OS4118

Statistical and Machine Learning

Neural Networks Part I

Prof. Sam Buttrey

Fall AY 2020

Two + 1 good resources

- Stanford **cs231n** "Deep Machine Learning"
  - A course about the mechanics of neural networks, specifically those whose architectures are tailored to computer vision tasks.
  - Includes coding assignments to complete partial Python scripts in iPython notebooks..

- Goodfellow, Bengio, Courville (2016) *Deep Learning*
  - http://www.deeplearningbook.org/ (slides, chapters, etc.)
  - Good modern overview, with lots of details.

- Chollet and Allaire (2018) *Deep Learning in R*
  - (there is one for Python too)
  - Great examples of using Keras via R
  - Some flakey R code and ideas about machine learning in early chapters.

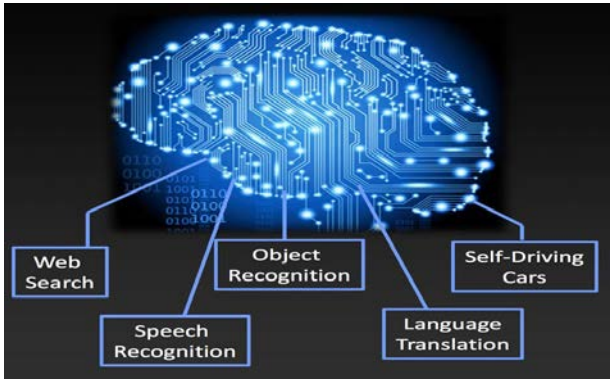
(Artificial) Neural Networks

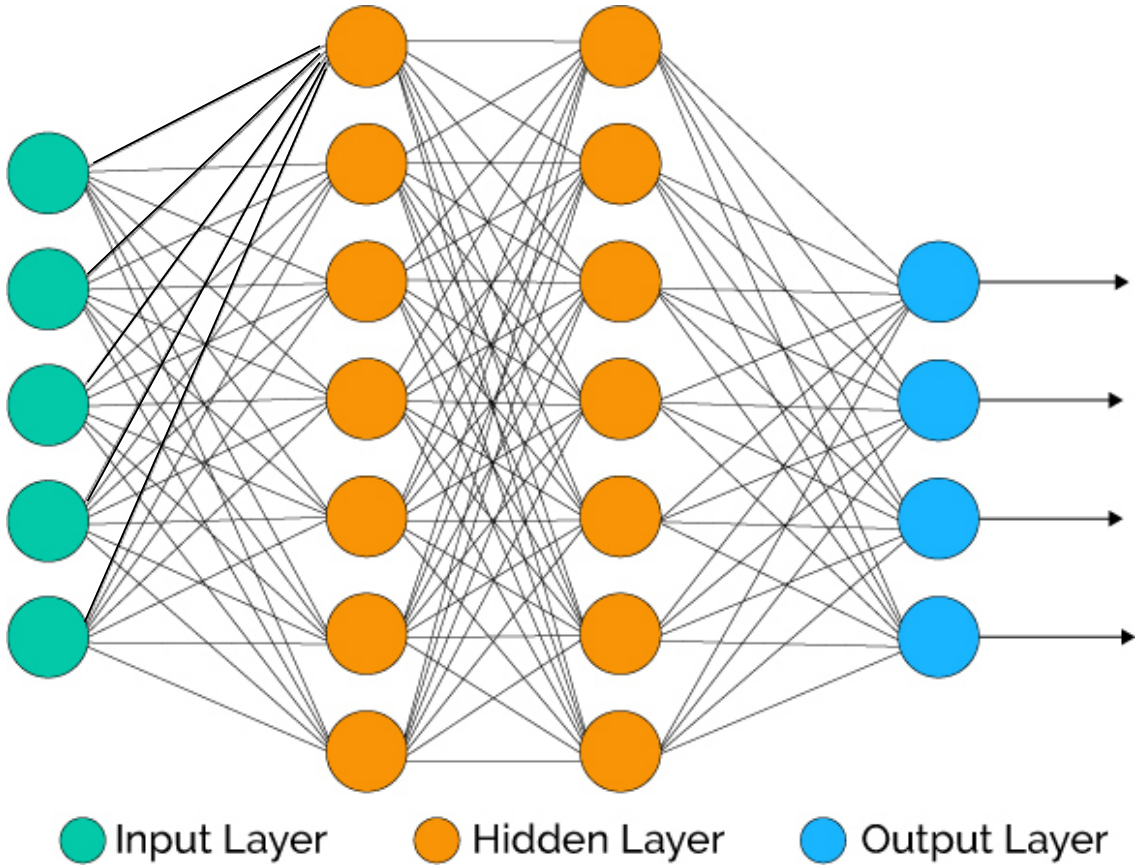
- **Supervised Data:** Most NNs tasks are **Classification** or **Prediction** (self-driving car, voice recognition, computer vision).
  - In the last 10 years NNs have become very good at these tasks.
  - Training these NNs from scratch takes Google like resources. Ng says practical knowledge about training NN in one context does not easily translate to other NN tasks.
  - And, notice that the inputs for these tasks have very specific structure.

- **Unsupervised Data:** To construct new "features" or "code" vectors in lower –D space based on unsupervised data (**auto-encoding**) e.g. 2D codes for images at right.

- **Reinforcement Learning**: Output is unknown, but get positive (delayed) feedback (Google Alpha-Go, and all sorts of AI tasks)

Deep NN models are loosely patterned after neuron and brain

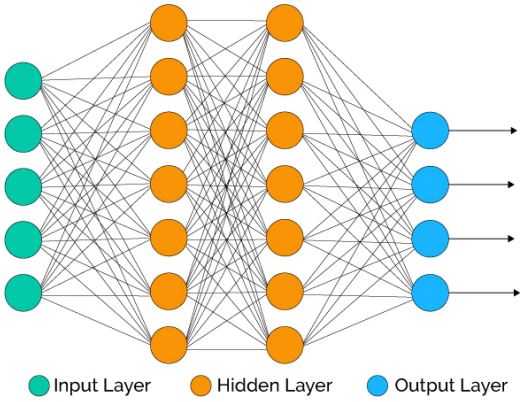




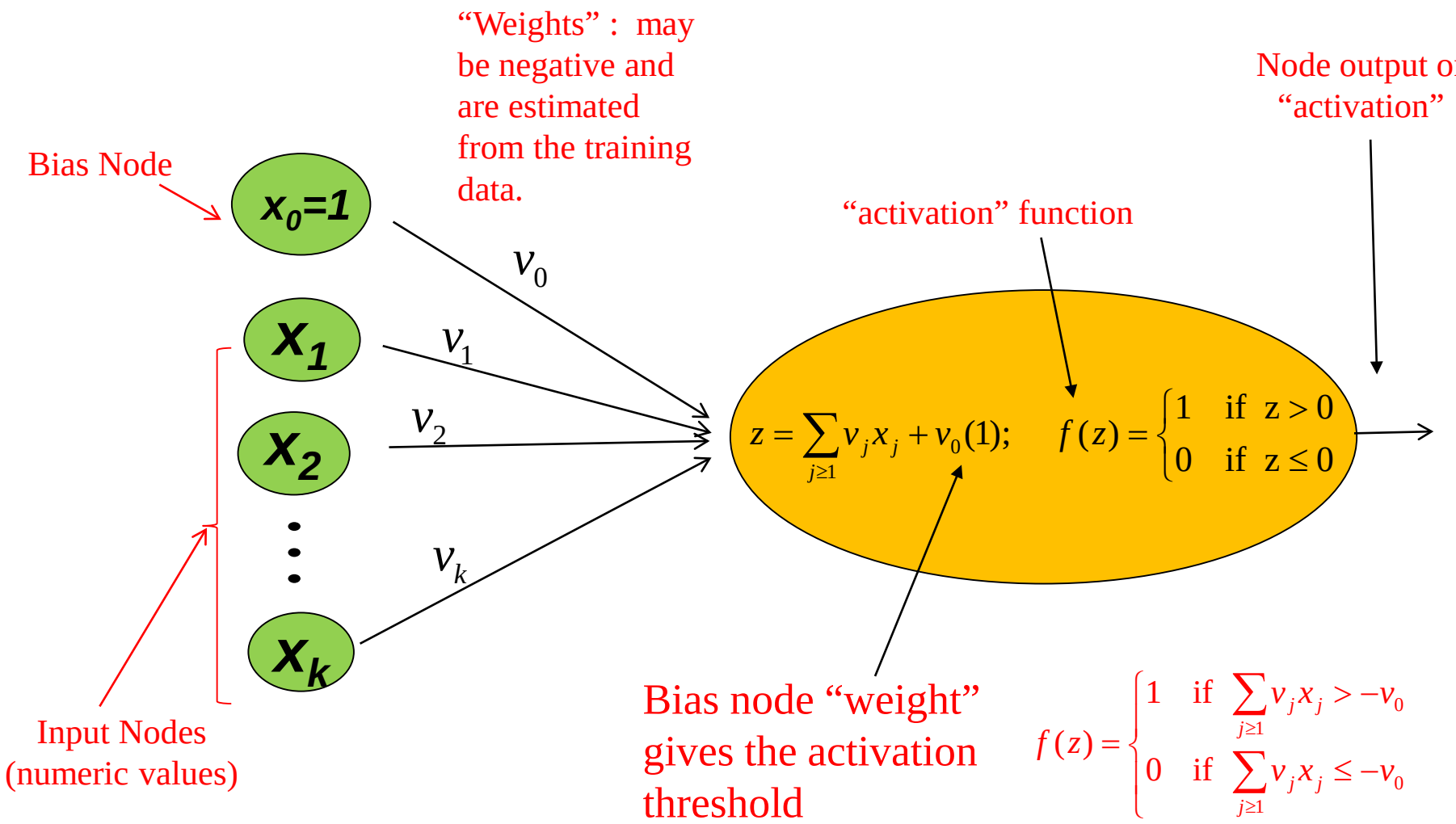
Neural Networks

- NN are **layers** of features. The key idea is: the input to each node in a layer is a new feature which is a function of the outputs of the previous layer.
- NN training automatically new features in each layer. The features they generate are better able to predict or classify than the original features.
- (Compare SVMs)
- The biological analogy motivates the basic structure and goals of neural network models, but that's where it ended… until fairly recently, with the advent of deep neural networks.

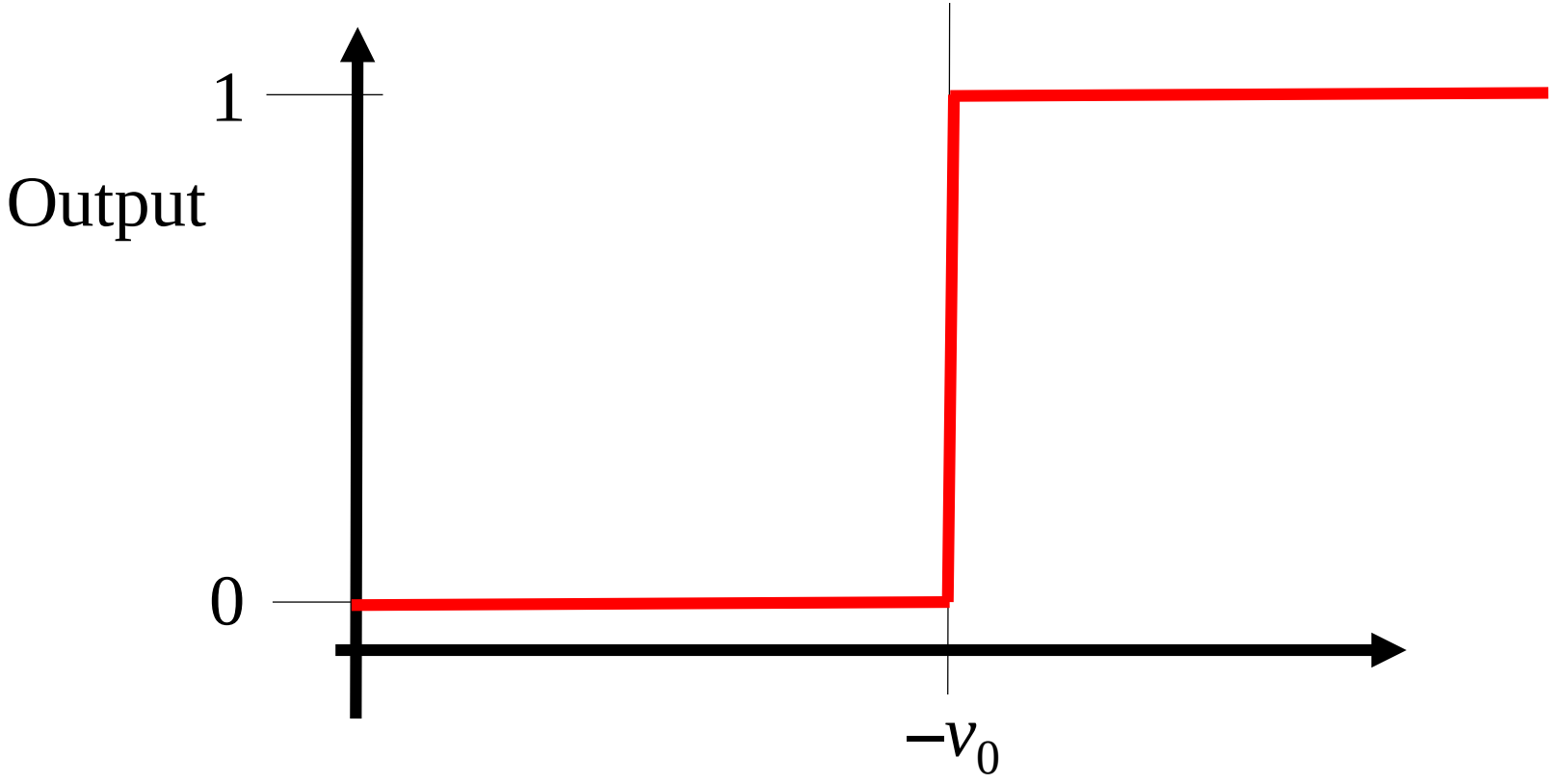
A Historical Perspective of Neural Network Model



McCullough and Pitts (1943) Perceptron Model

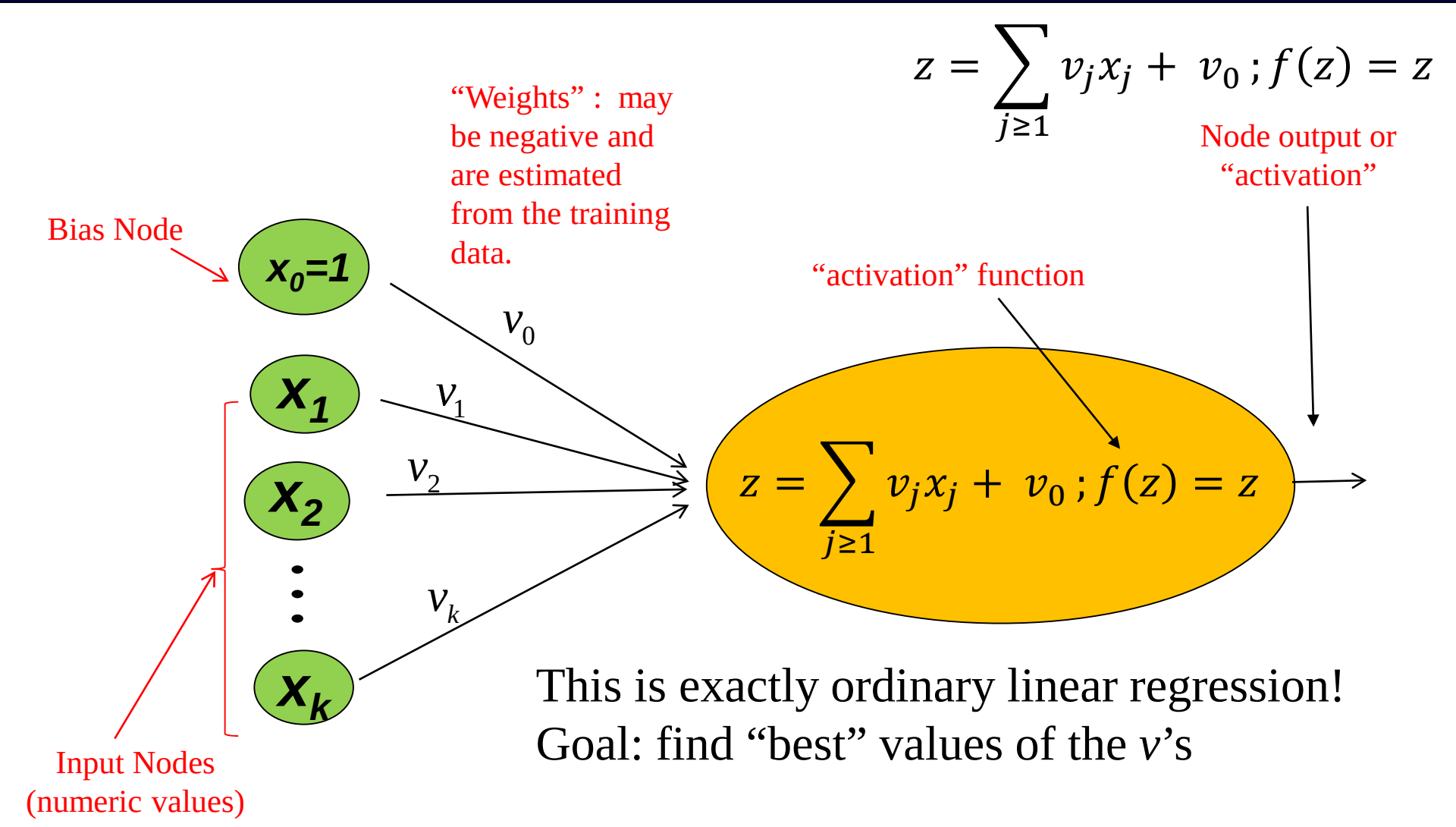


Mc.C. and P. Perceptron Output



Total Input Value = $\Sigma_{j \geq 1} v_j x_j$

What if we used linear output?

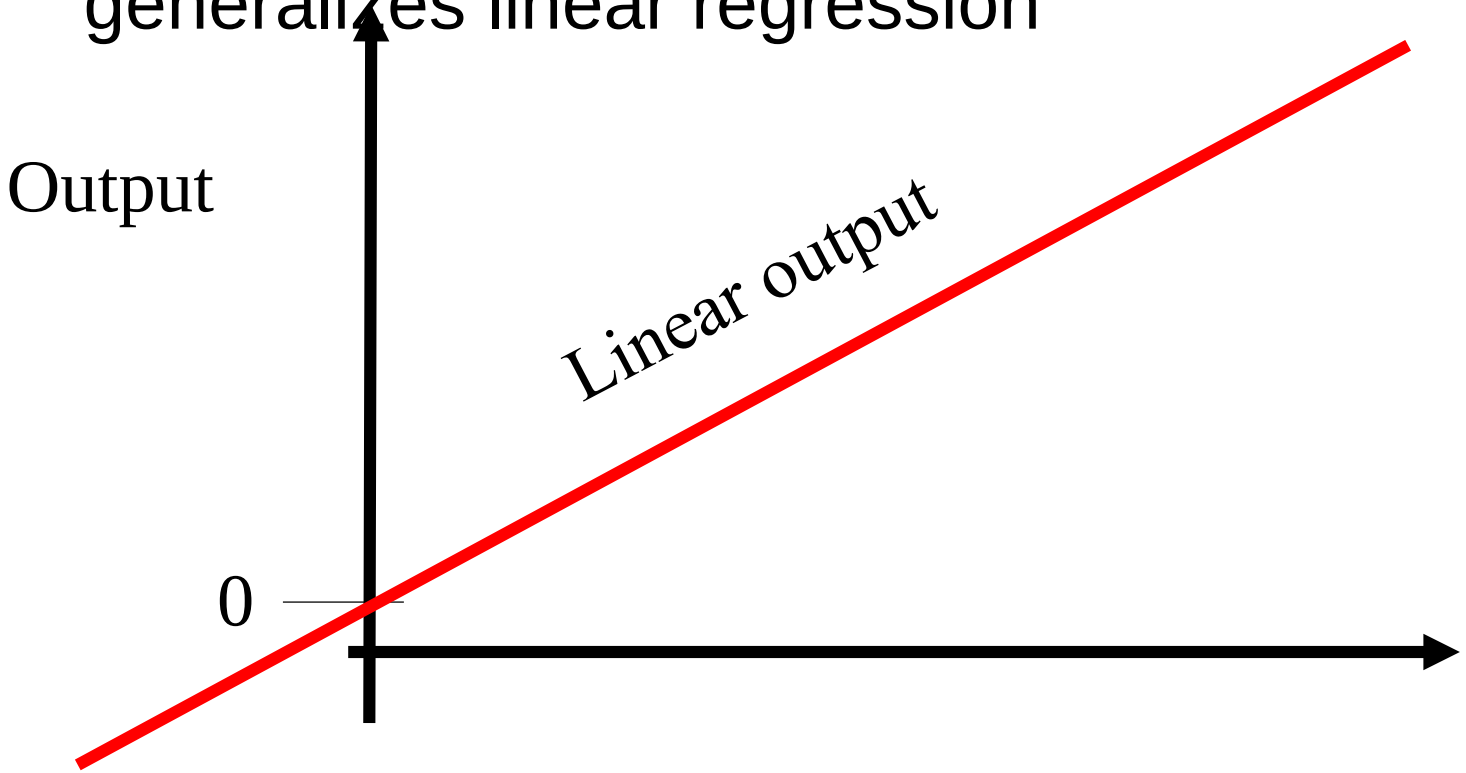


This is exactly ordinary linear regression!

Goal: find "best" values of the v's

What if we used linear output?

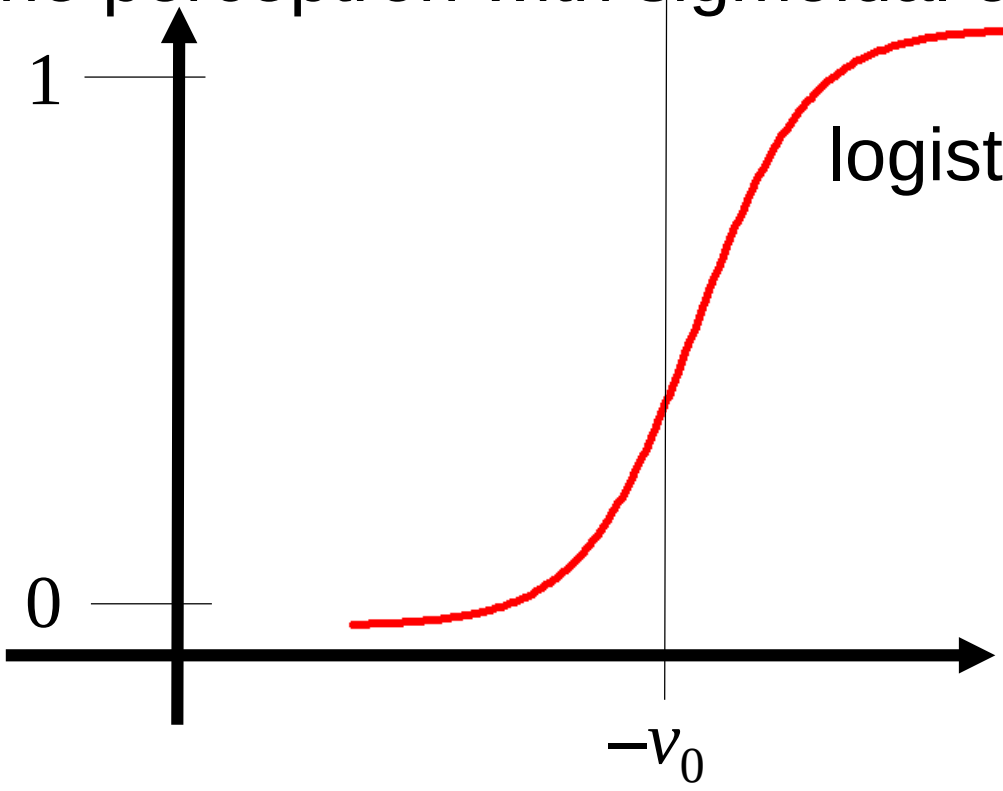
- So one perceptron with linear output generalizes linear regression



Total Input Value = $\Sigma_{j \geq 1} v_j x_j$

Sigmoidal Activation

- $f(x) = 1/(1 + \exp(-x))$ has been very popular
- One perceptron with sigmoidal output generalizes logistic regression



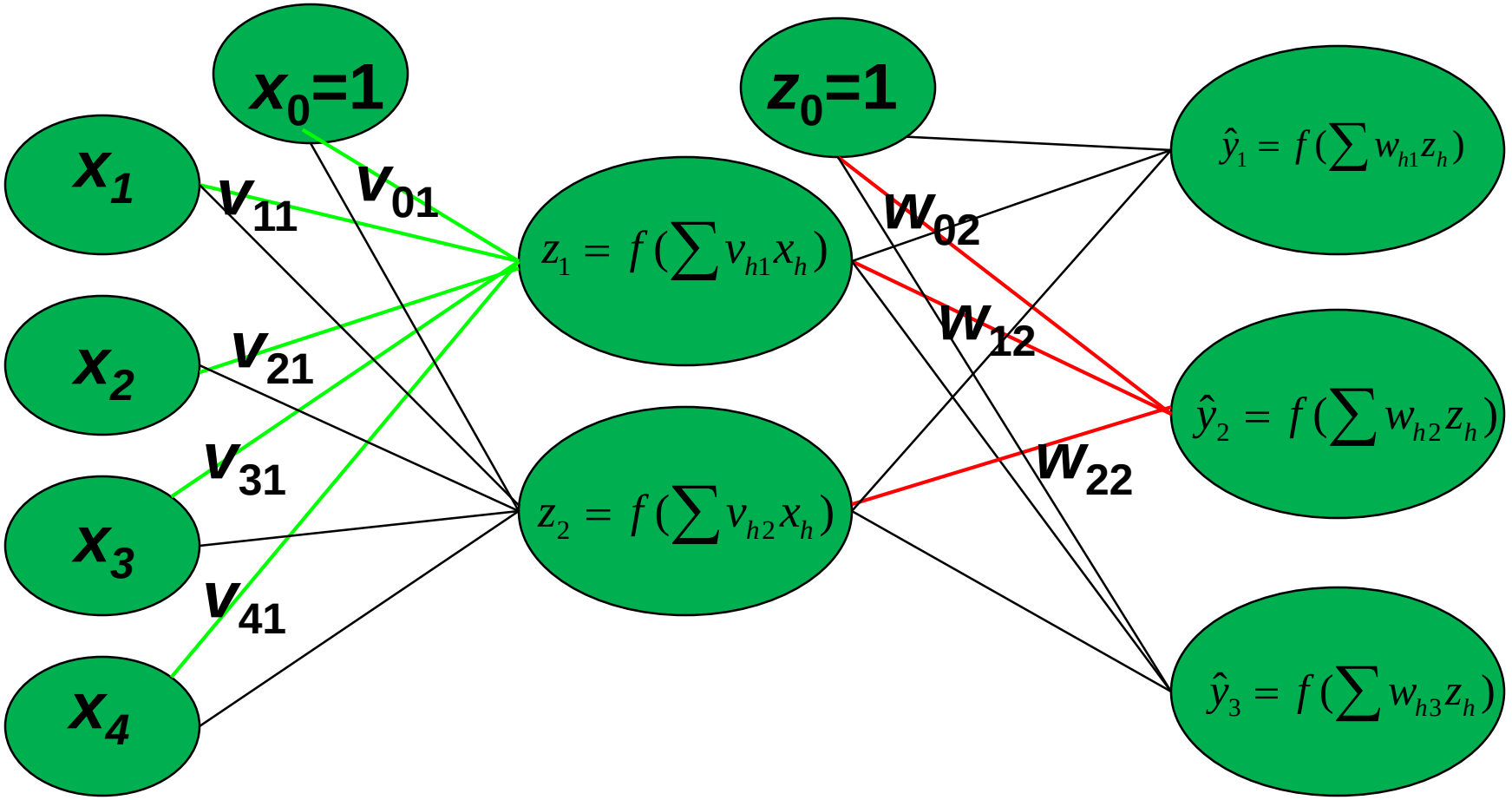
Big Deal

- So a perceptron with **linear** output implements linear regression
  - But without all the machinery for hypothesis testing, prediction/confidence intervals, etc.
- A perceptron with **sigmoid** output implements logistic regression
  - As we shall see – and, again, without the formal hypothesis testing framework
- So what's the point?

Multiple Layers

- What makes nets more powerful than ordinary regression is that they can have **hidden layers**
- This is another set of perceptrons between the input and output
- Very often, every input node (+ bias) will connect to every hidden node, and every hidden node (+ bias) will connect to every output node
- Such a network is **fully connected**

Two-Layer Neural Network Model



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

A Net Is A Statistical Model

- This two-layer neural net is a **statistical model** of the form
- $o_{ij} = f(\sum_{h \in \text{hidden}} W_{hj} f(\sum_{i \in \text{input}} v_{ih} x_i)$
- And then $\hat{Y}_i = \arg\max_j o_{ij}$ (classification)
- Or $j = 1$ and $\hat{Y}_i = o_i$ (regression)
- We select the weights ***w*** and ***v*** to optimize some criterion, just as in regression

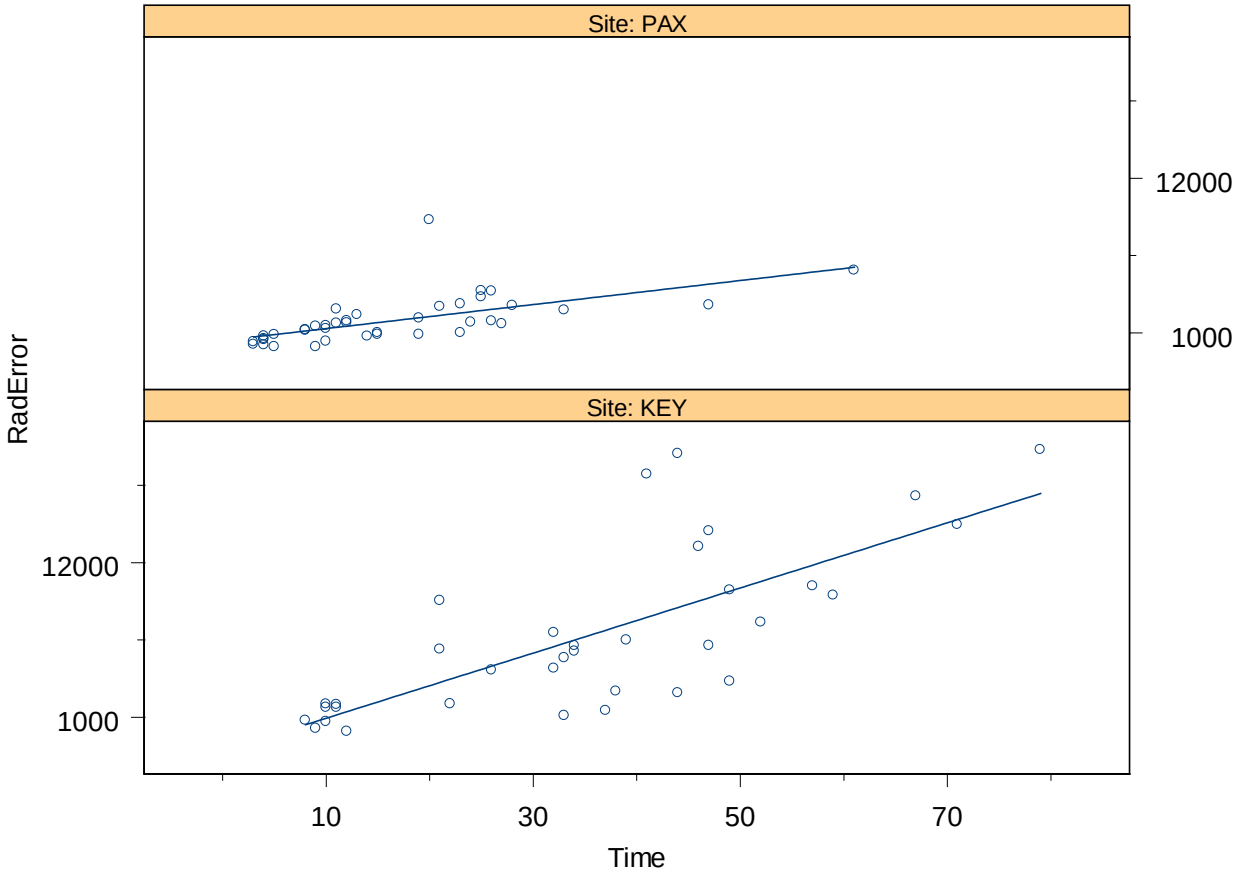
Output Nodes

- In a regression problem there will be one continuous-type output node
- In a *k*-category classification there will usually be *k* continuous-type output nodes, normalized to add up to 1, by, e.g., **softmax**: $p_j = \exp(k_j) / \Sigma_j \exp(k_j)$
- In a two-category classification there will usually be only one output node, not two

A Simple Example
(NN with no hidden layer)

Tacnav data: y = RadERROR

x1 = SITE (2 levels); x2 = TIME (numeric);



In R use nnet {nnet}

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

initial  value 3041220547.434324
final  value    671693404.335941
converged
a 2-0-1 network with 3 weights
options were - skip-layer connections  linear output units
     b->o    i1->o    i2->o
 -1207.64  1461.15   185.37
```

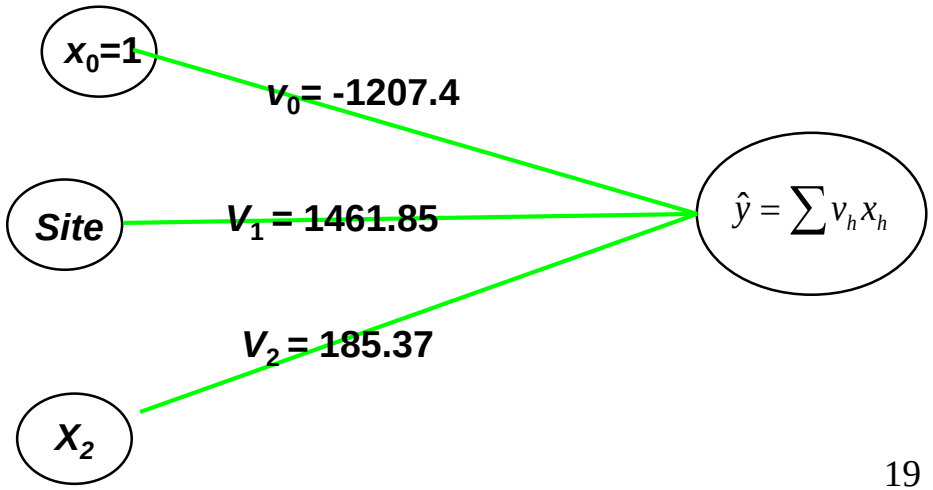
All inputs "skip" to output

Size of hidden layer = 0

Linear Activation Function at Output

Initial value of SSE; initial values of parameters are chosen at random.

Weight from Bias node to Output node



This Neural Network is equivalent to fitting an additive linear model:

$$\hat{y} = -1207.639 + 1461.36x_1 + 185.36x_2$$

$$\text{where} \qquad x_1 = \begin{cases} 1 & \text{PAX} \\ 0 & \text{KeyWest} \end{cases}$$

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

Coefficients:
 (Intercept)     SITE     TIME
   -1207.639 1461.152 185.3652

Degrees of freedom: 74 total; 71 residual
Residual standard error: 3075.788
```

```
sqrt(671693404.335941/71)
 [1] 3075.788
```

Incorporating Categorical Variables

- In this 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 has one 1 and (*k* – 1) 0's…
- …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

(Multinomial) Logit Regression

- Two-class logistic regression example:

```
glm (Species == "versicolor" ~ .,
data = iris, family = binomial)$coef
```

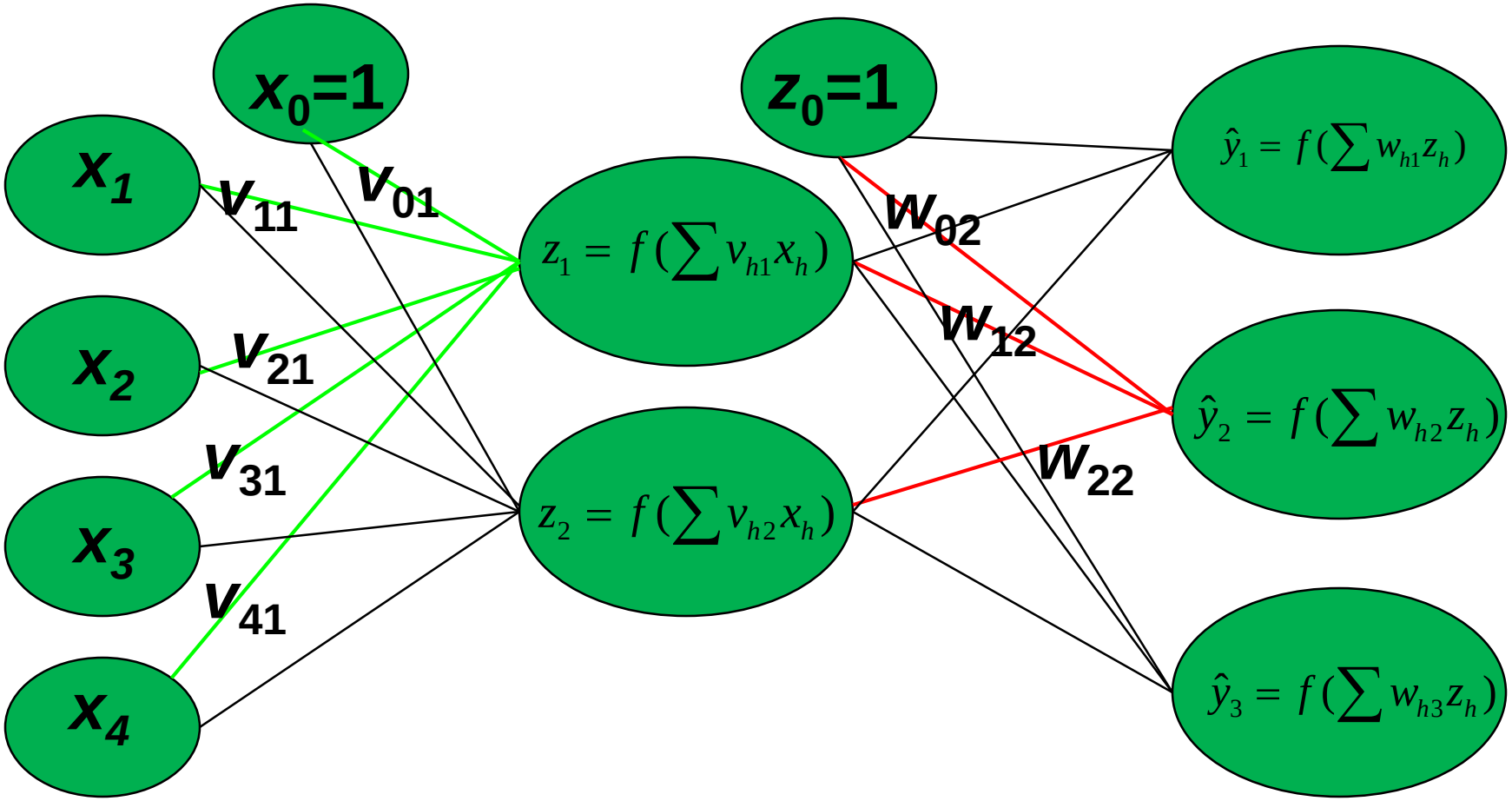
- Compare the nnet version, no hidden layer, skip connections, entropy (as opposed to least-squares) fitting

```
nnet (Species == "versicolor" ~ .,
data = iris, size = 0, skip = T,
entropy = T)$wts
```

and this extends to

```
nnet (Species ~ ., data = iris, size
= 0, skip = T, entropy = T)
```

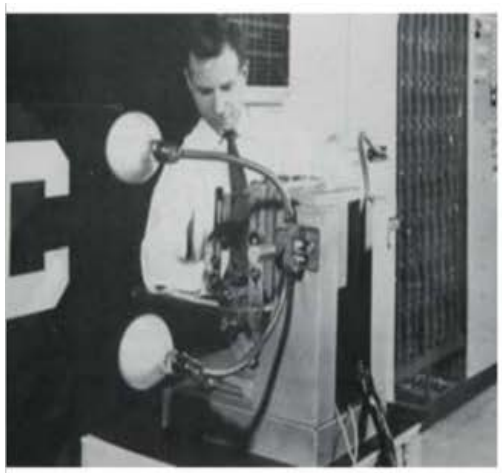
Back to Multi-Layer Networks



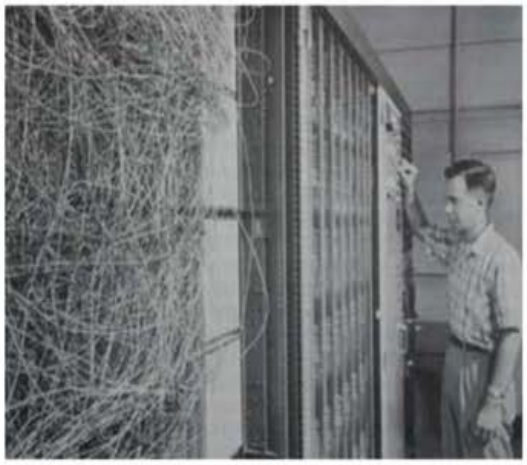
…where $f(t)$ is the activation function

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
- Multi-layer perception networks are still used for tasks with large numbers of inputs, e.g. Google and have been used in natural language processing.



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 stopped until the 1980's

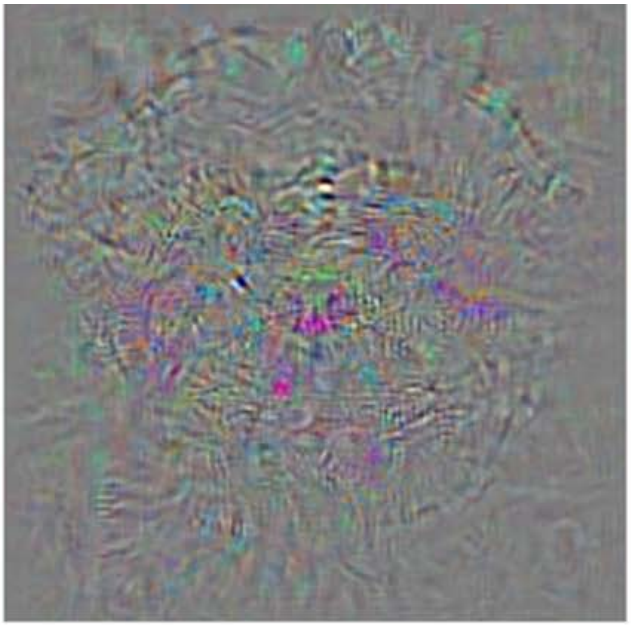
Problem #1: Exaggerated claims were made about perceptron networks resulting in bad PR.

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 miss-cited to apply to multi-layer perceptron models too.

Problem #1: Modern day version.
What do NN ‘s “see”? And how to fool them.

dog



\+ Small Perturbations

ostrich

Christian Szegedy (Google) et al.(2014)

Modern (Artificial) Neural Networks

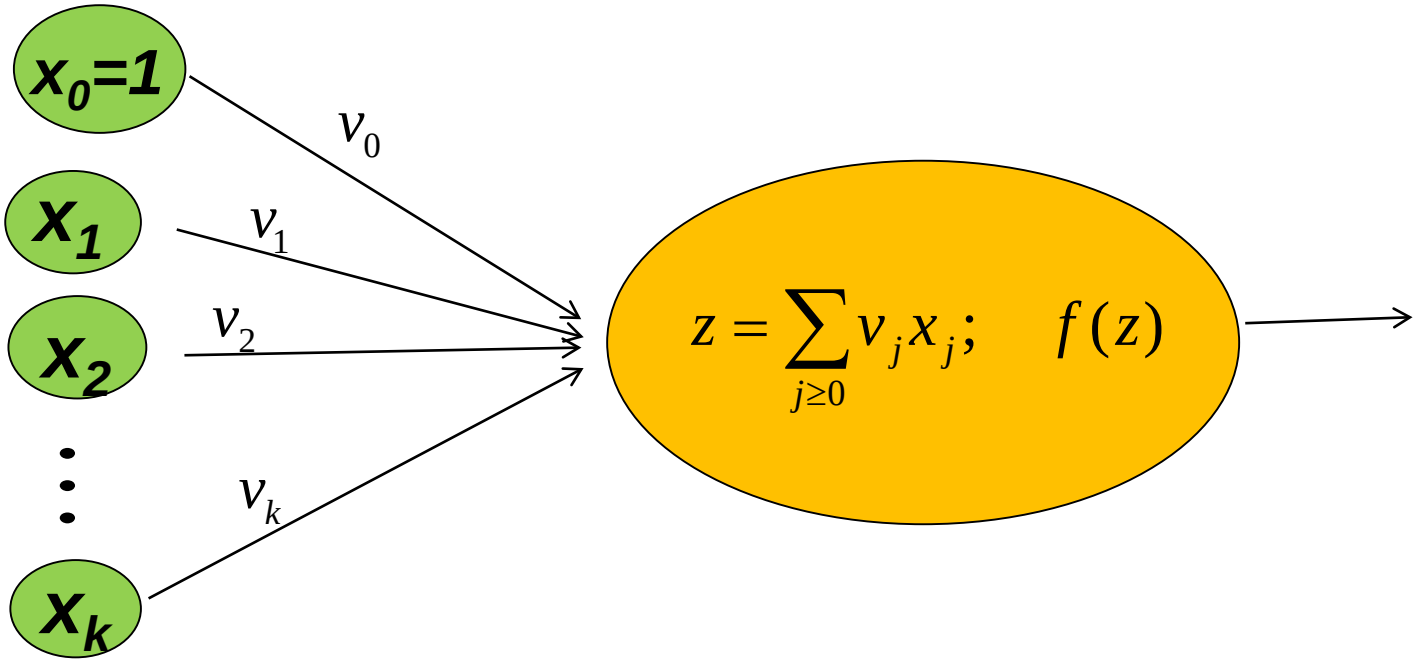
Problem #1: Exaggerated claims were made about perceptron networks resulting in bad PR.

Problem #2: The perceptron only gives binary outputs 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.

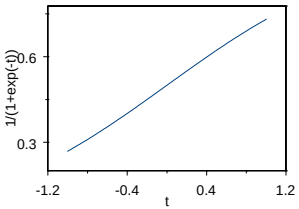
Rumelhardt and McClelland (1986) introduce continuous activation functions

This allows numeric outputs for numeric targets



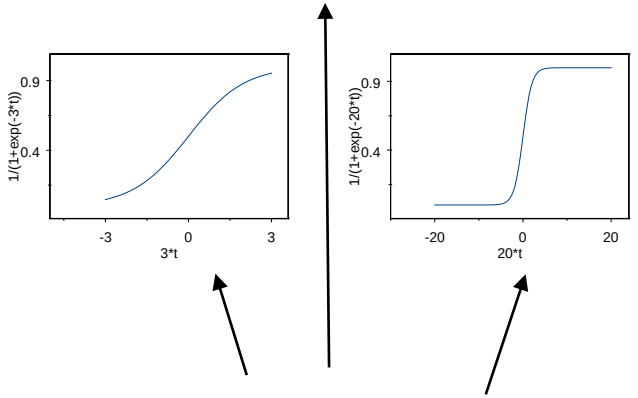
Popular Activation Functions

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



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

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

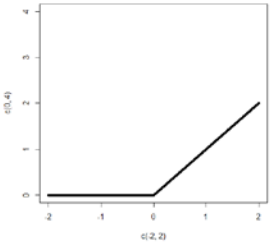


Hyperbolic Tangent $$f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Bounded Numeric output

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

Numeric positive output

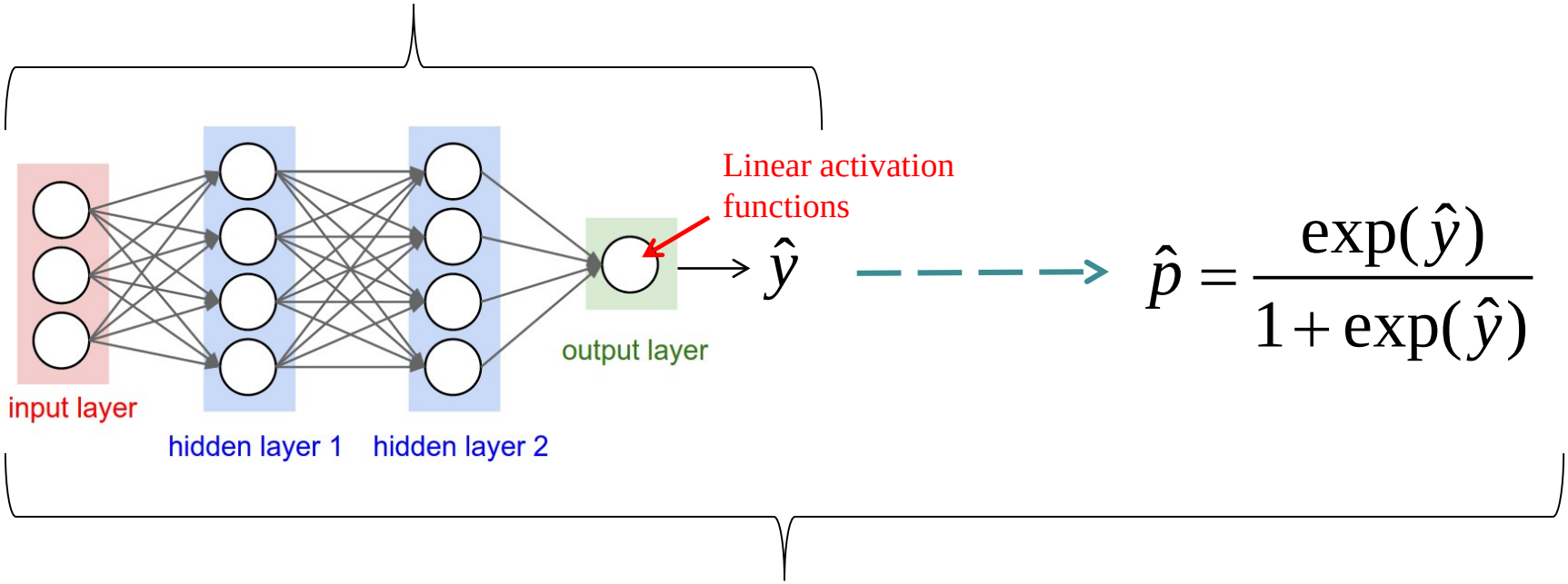


Another benefit of continuous activation functions is:

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

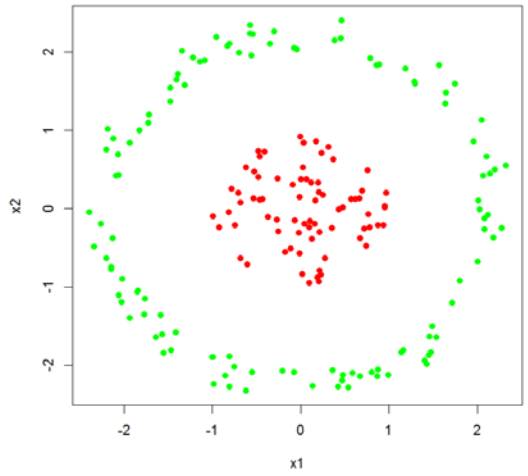


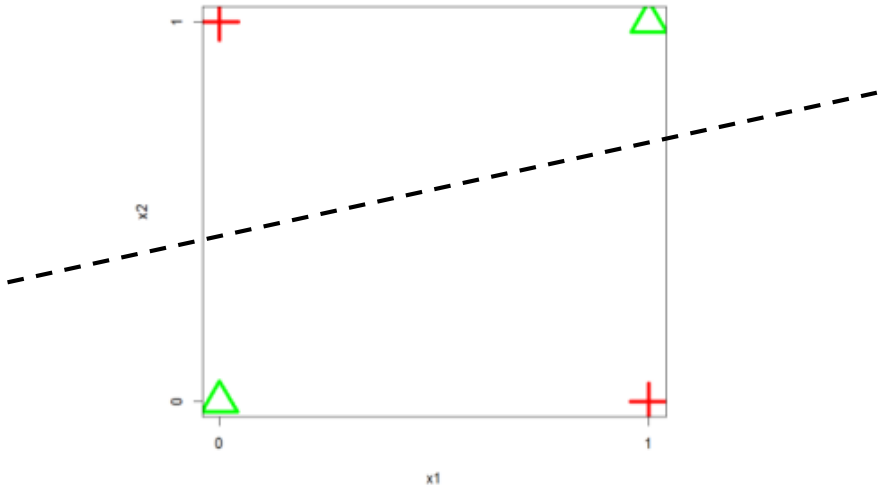
For Categorical Targets with two levels:

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

The Minsky – Papert problem

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

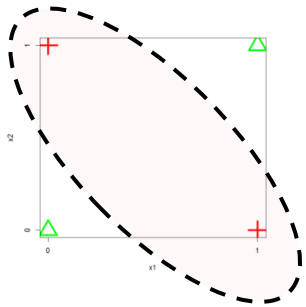




Linear Boundary: $\upsilon_0 x_0 + \upsilon_1 x_1 + \upsilon_2 x_2 = 0$

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.



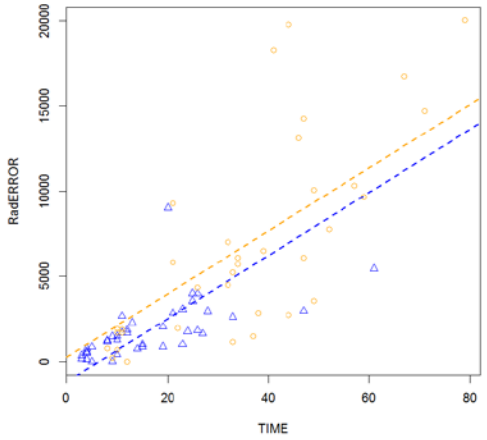
Linear Boundary: $\upsilon_0 x_0 + \upsilon_1 x_1 + \upsilon_2 x_2 + \upsilon_3 x_3 + \upsilon_4 x_4 + \upsilon_5 x_5 = 0$

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

- 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

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

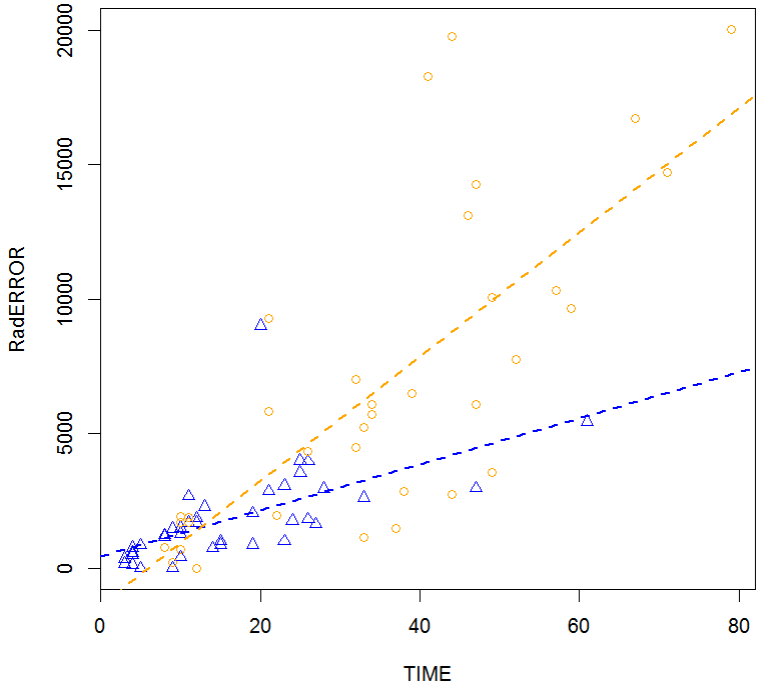
> lm(RadERROR~SITE+TIME,data=tacnav.data)



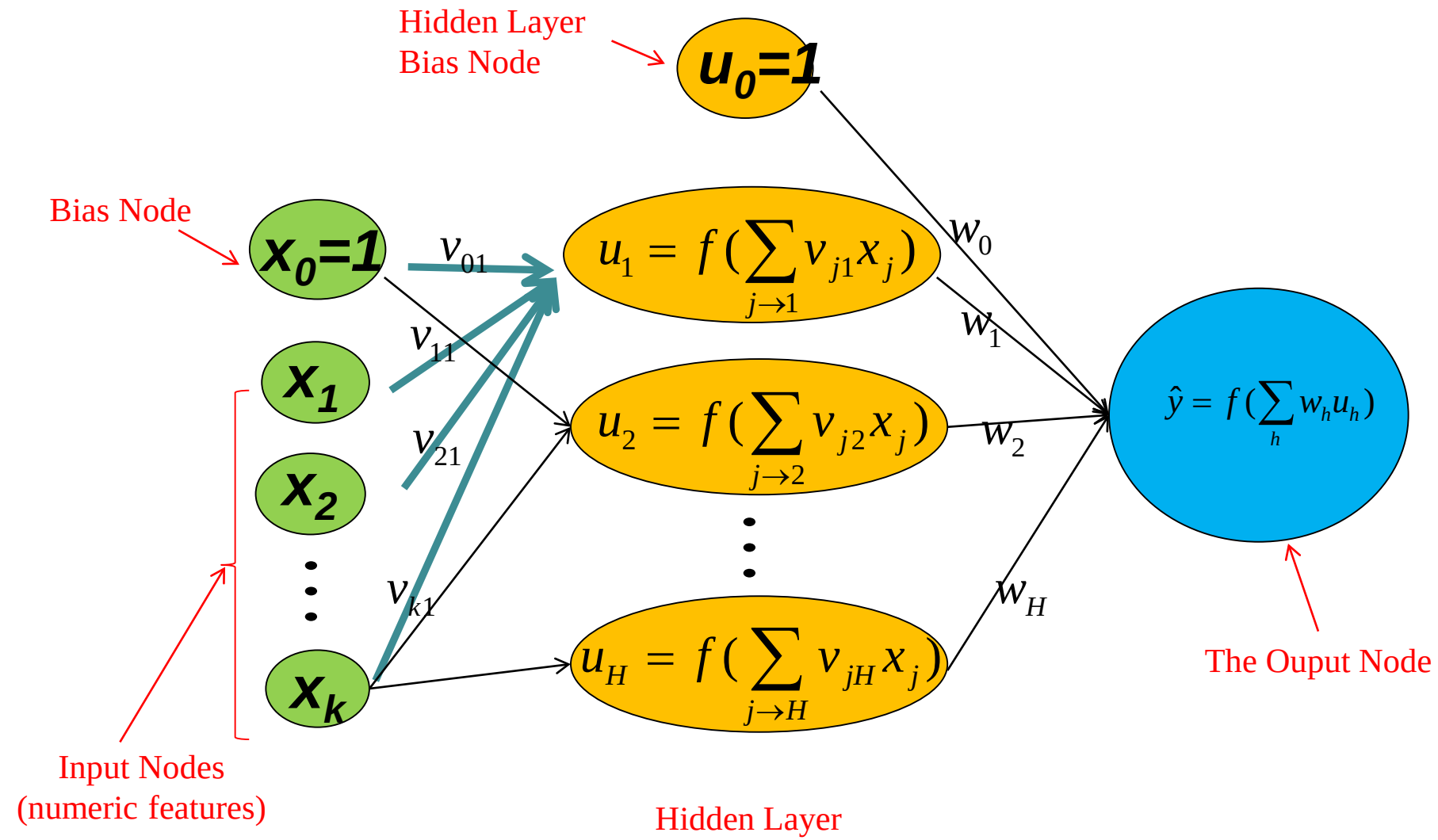
By defining new input features, a NN can handle any model a glm can, but….

```
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
```



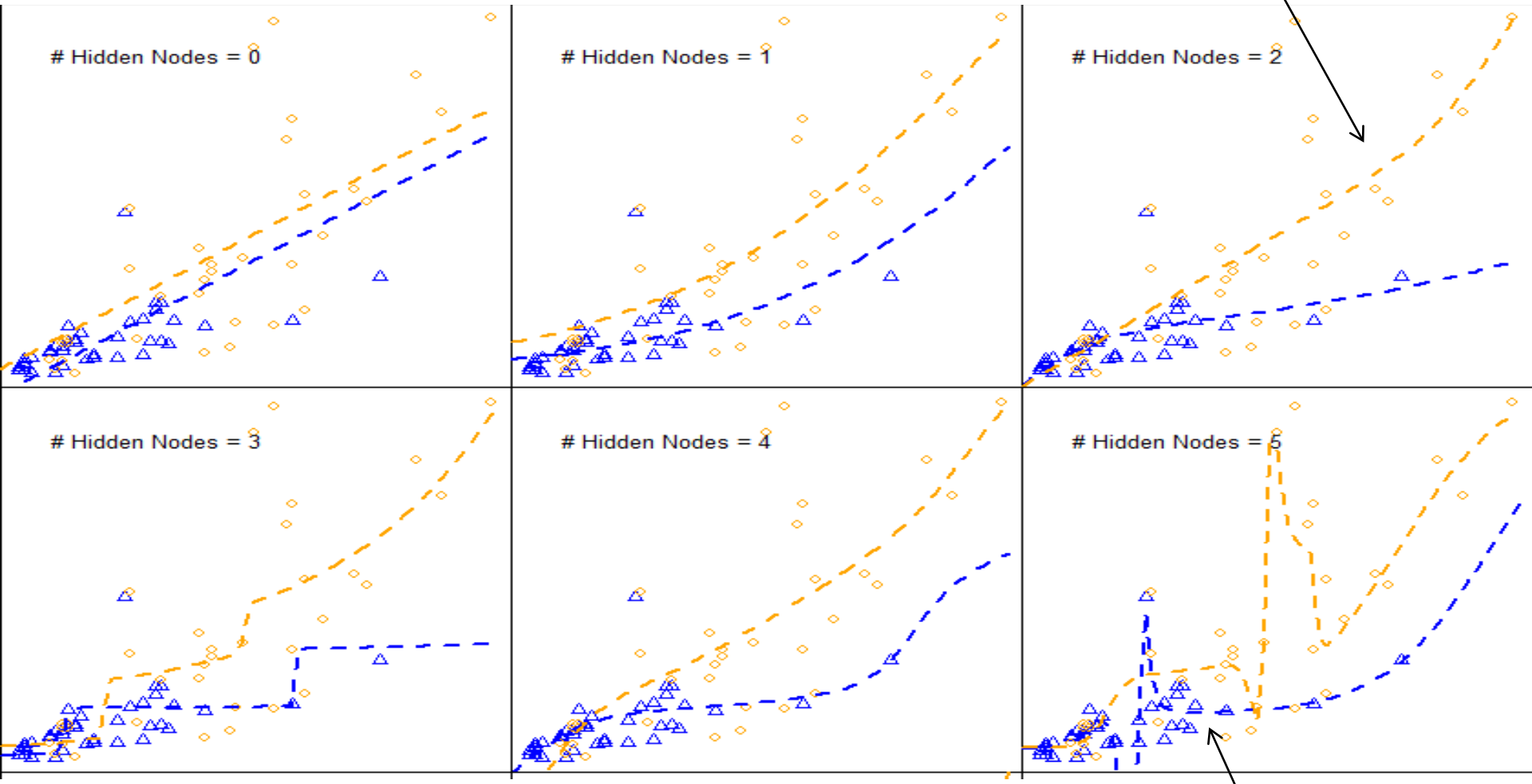
So what about the hidden layer?



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 not currently available!)

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

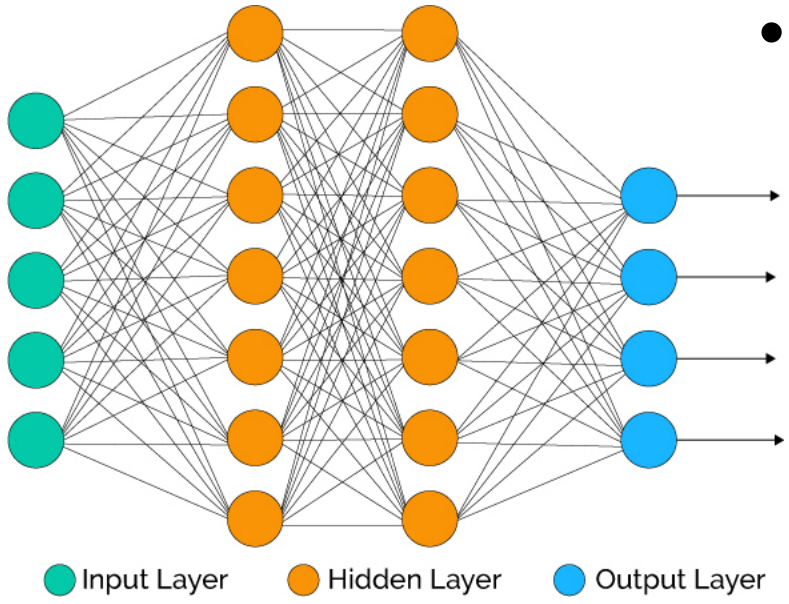
- Simple 1 - hidden layer fully connected with NN, numeric target

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

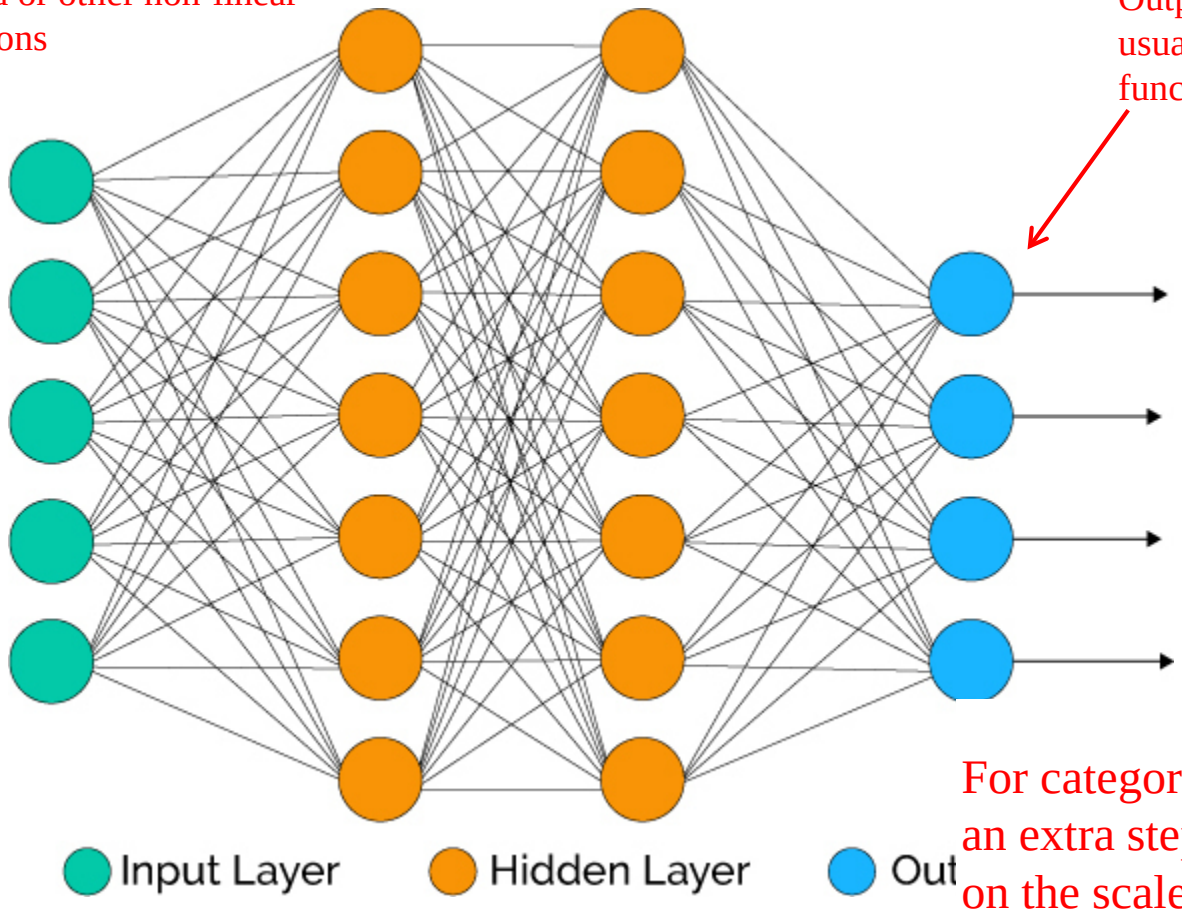


- This makes them able to emulate input signals more complicated that what a simple linear regression can deal with

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].

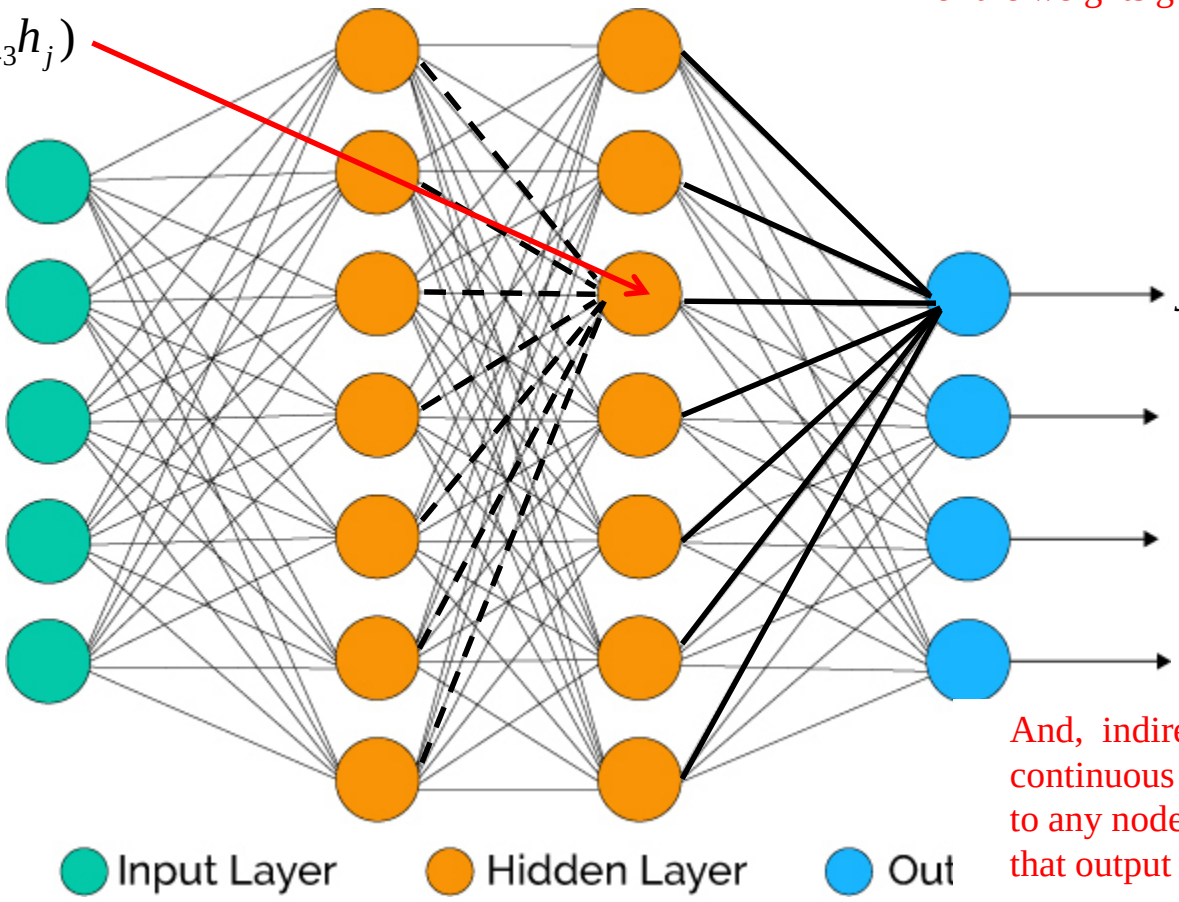
Outputs are continuous functions of weights

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

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.

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

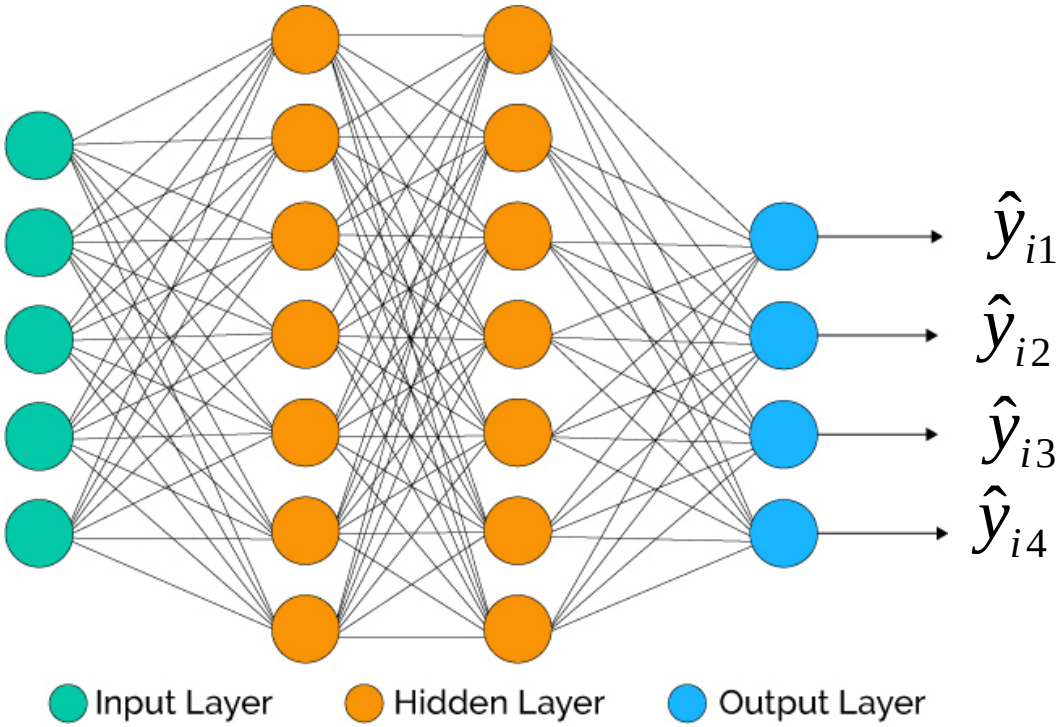
How many layers / nodes?

- 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 for Multiple, M, Numeric Targets (maybe?)

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

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

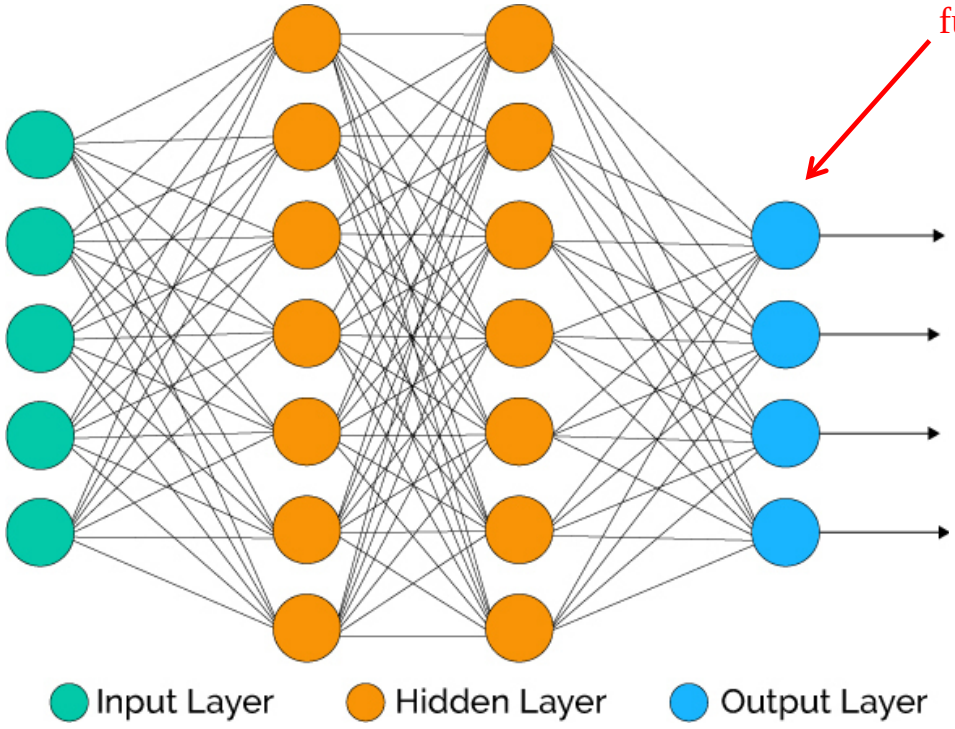


Linear activation functions

Softmax for Mutually Exclusive Categorical Targets with M classes

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

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



Linear activation functions

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

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

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

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

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

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? (*)

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.