)$	$\rightarrow 0$	$\rightarrow -1$	$(-)$
$(1, 0)$	$\rightarrow 1$	$\rightarrow 0$	$(+)$
$(0, 1)$	$\rightarrow 1$	$\rightarrow 0$	$(+)$
$(1, 1)$	$\rightarrow 2$	$\rightarrow 1$	$(-)$



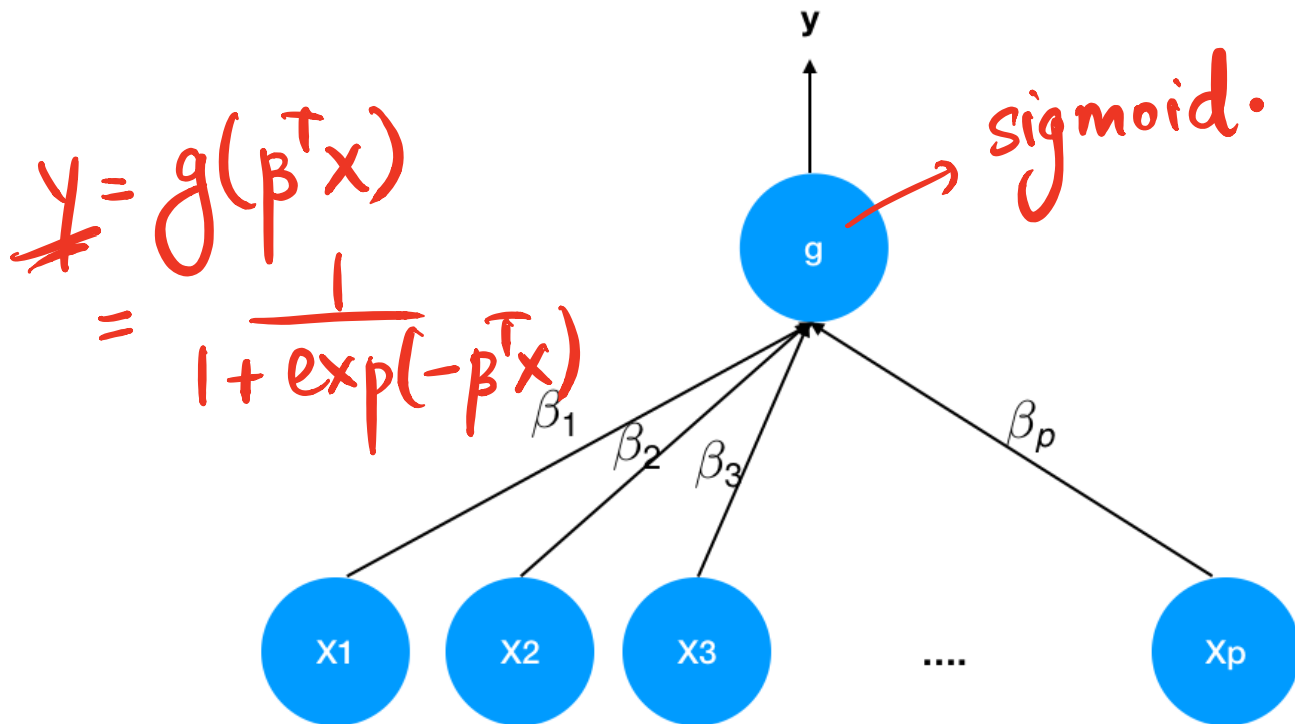
Representing Classifiers/Regressors as Networks

It will be convenient for us to think about classifiers/regressors by a network representation. So we can then make our classifiers more interesting just by playing with the network.



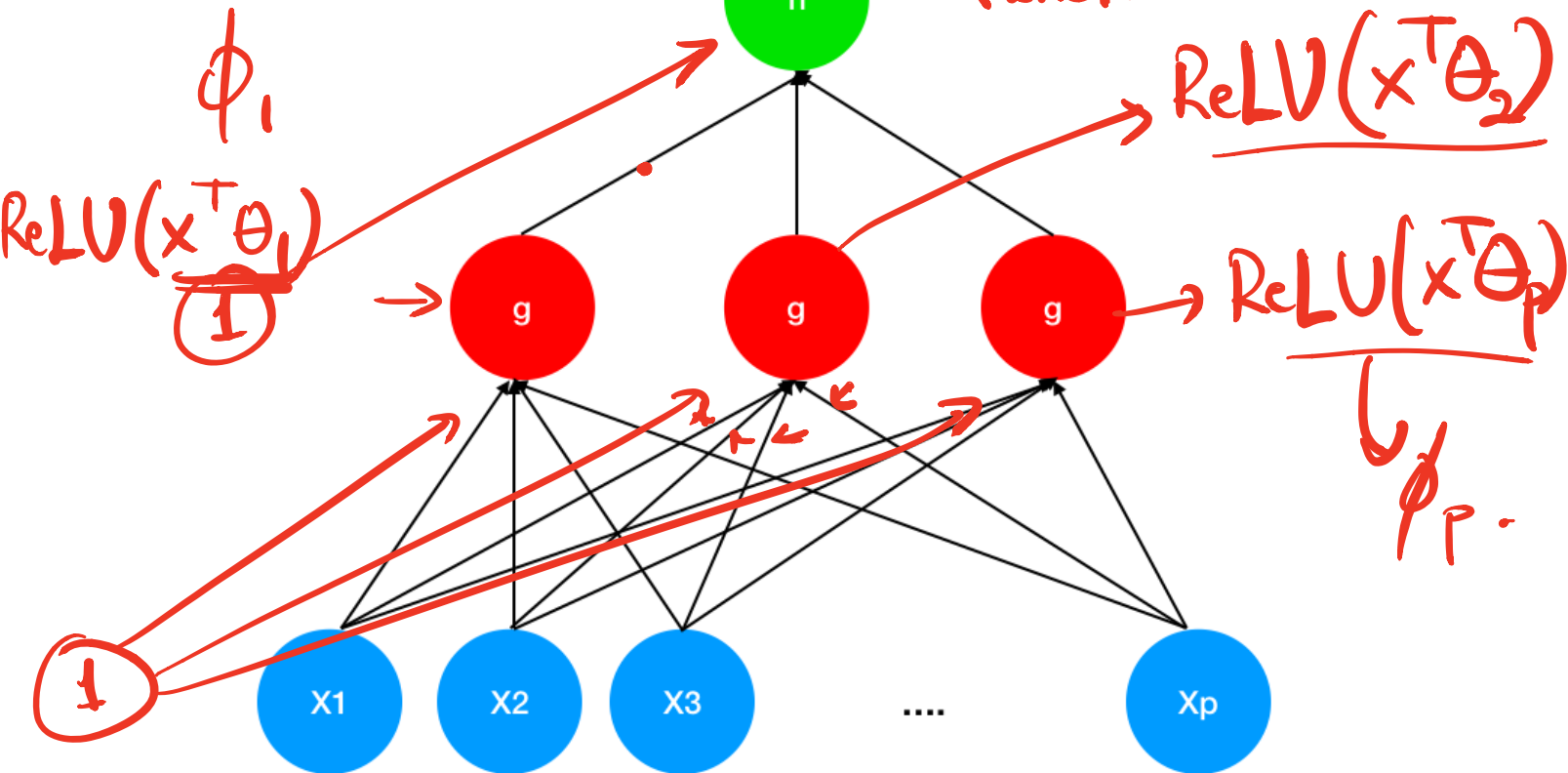
Logistic Regression as a Network

It will be convenient for us to think about classifiers/regressors by a network representation. So we can then make our classifiers more interesting just by playing with the network.



1-Hidden Layer Neural Network

$$\sum_{j=1}^p \beta_j \phi_j$$



More complexity

"deep NNs"

To get more complex features, we can just recurse! Build more layers of features, each of which is a non-linear function of a linear transformation of the previous.

"every edge is a parameter"



This is the basic idea of feed-forward neural networks or multilayer perceptrons.

↳ "learn non-linearities"
projection pursuit.

Feed-forward neural networks

one hidden layer is "enough" as long as it is (wide enough.)

This is a very flexible structure. It was shown (1989!) that feed-forward networks with a single hidden layer can approximate functions to any desired precision, given enough hidden units!

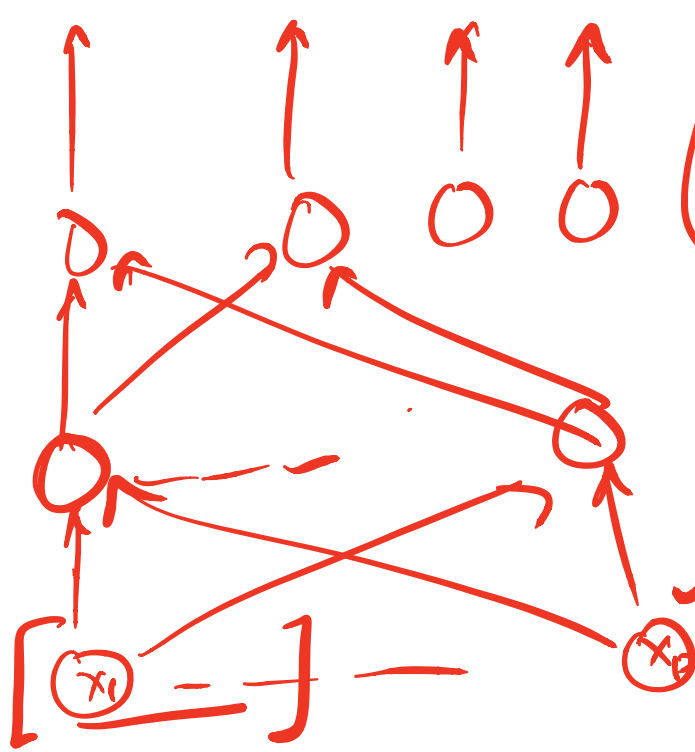
Of course, this doesn't say that you can find such a network using data...

In practice, it's hard to find such a single layer network. The abstraction of having multiple layers makes this simpler.

depth helps with optimization
with representation
⋮

could be exp. wide,

Lots of choices



- # hidden layers
- # neurons in each layer (width)
- type of non-linearity
- network need not be fully-connected
- loss function, algorithm choices.

Outputs

We can stick a bunch of final functions on top to get different outputs. Let h be the output of the final layer.

→ ▶ Continuous outcome: just use a $w^T h + b$ linear function!

→ ▶ Binary outcome: Just use logistic

$$\frac{1}{1 + e^{-w^T h}} =$$

$$\underline{\mathbb{P}(y=1 | x)}$$

→ ▶ Multiple categories: use multinomial-logistic, i.e. we produce K outputs of the form:

$$\underline{\mathbb{P}(y=i | x)} = \frac{e^{w_i^T h}}{\sum_j e^{w_j^T h}}$$

hidden layers
are doing
feature engineering

So we have a model...

We still need to consider:

1. How do we fit this model to data? Often huge number of parameters, and huge number of training examples.
2. They seem like very flexible models – how do we regularize them well?
3. How do we make the various architecture choices?

Loss Functions

The basic idea should be familiar.

- ▶ We use a loss function to measure how well we are fitting, and then try to find weights that make our loss small.

For continuous outputs, we might look $\|y - \hat{y}\|_2^2$.

For binary or multiple category outputs, we might look at the log likelihood of y_i :

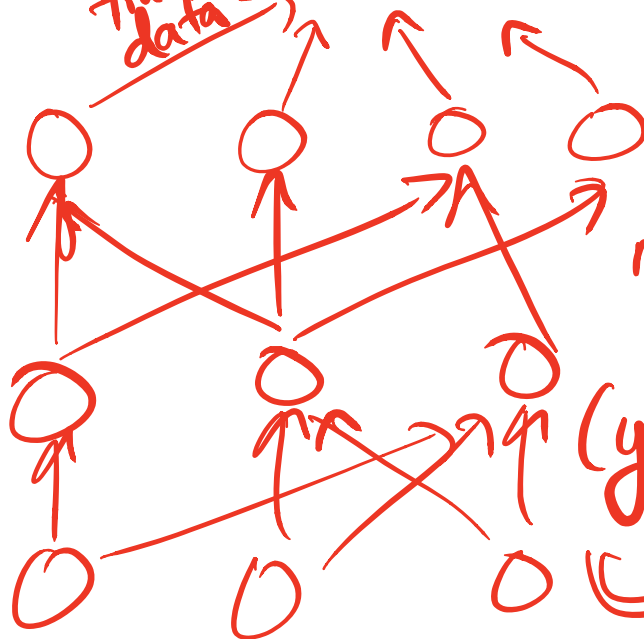
$$\text{Logistic: } \begin{cases} \log\left(\frac{1}{1+e^{-w^T h}}\right) & y_i = 1 \\ \log\left(\frac{e^{-w^T h}}{1+e^{-w^T h}}\right) & y_i = 0 \end{cases}$$

$$\text{Multinomial: } \log\left(\frac{e^{w_{y_i}^T h}}{\sum_j e^{w_j^T h}}\right)$$

We can generally design other loss functions for interesting problems.

~~y~~ ← true y from training data

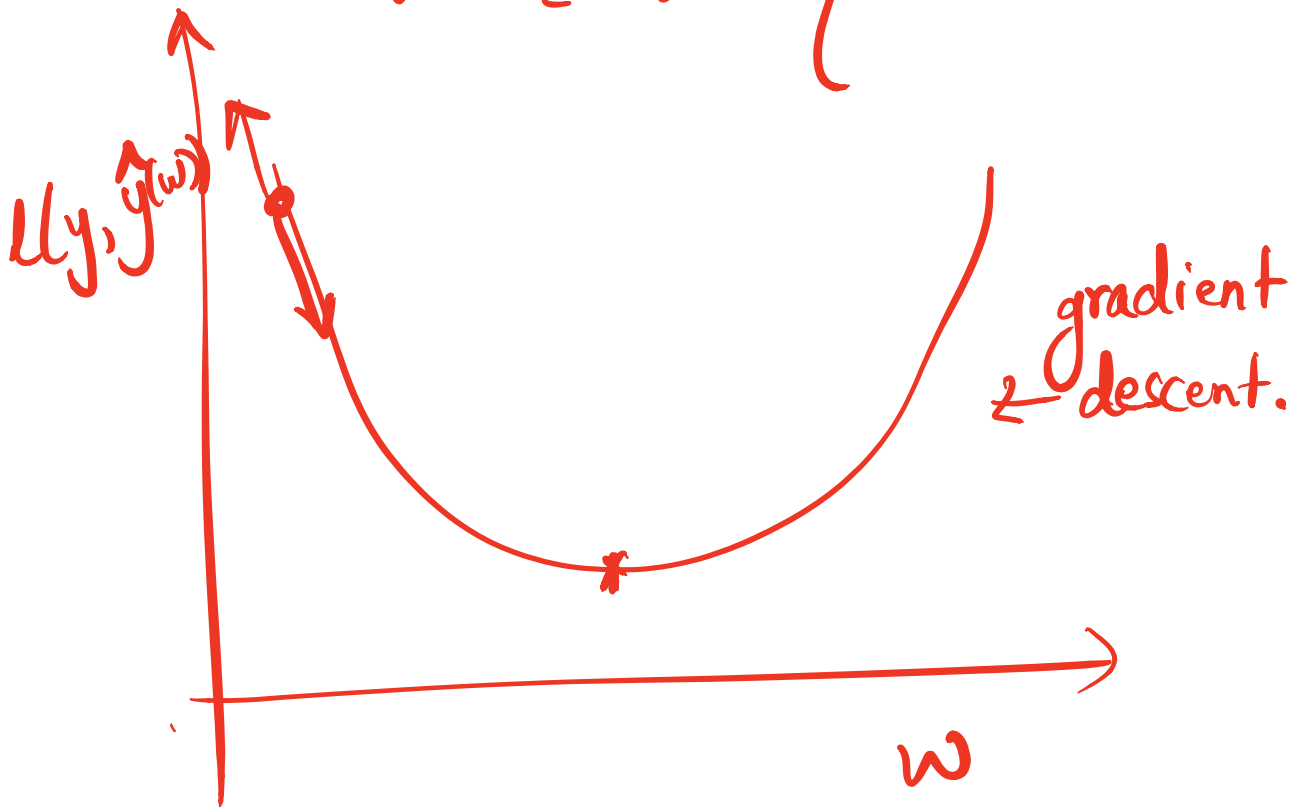
$$\hat{y} = \beta^T g(W^T g(W^T x + b))$$



$$\min \frac{1}{2} \sum_{i=1}^n (y_i - \hat{y}_i(w))^2$$

$$(y_i - \hat{y}_i(w)) \times \frac{\partial \hat{y}_i}{\partial w}$$

$$w^{t+1} \leftarrow w^t - \eta$$



Gradient descent

The neural network problem is non-convex and has many local minima and saddle points. The solution has no closed form.

The most common approach to this optimization problem is a variation of *gradient descent*.

Suppose we want to minimize $F(x)$, $x \in \mathbb{R}^p$. We could compute the gradient at the current point, $x^{(k)}$

$$\nabla F(x^{(k)}) = \left(\frac{\partial F}{\partial x_1}, \dots, \frac{\partial F}{\partial x_p} \right) \Big|_{x=x^{(k)}}$$

Gradient descent

$$\nabla F(x^{(k)}) = \left(\frac{\partial F}{\partial x_1}, \dots, \frac{\partial F}{\partial x_p} \right) \Big|_{x=x^{(k)}}$$

The gradient points in the local direction of steepest increase. To walk down hill, we would take a step in the opposite direction:

$$x^{(k+1)} = x^{(k)} - \eta \nabla F(x^{(k)})$$

The step size, η , is also called the *learning rate*. Picking a good value of η is important and there are a variety of approaches.

Gradient descent

Of course, actually computing the gradient is expensive! Our F is the loss, possibly (ignoring regularization):

$$\sum_{i=1}^n (y_i - \hat{y}(x_i))^2$$

The first sum might have 10,000,000 terms! The \hat{y} function might have millions of parameters!

We will discuss a bit more about tricks for optimizing and regularizing neural networks in the next lecture but for now let us just understand one simple but powerful idea.

Computing gradients

Only takeaway: Backpropagation

Are you looking forward to computing the gradient of

$$\sum_{i=1}^n \|y - \hat{y}\|^2,$$

computes gradients with 1 pass through network.

with respect to each of the parameters? For instance, with a 2-hidden layer network we would have to find the derivative of:

$$\begin{aligned}\hat{y}(x) &= w^T g \left(W^{(2)} g \left(W^{(1)} x + b_1 \right) + b_2 \right) + b \\ &= w^T \max \left\{ 0, W^{(2)} \max \left\{ 0, W^{(1)} x + b_1 \right\} + b_2 \right\} + b\end{aligned}$$

with respect to all the elements of $W^{(1)}$ and $W^{(2)}$?

Backpropagation

We can write out all of our computations as a graph (because the network is nice) and then work our way backwards using the chain rule.

$$\begin{aligned}\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \\ &= h'(y)g'(x)f'(w)\end{aligned}$$

Backpropagation in an Example

Where are we so far?

- ▶ We introduced neural networks as a non-linear model for classification/regression.
 - ▶ Discussed their motivation (flexible models, try to automate feature engineering).
- ▶ How to understand the network representation of a predictor, and how to construct deep neural networks.
- ▶ Loss functions for measuring how well we are doing.
- ▶ Backpropagation for trying to fit the network parameters to data.