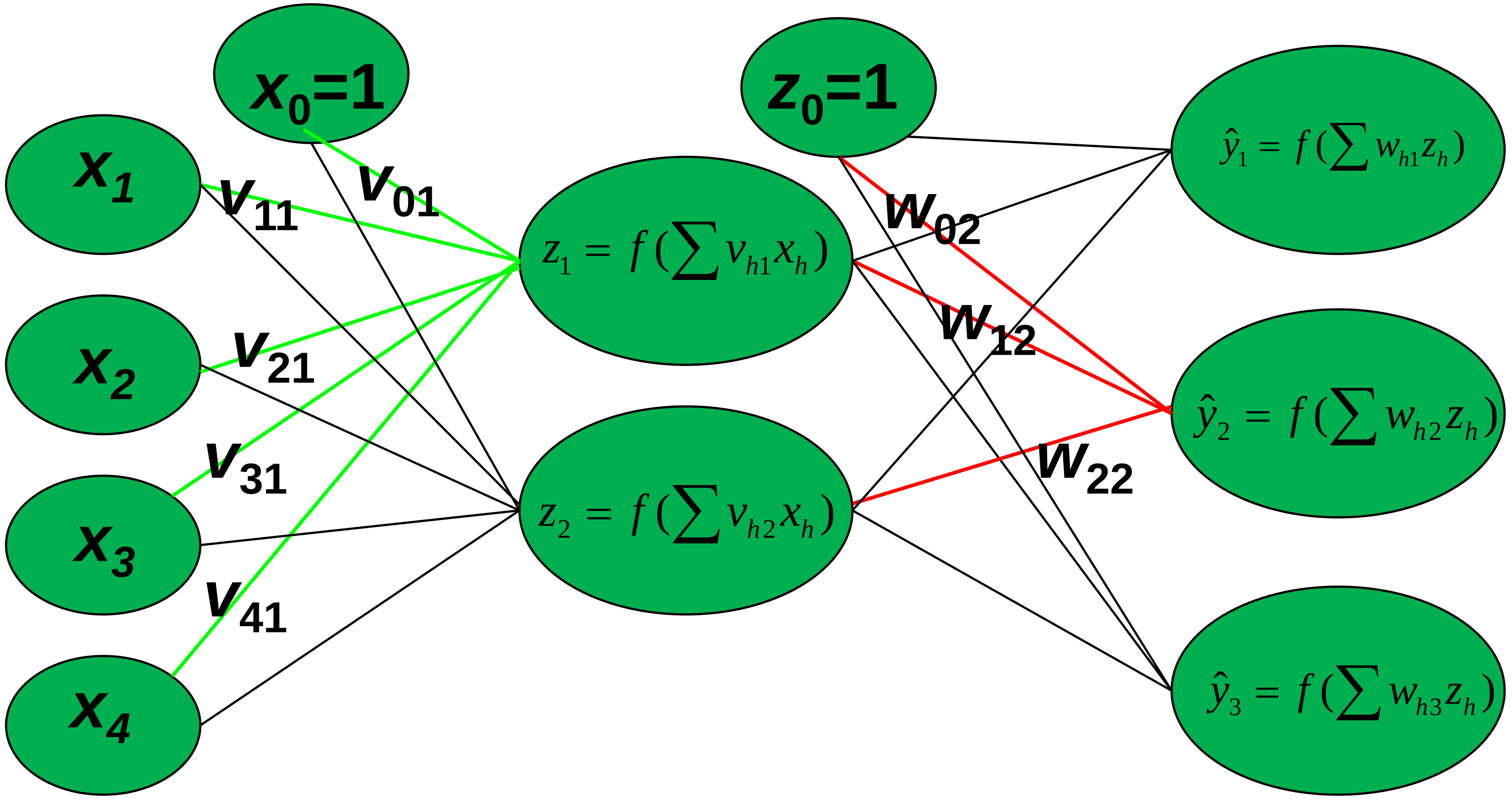
Incorporating Categorical Variables

- In an earlier example we generated a 0-1 **dummy** variable to reflect PAX/Key West
- With *k* categories we normally generate *k* 0-1 variables (not the k – 1 usually seen in regression)
- Each observation produces one 1 and (*k* – 1) 0's, and each of those things gets its own node
- …leading to the neural network phrase "**one-hot coding**"
- What about scaling?

Scaling

- Scaling numeric inputs will almost always be an issue in neural networks, for a couple of reasons:
   1. Starting values for the weights are often chosen at random so scale matters there
   2. In the sigmoidal activation, large or small values of *x* land on flat spots, where changing the weight does not affect the value of the output. making it hard to learn
   3. Variables on very different scales produce error surfaces that are like pointed elliptical bowls, making it harder to find the optimum

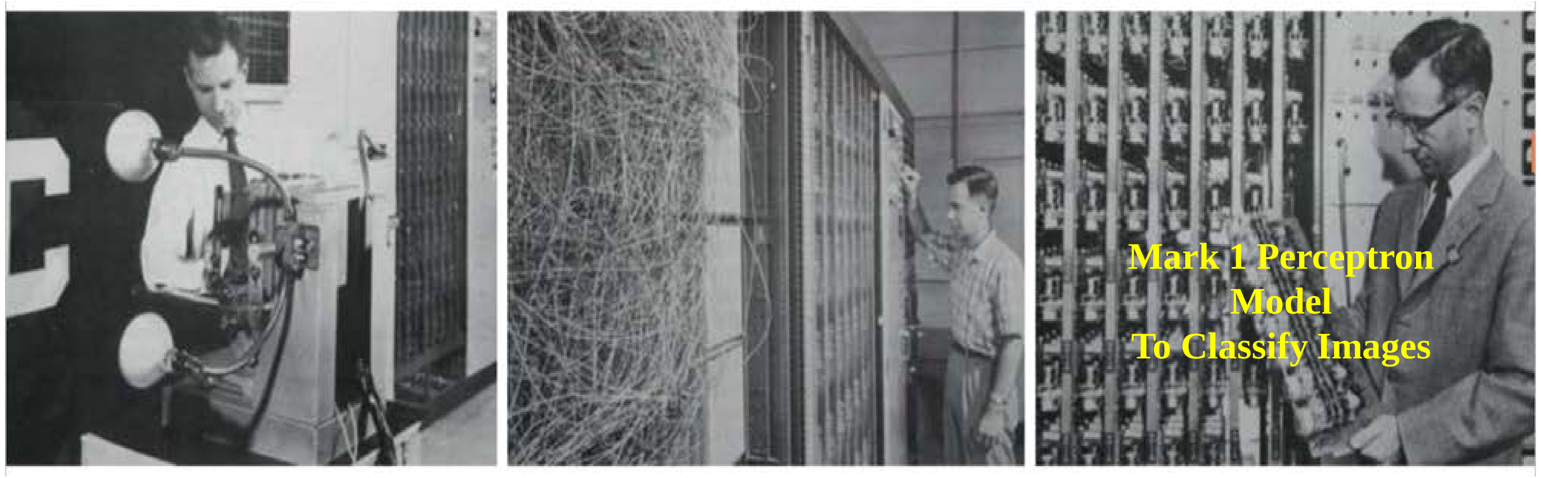
Back to Multi-Layer Networks


$x_0=1$
$z_0=1$
x_1
x_2
x_3
x_4
v_{01}
v_{11}
v_{21}
v_{31}
v_{41}
w_{02}
w_{12}
w_{22}
$z_1 = f(\sum v_{h1}x_h)$
$z_2 = f(\sum v_{h2}x_h)$
$\hat{y}_1 = f(\sum w_{h1}z_h)$
$\hat{y}_2 = f(\sum w_{h2}z_h)$
$\hat{y}_3 = f(\sum w_{h3}z_h)$

…where $f(t)$ is the activation function

But How Do You Train The Weights?

- Rosenblatt (1962) introduced "**online**" updating i.e.
  - start with an initial set of weight values;
  - predict a ***y*** for one or a few observations (examples);
  - see how well you did and update the parameters
  - Hard because threshold function was discontinuous



Digitize images with 20 x 20 grid of photocells (pixels). These are the input nodes.

Wires are the arcs connecting the input nodes to the output nodes.

The weights are encoded in potentiometers, shown here. They are updated using electric motors.

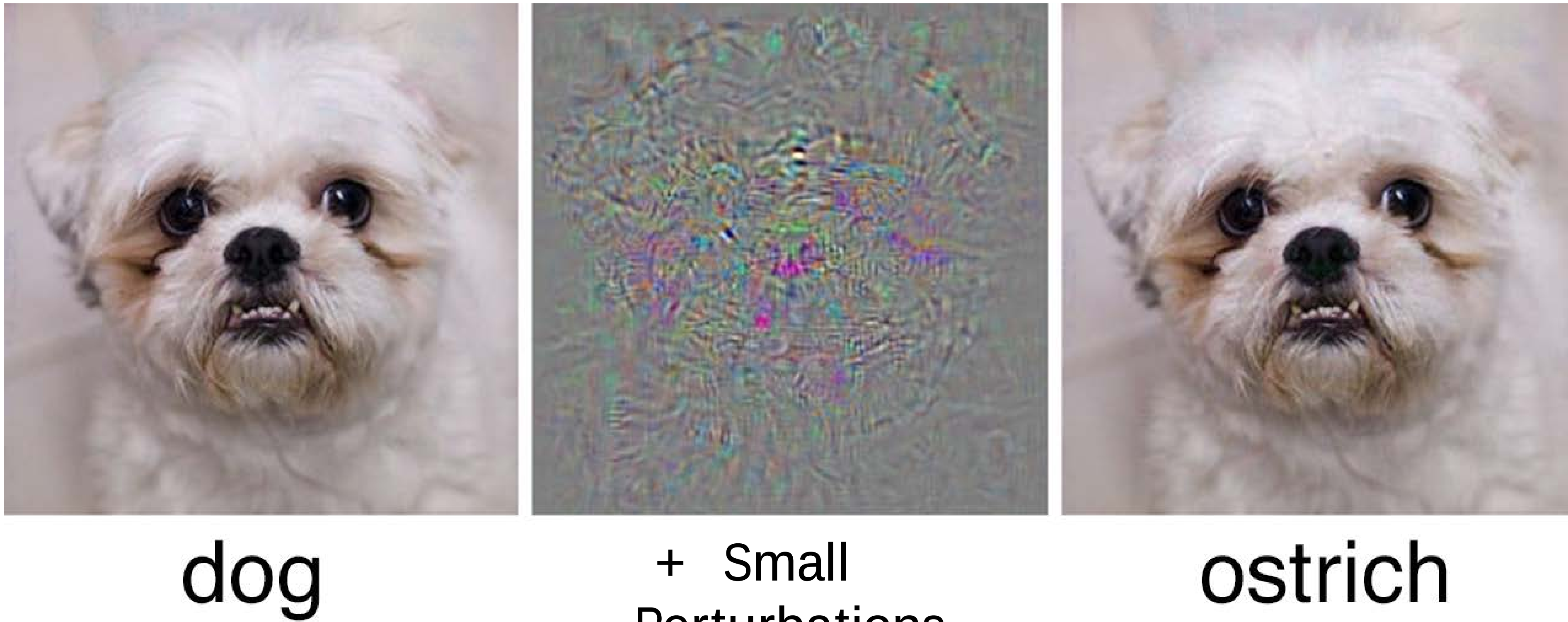
In the late 60’s, work on all neural networks slowed until the 1980’s

Problem #1: Exaggerated claims were made about perceptron networks resulting in bad PR.

Problem #2: The perceptron only gives binary output and hard to train because the error function is a discontinuous function of the weights.

Problem #3: The Minsky and Papert (1969) text showed that single layer perceptron models can not classify well if the inputs are not linearly separable. Their work was mis-cited to apply to multi-layer perceptron models too.

Problem #1: Modern day version. What do NN's "see"? And how to fool them.



+ Small Perturbations

Christian Szegedy (Google) et al.(2014)

Modern Artificial Neural Networks

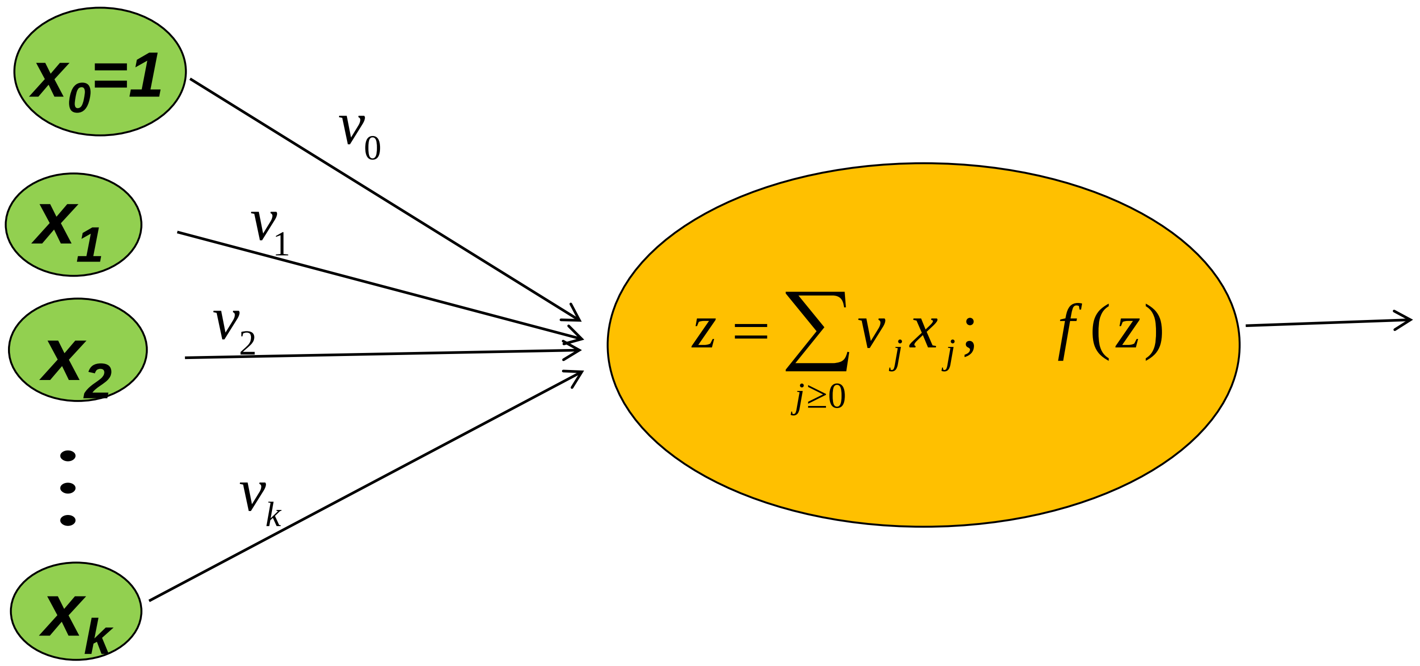
Problem #1: Exaggerated claims were made about perceptron networks resulting in bad PR.

Problem #2: The perceptron only gives binary output and is hard to train, because the error function is a discontinuous function of the weights (and the perceptron updating doesn’t always do a great job at updating weights)

Problem #3: The Minsky and Papert (1969) text showed that single layer perceptron models can not classify well if the inputs are not linearly separable. Their work was mis-cited to apply to multi-layer perceptron models too.

Rumelhardt and McClelland (1986) introduce continuous activation functions

This allows numeric outputs for numeric targets



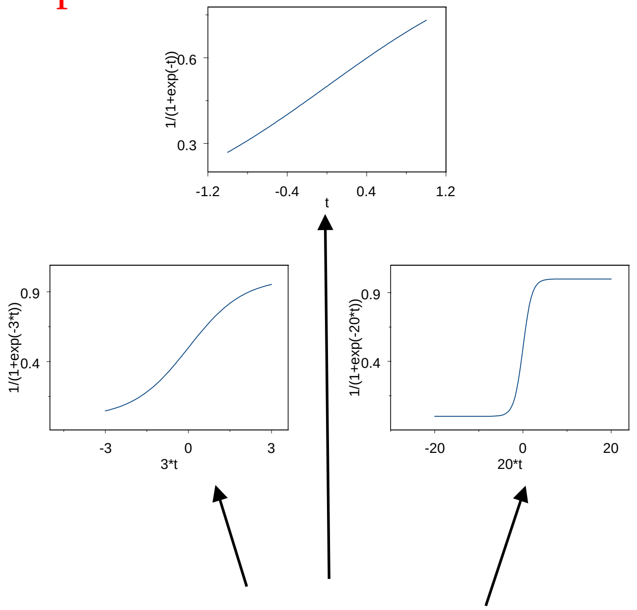
Popular Activation Functions

Threshold $$f(z) = \begin{cases} 1 & z > 0 \\ 0 & z \leq 0 \end{cases}$$ Binary output

Linear $$f(z) = z$$ Numeric output

Logistic* $$f(z) = \frac{e^z}{1+e^z} = \frac{1}{e^{-z}+1}$$

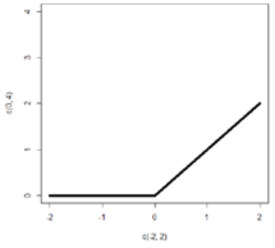
Hyperbolic Tangent $$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



Bounded Numeric output

Rectified Linear** $$f(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$

Numeric positive output

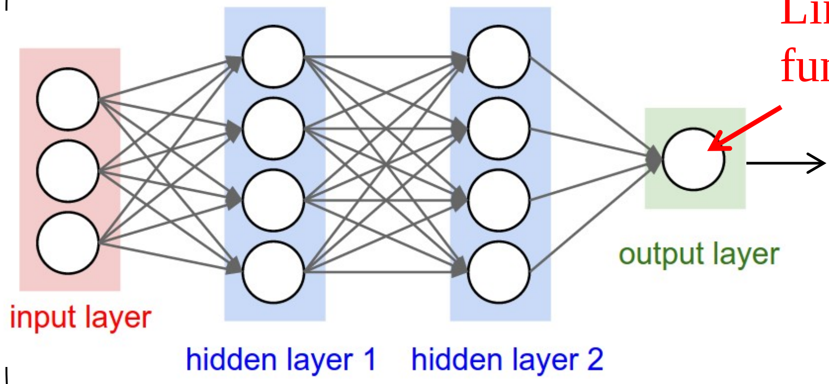


Another benefit of continuous activation functions is:

- The output is a continuous function of all or the weights in the network
- The error function is a continuous function of weights if the loss function is a continuous function of the output (e.g. squared error loss or Bernoulli deviance loss)
- Now, optimization is easier.

Error Function for Single Target

For **numeric** targets: $E = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$



Linear activation functions

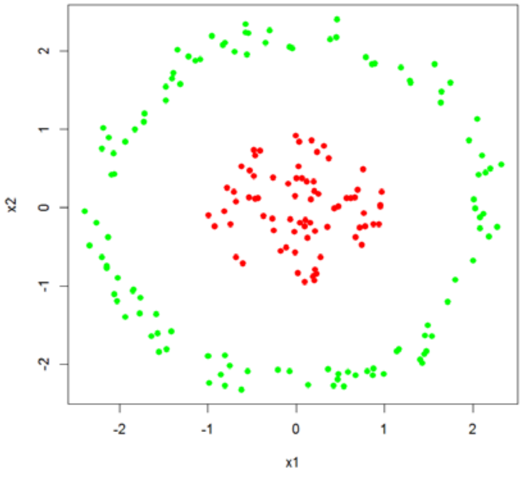
$\longrightarrow \hat{y}$ $\dashrightarrow$ $\hat{p} = \frac{\exp(\hat{y})}{1+\exp(\hat{y})}$

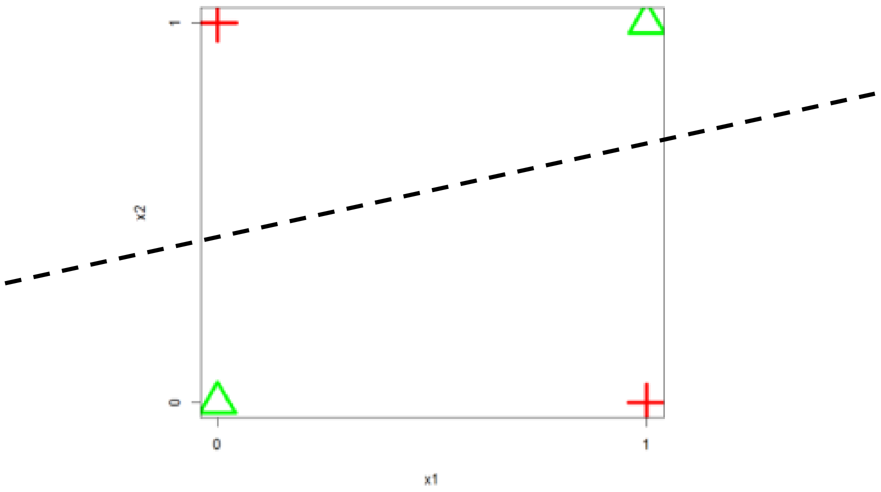
For Categorical Targets with two levels:

$$E = -\frac{1}{n}\sum_{i=1}^{n} y_i \ln\left(\hat{p}_i\right) + (1 - y_i)\ln\left(1 - \hat{p}_i\right)$$

The Minsky – Papert problem

Problem #3: Minsky and Papert (1969) showed that single layer perceptron models cannot classify well if the inputs are not linearly separable.



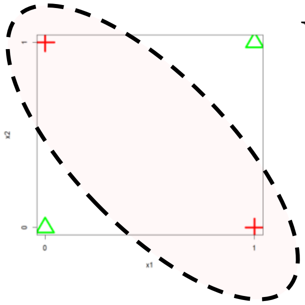


Linear Boundary: $\upsilon_0 x_0 + \upsilon_1 x_1 + \upsilon_2 x_2 = 0$

Yes, there is a problem, but not that!

- The problem is that for the simple NN to be useful, input variables must be crafted, **by hand,** carefully.

$$\text{Linear Boundary: } \upsilon_0 x_0 + \upsilon_1 x_1 + \upsilon_2 x_2 + \upsilon_3 x_3 + \upsilon_4 x_4 + \upsilon_5 x_5 = 0$$

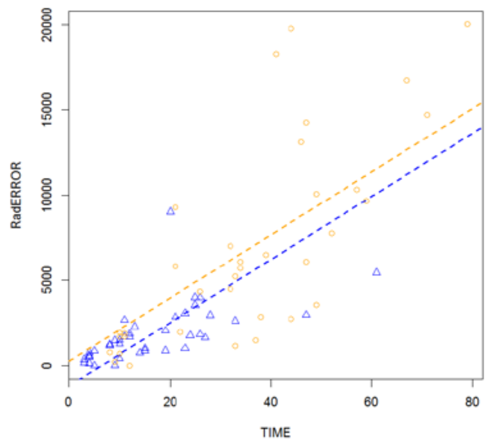


$$\text{where} \quad x_3 = x_1^2, \quad x_4 = x_2^2, \quad x_5 = x_1 x_2$$

- For a long time, NN were fit by subject matter experts who constructed features tailored to the application e.g. computer vision, voice recognition, natural text processing.
- This is also why SVMs made such a splash in the late 90's – they "automatically" add features

TACNAV Example continued:

> lm(RadERROR~SITE+TIME,data=tacnav.data)
Call:

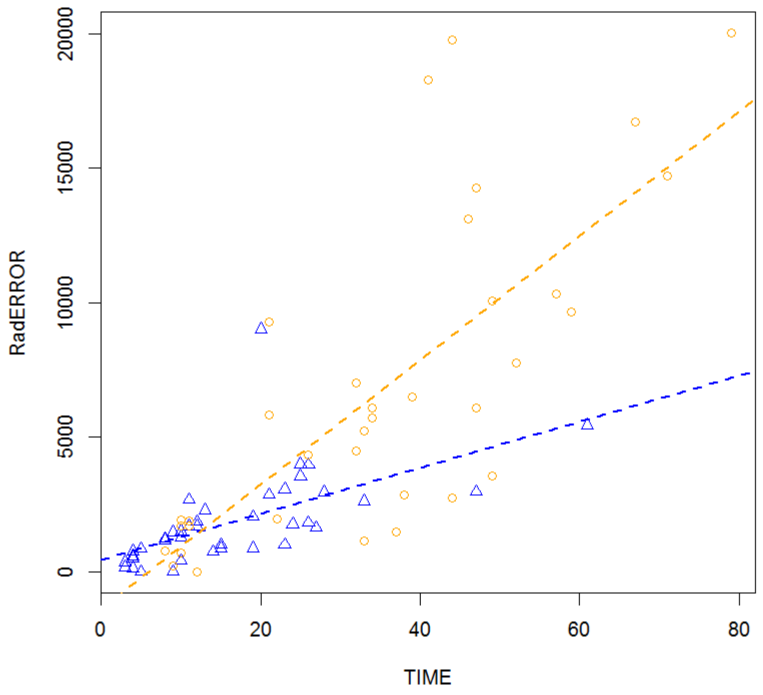


If we expand the feature space by introducing a third node…

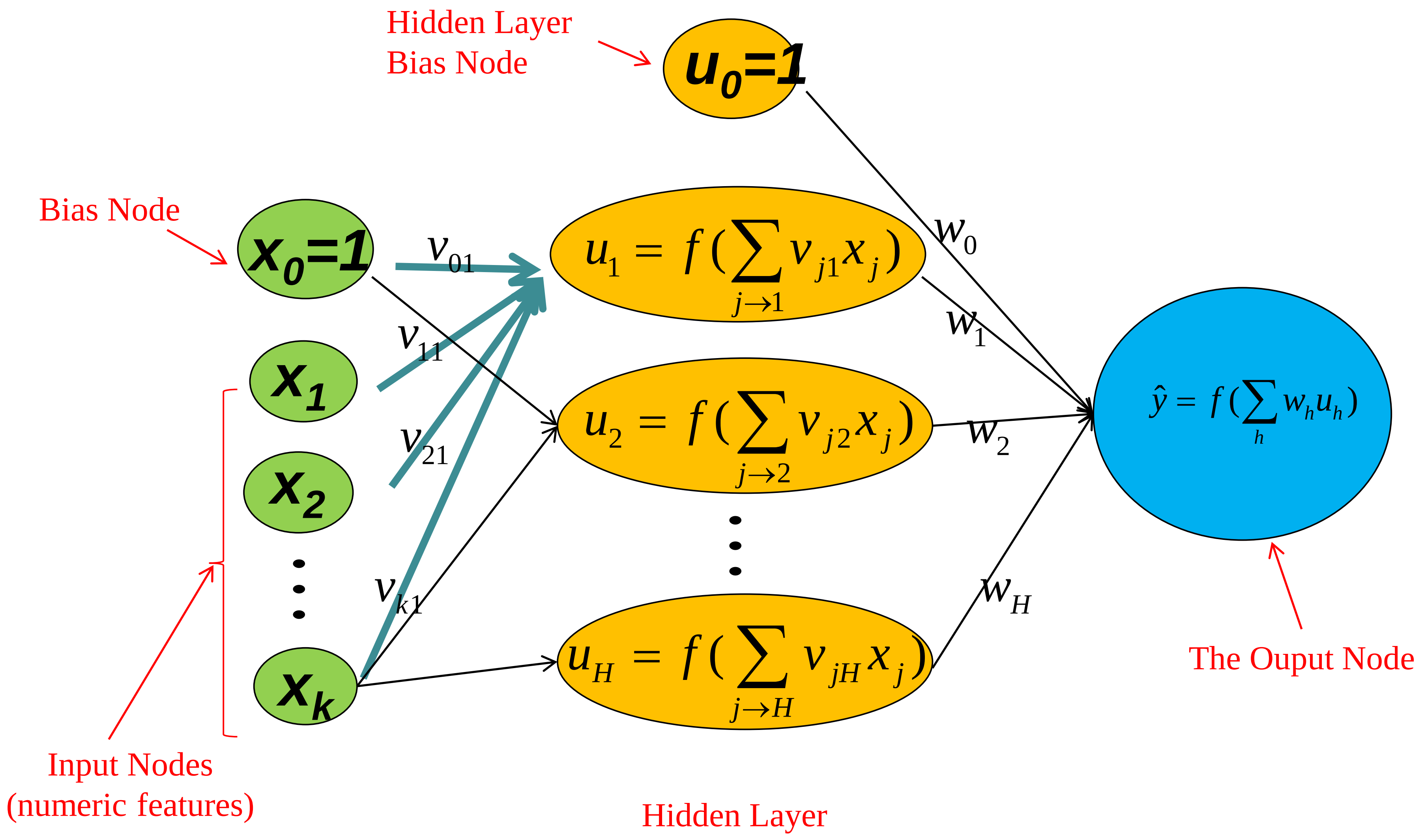
By defining new input features, a NN can handle any model a glm can, but….

```
summary(nnet(RadERROR~SITE+TIME+I(SITE*TIME),
     data=tacnav.data,  skip=T,   size=0, linout=T))

a 3-0-1 network with 4 weights
options were - skip-layer connections  linear output
units
    b->o    i1->o    i2->o    i3->o
  446.43 -1830.28    85.51   146.26
```



So what about the hidden layer?

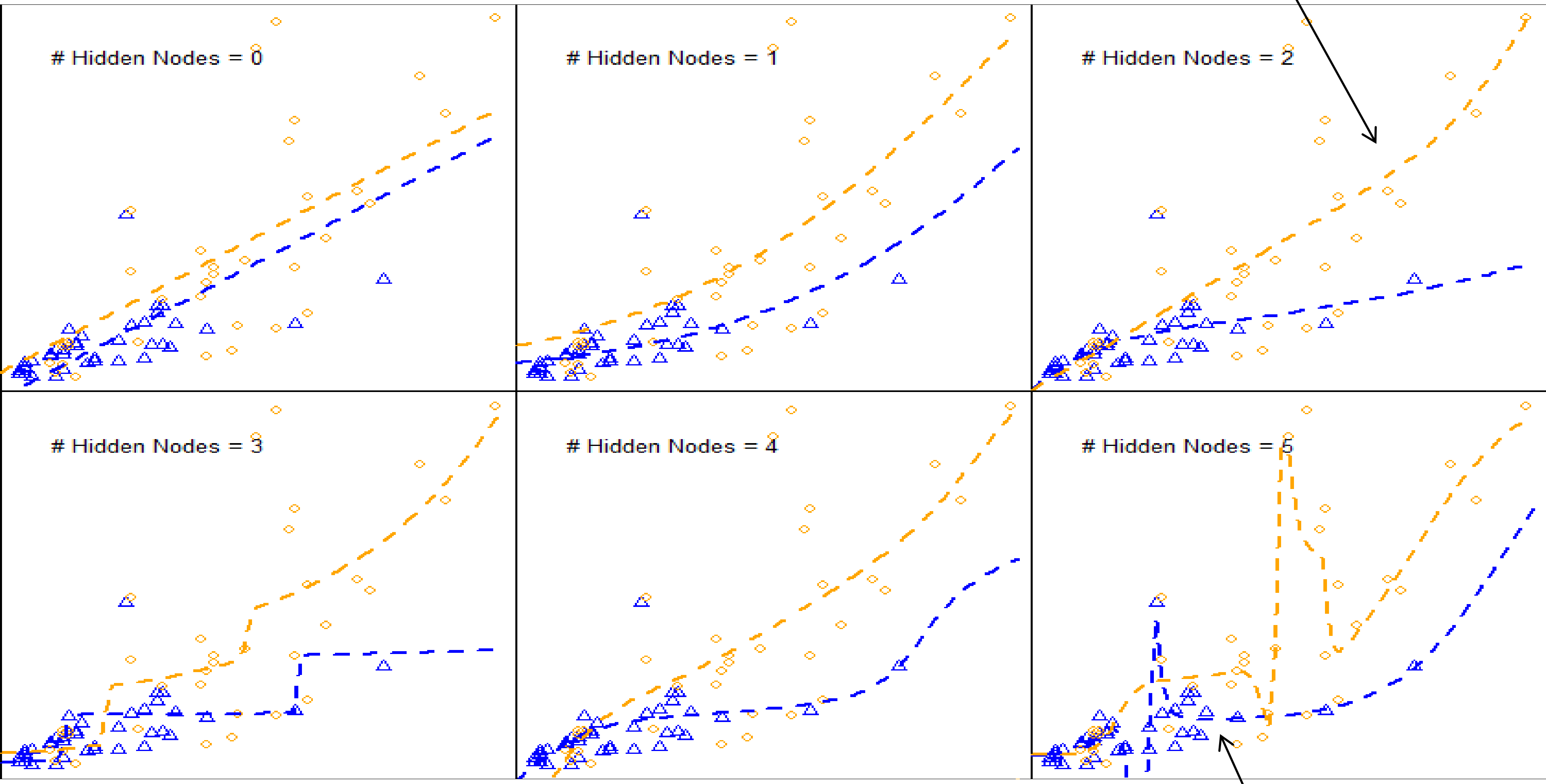


Hidden Layer Bias Node
$u_0=1$
Bias Node
$x_0=1$
v_{01}
v_{11}
v_{21}
v_{k1}
x_1
x_2
x_k
Input Nodes (numeric features)
$u_1 = f(\sum_{j\to 1} v_{j1}x_j)$
$u_2 = f(\sum_{j\to 2} v_{j2}x_j)$
$u_H = f(\sum_{j\to H} v_{jH}x_j)$
Hidden Layer
w_0
w_1
w_2
w_H
$\hat{y} = f(\sum_h w_h u_h)$
The Ouput Node

In the early 90's there was the sense that shallow NN sufficed.

- As the number of nodes increase in a single hidden layer, the neural network can approximate arbitrarily well any piecewise continuous function. Hornik et al (1989)
- Really? Let's try this out.
- As we saw the relationship between y (RadError) and x_2 (Time) is not really linear and the relationship is different for the two levels of the categorical variable x_1.
- Train neural networks with:
  - just two input units
  - 0,1,…,5 hidden nodes in the hidden layer, with a logistic activation function.
  - one output node with a linear activation function
  - with SSE error function..

Not Bad: The hidden layer “learned” the interaction and some non-linearity



Over -fitting

And it worked for ALVINN too.

- ALVINN **1989,** an early self-driving HMMV.

  https://www.theverge.com/2016/11/27/13752344/alvinn-self-driving-car-1989-cmu-navlab

(This link is no longer available)

- Input, digitized 30 x 32 pixel, grey scale image . Output direction of steering wheel.

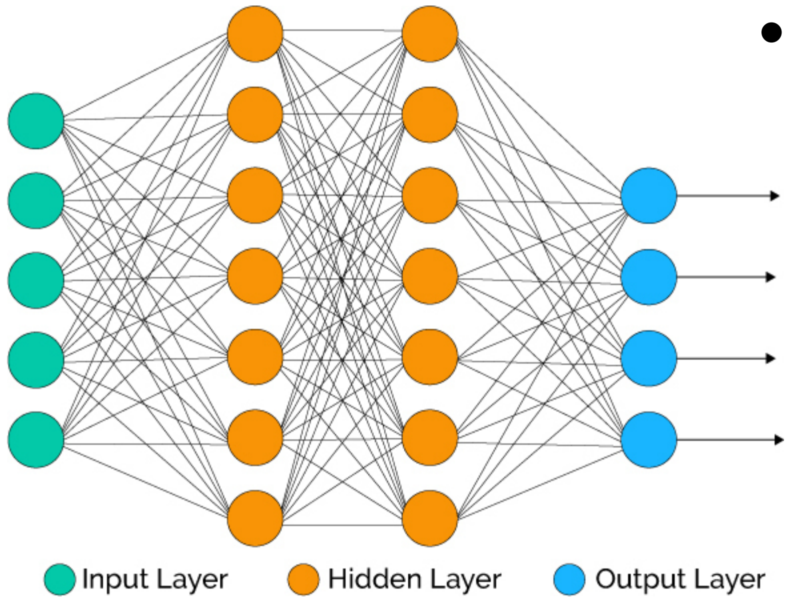
- Simple 1 - hidden layer fully connected with NN, numeric target

Neural Networks fell out of favor again because...

- Shallow NN are very dependent on how the input features are defined…SMEs spent months crafting features for NNs.
- E.g. NN couldn't identify patterns such as translations of images unless important features of the pattern are hard coded as features…
- At this time random forests and gradient boosted models far out-performed NN in the major data mining competitions.
- Need a model which can learn new features by reshaping the original inputs…

Multiple Layers

- Neural Networks almost always consist of two or more **layers** of perceptron units

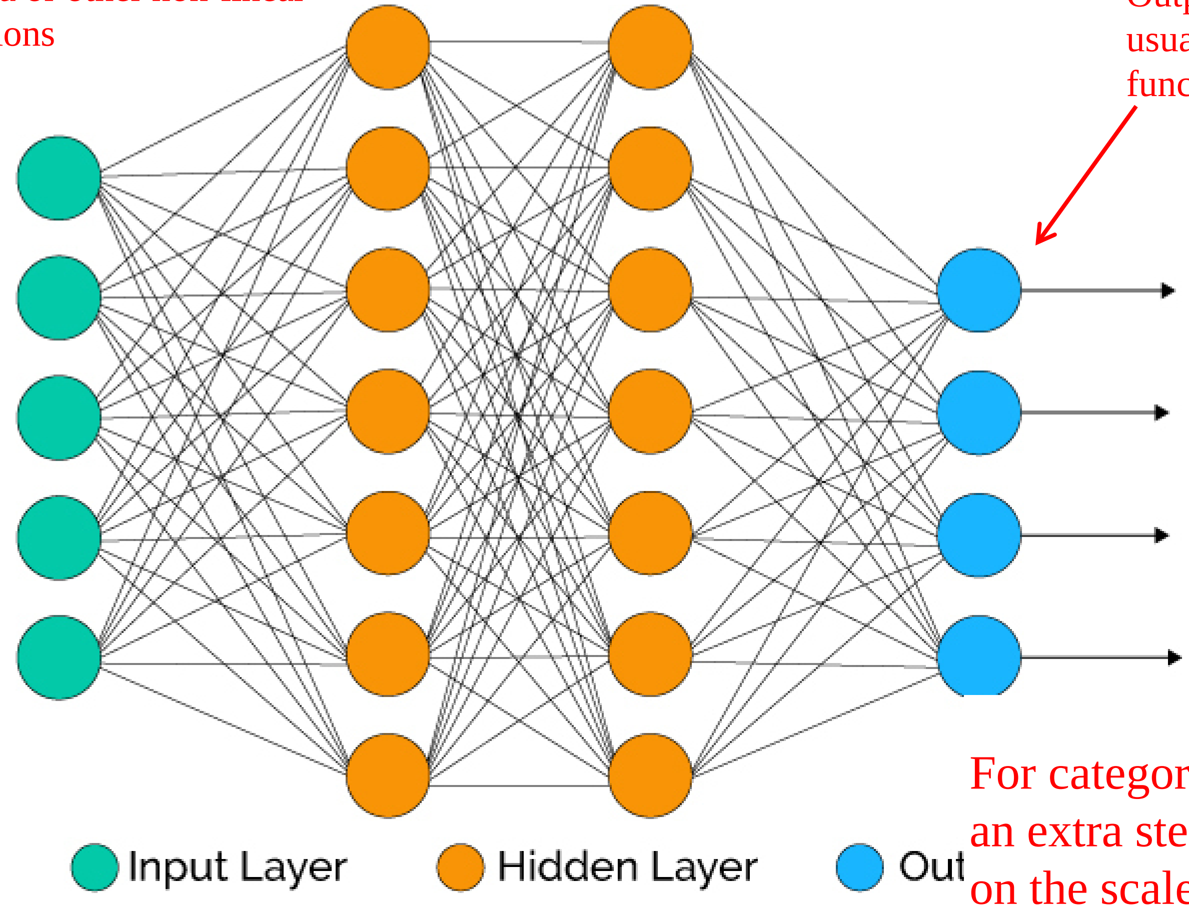


- This makes them able to emulate input signals more complicated that what a simple linear regression can deal with

Nodes in Hidden Layers usually have the same activation function

Hidden nodes
Often have ReLu or other non-linear activation functions

Output nodes
usually linear activation functions

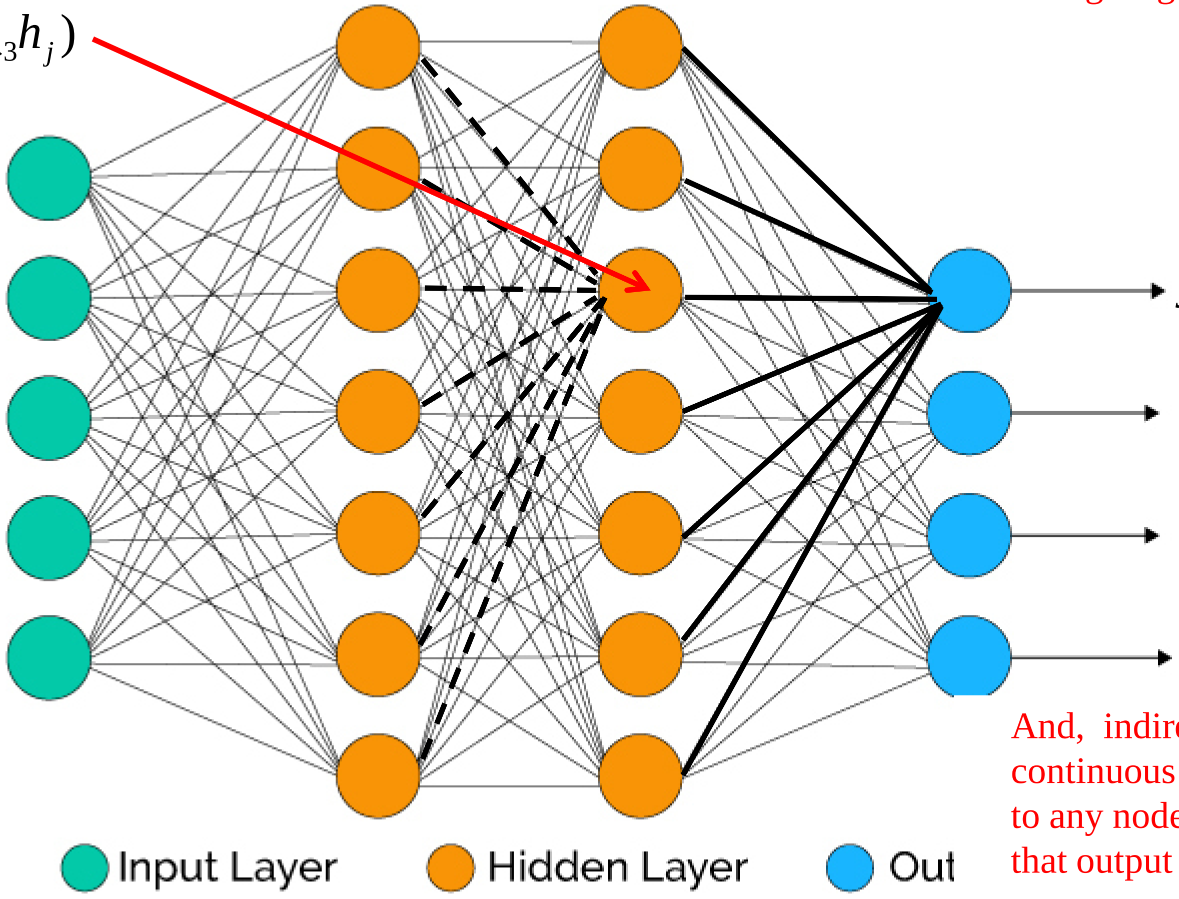


For categorical targets, there is an extra step to put the output on the scale of [0,1].

Outputs are continuous functions of weights

$o_3 = f(\sum_j \upsilon_{j\to 3} h_j)$

The output is a continuous function of the weights going into it. E.g.



$\hat{y}_1 = \sum_k w_{k\to 1} o_k$

It is also a continuous function of the outputs of the nodes feeding into it.

And, indirectly, the output is a continuous function of weights going to any node that ultimately feed into that output node. E.g.

$$\hat{y}_1 = \sum_k w_{k\to 1} o_k = \sum_k w_{k\to 1} f(\sum_j \upsilon_{j\to k} h_j)$$

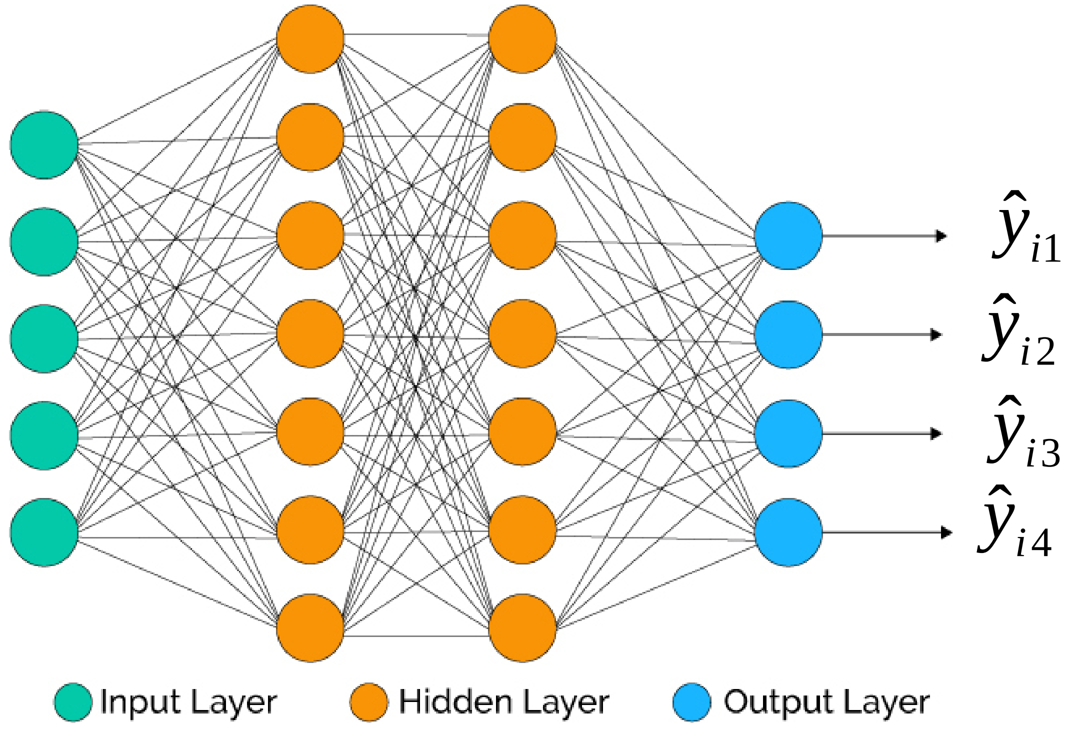
How many layers / nodes?

- The input layer is set by the input data, and the output layer by the type of problem
- How many hidden layers, and how big should they be? These are hard problems to answer
  - Of course we'll try to answer these in the course
- Today's software makes it possible to have "many" layers (even dozens), some of which are "large" (hundreds of nodes)
- The hip term "**deep learning**" just means "a neural network with lots of layers

Error Function: Multiple Numeric Targets

$$E = \frac{1}{2n}\sum_{i=1}^{n}\left[\sum_{m=1}^{M}(y_{im} - \hat{y}_{im})^2\right]$$

This is equivalent to measuring the squared Euclidean distance between the multivariate target and multivariate output

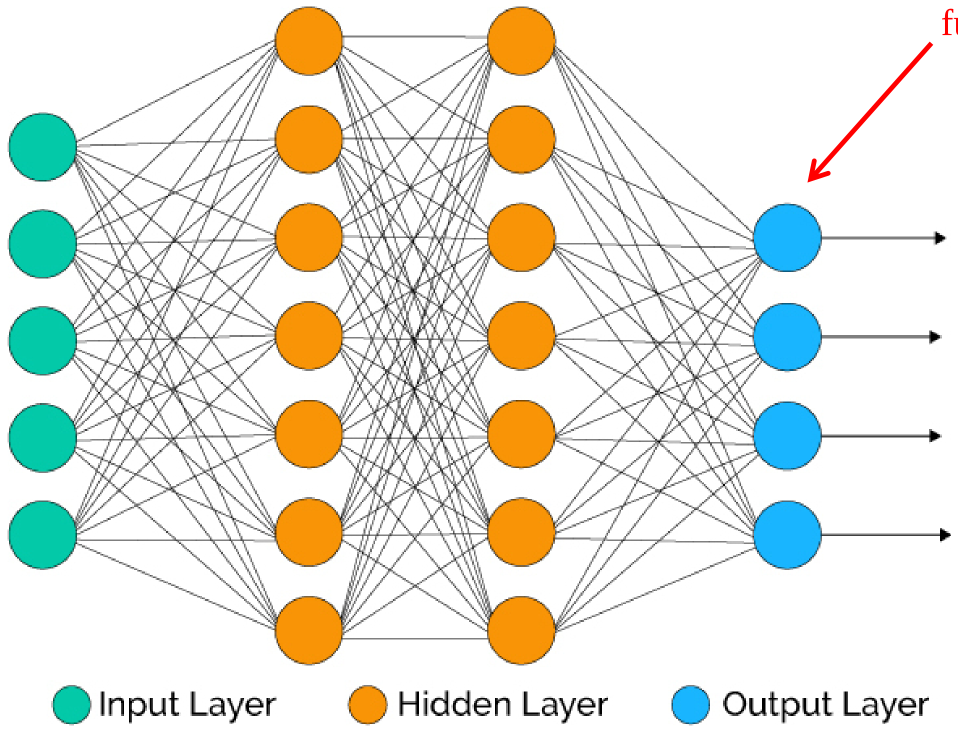


Linear activation functions

Softmax for Mutually Exclusive Categorical Targets with M classes

$$E = -\frac{1}{n}\sum_{i=1}^{n}\left[\sum_{m=1}^{M} y_{im}\ln(\hat{p}_{im})\right]$$

$$y_{im} = \begin{cases} 1 \text{ if the categorical variable is at the m}^{\text{th}} \text{ level} \\ 0 \text{ otherwise} \end{cases}$$



$\hat{y}_{i1} \rightarrow \hat{p}_{i1} = \dfrac{e^{\hat{y}_{i1}}}{e^{\hat{y}_{i1}} + e^{\hat{y}_{i2}} + e^{\hat{y}_{i3}} + e^{\hat{y}_{i4}}}$

$\hat{y}_{i2} \rightarrow \hat{p}_{i2} = \dfrac{e^{\hat{y}_{i2}}}{e^{\hat{y}_{i1}} + e^{\hat{y}_{i2}} + e^{\hat{y}_{i3}} + e^{\hat{y}_{i4}}}$

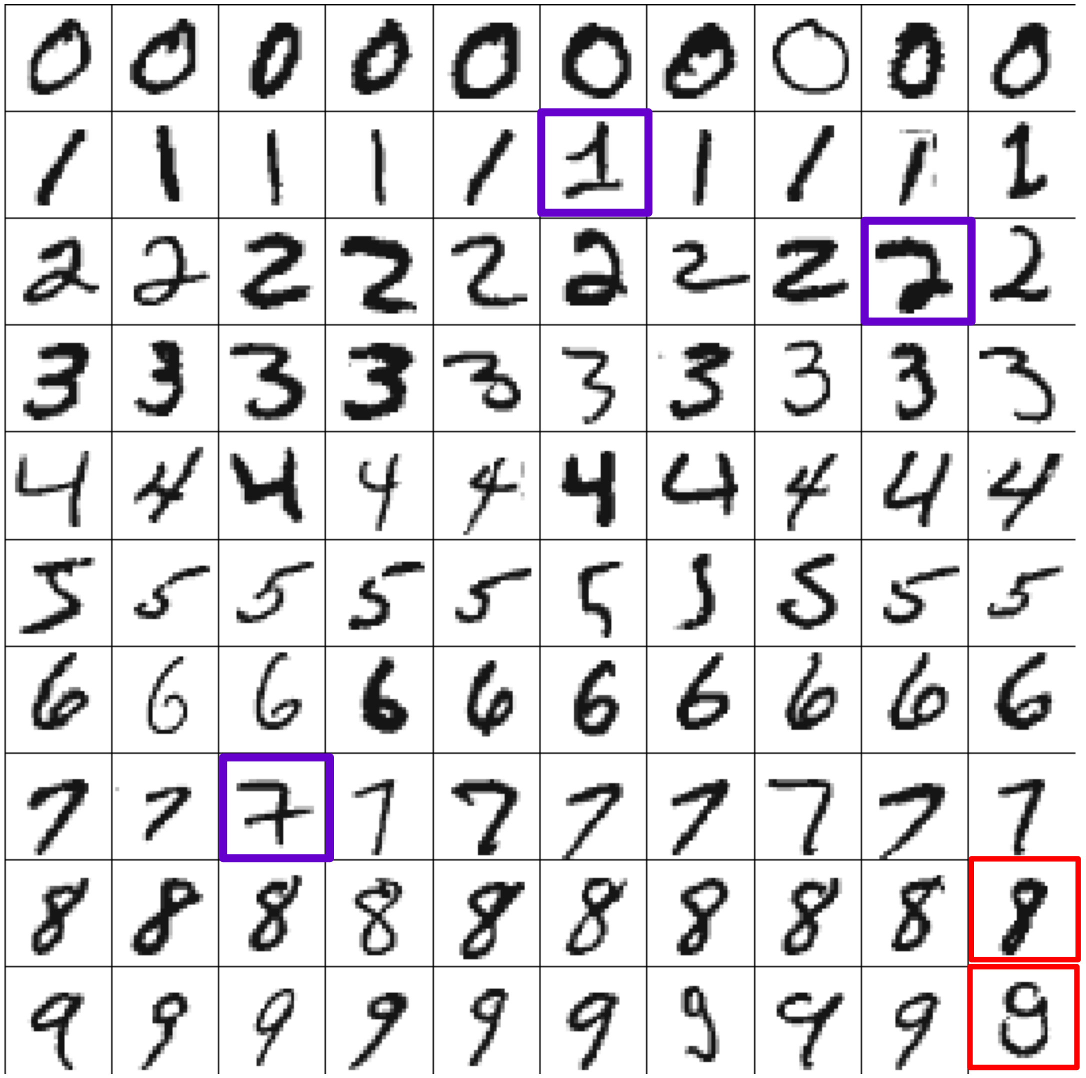
$\hat{y}_{i3} \rightarrow \hat{p}_{i3} = \dfrac{e^{\hat{y}_{i3}}}{e^{\hat{y}_{i1}} + e^{\hat{y}_{i2}} + e^{\hat{y}_{i3}} + e^{\hat{y}_{i4}}}$

$\hat{y}_{i4} \rightarrow \hat{p}_{i4} = \dfrac{e^{\hat{y}_{i4}}}{e^{\hat{y}_{i1}} + e^{\hat{y}_{i2}} + e^{\hat{y}_{i3}} + e^{\hat{y}_{i4}}}$

$$\sum_{m} \hat{p}_{im} = 1$$

MNIST n =60,000 handwritten images,

28x28 pixels, each takes values in {0,1,...,255}



Handwritten Digit Recognition

How would you start to build a neural network to recognize handwritten digits based on MNIST data?

» Input Layer?

» Output Layer?

» Hidden Layers: how many, how big?

» Activation Functions?

Handwritten Digit Recognition

How would you start to build a neural network to recognize handwritten digits based on MNIST data?

» Input Layer?

784 numeric inputs, probably scaled

» Output Layer?

10 numeric outputs

» Hidden Layers: how many, how big?

Great questions. Stay tuned.

» Activation Functions?

Relu (or sigmoid), Linear + softmax output

Things We Still Need

- We should still have lots of questions:
- What is the best way to scale or normalize the input features?
- How, exactly, do we optimize the weights?
- How do we regularize, or otherwise protect against over-fitting?
- How can we fit big NNs on our machines? (*)

nnet() in R

- The `nnet()` `{nnet}` function in R fits simple (one hidden layer, fully connected) networks
  - Also implements linear, logistic, and multinomial logistic regression with no hidden layer and skip connections (directly from input to output)
- Sensitive to random choice of starting values
- Options include “weight decay” (a type of regularization)
- Perhaps best for demonstration rather than serious use

Keras and TensorFlow

- TensorFlow is an open-source machine learning system, in which data in the form of **tensors** (multidimensional arrays) flows from one mathematical operation to the next
- Originally in-house at Google but now open
- Does more than just neural networks
- Keras is a front-end to TensorFlow that hides lots of the pesky details
- Available in R, Python, etc.

A Reminder on Regularization

- "Regularization" is when we intentionally constrain the parameters of the model to prevent over-fitting
  - Our data is correlated in a complicated way; some of our variables are probably noise; we have too many parameters in our model
- We can penalize the sum of squared coefficients ("ridge") or the sum of absolute values ("**lasso**") or both ("elastinet")
- Other strategies also possible, but in NNs, full of parameters, regularization is a must

Keras Notes

- Keras in R uses the "pipe" operator, %>%
  - From maigrittr package
- E.g. `8 %>% log() %>% cos()`

  `≡ cos (log (8))`
- Steps: initialize model, add layers, "compile" (define fitting method, validation set, etc.) and run
- We can add regularization, change activation functions, add validation set…

Keras Notes II

- Keras works great on my local machine, and even better on machines equipped with GPUs
- It can work well in a properly provisioned AWS instance, but ours isn’t
- We’ll have to make do with smaller examples
- Not a limitation of Keras in real life!