Lecture 8:
Feed-forward neural networks (FFNNs)

CS 4740 (and crosslists): Introduction to Natural Language Processing

Slides developed by:
Magd Bayoumi, Claire Cardie, Tanya Goyal, Dan Jurafsky, Lillian Lee, James Martin, Marten van Schijndel

Announcements

- **HW0 and HW1 milestone graded on Gradescope**
  HW1 due on Gradescope on Friday by 11:59pm

So far this semester…computational models + NLP tasks

- N-gram language models
  - Predict the probability of a sequence of tokens
  - Predict the next token in a sequence
- HMMs for sequence tagging tasks
  - E.g., POS tagging, NER
  - MEMMs (briefly covered)
    - $P(t_i|t_{i-1}) \rightarrow P(t_i|t_{i-1}, o_{1:N})$
- Logistic regression for text classification (binary)
  - E.g., document/sentence-level sentiment classification
  - Convert input into a set of *feature-value* pairs
  - Learns *weight* to apply to each feature-value pair

Last class: Representing word types

- How did we represent words for each of the NLP models we've covered?

Recall from the last class

- Words as vectors

  Vectors of words with similar meanings/uses should be close in the vector space

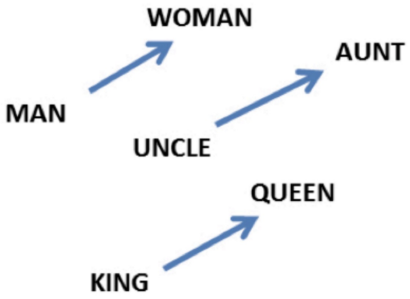
  Co-occurrence vector that encodes ***context*** using raw counts

	aardvark	…	computer	data	result	pie	sugar	…
digital	0	…	1670	1683	85	5	4	…
information	0	…	3325	3982	378	5	13	…

Improve using *tf-idf* representation

Word embeddings – (short) dense word vectors

- Learn via the *skip-gram* model (Word2Vec)
- Each vector element, $0 < v_i < 1$
- Nice properties naturally emerge
  (but so do ***biases*** from the data)



Immediately improved performance on virtually all NLP tasks!!!

Today

- Feed-forward neural networks
  - As a mechanism for ***representation learning*** → **for word embeddings**
  - As another approach to classification problems for text/language

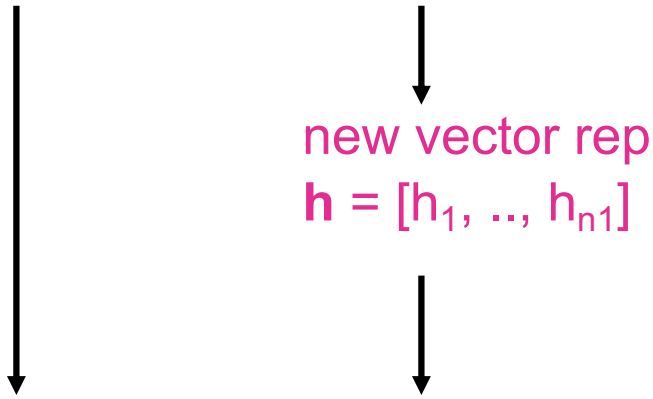
The problem: generic word vectors might not be optimal

For a *specific* task, such as NE tagging, we could learn better representations.

- Example: it'd be nice if the vectors for LOC words were separate from the vectors for PER words.

The problem: generic word vectors might not be optimal

Given: word2vec rep for word *w*,
x = [x_1, ..., x_i, …, x_{n0}]



new vector rep
h = [h_1, .., h_{n1}]

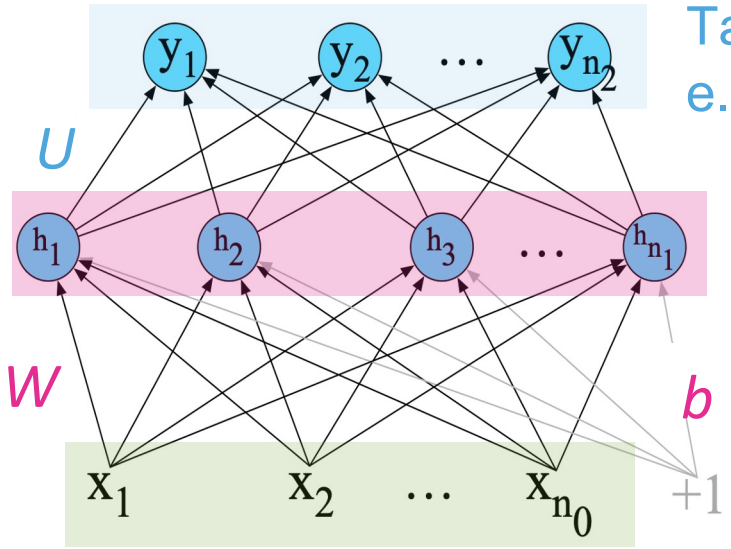
Predict: set of n_2 task-specific outputs,
y = [y_1 = P(ORG|*w*), y_2 = P(PER|*w*), … y_{n2} = P(LOC|*w*)]

FFNNs will acquire the ability to perform this task by learning to derive better representations of the word vectors.

The solution we're building up to today: feed-forward neural networks (FFNNs) for classification

- Network of small simple(!) computing units
  - Each takes a vector of input values and produces a single output value
- Computation for classification proceeds from one layer of units to the next
- **Deep learning** when the network has many layers
- Logistic regression is a really basic FFNN
- Unlike LR, **no feature engineering** is needed for FFNNs!!!
  - Take dense word embeddings as input
  - Induce the necessary features as part of learning to classify
- Downside: need a large amount of training data

The solution we're building up to today: feed-forward neural networks (FFNNs)



Task-specific outputs,
e.g., y_1 = P(ORG|w)

new vector rep
h = [h_1, .., h_{n1}]

for word w, word2vec input
x = [x_1, ..., x_i, …, x_{n0}]

Intuition: FFNNs learn how to better represent **x** (i.e, as **h**) so as to be able to accurately predict **y**.

The problem again: generic word vectors might not be optimal for a specific task

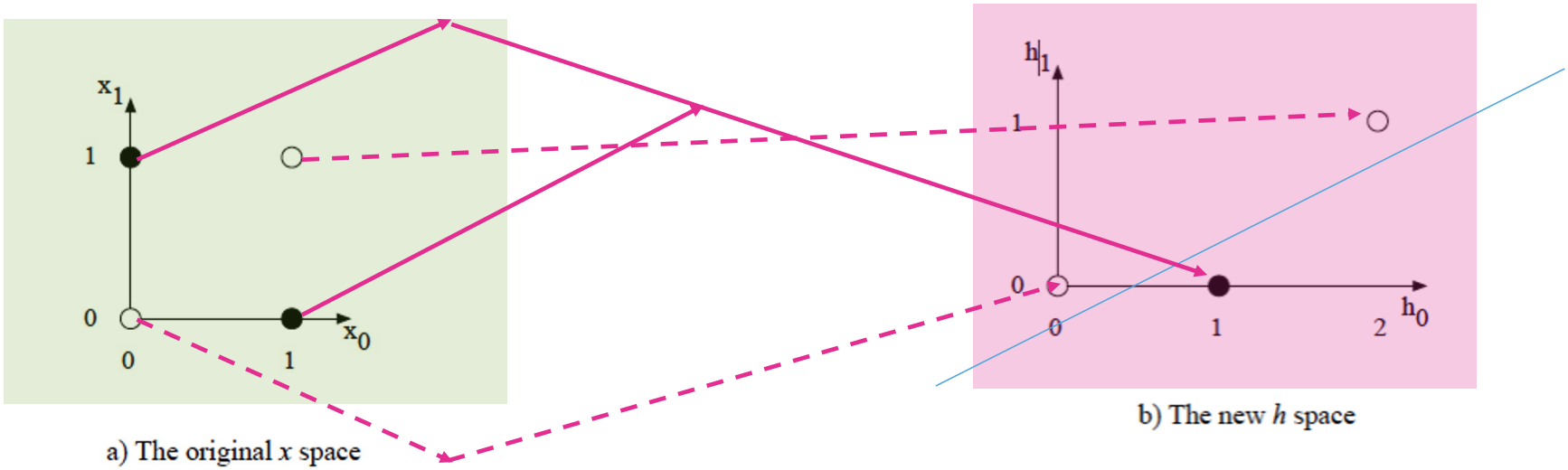
Recall: for a *specific* task, such as NE tagging, we could learn better representations.

- Example: it'd be nice if the vectors for ORG words were separate from the vectors for PER words.

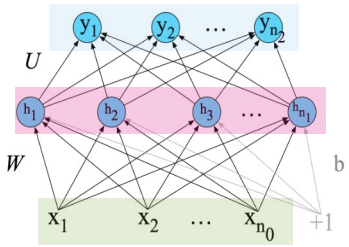
Changing the original x's into new h's where items with different labels are separable

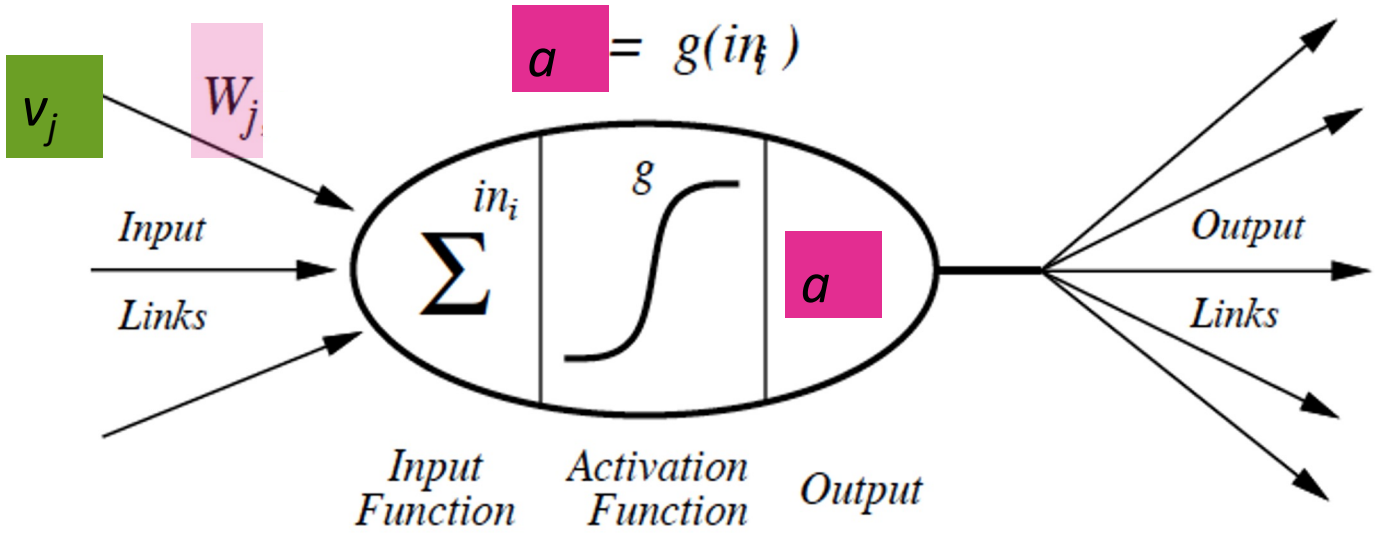
Suppose that ⬛ = PER, and ○ = LOC, and our w2v vectors looked like this:

There's a transform whereby the PERs can be separated from the LOCs:



Our transformations will use "computational neurons"/units.

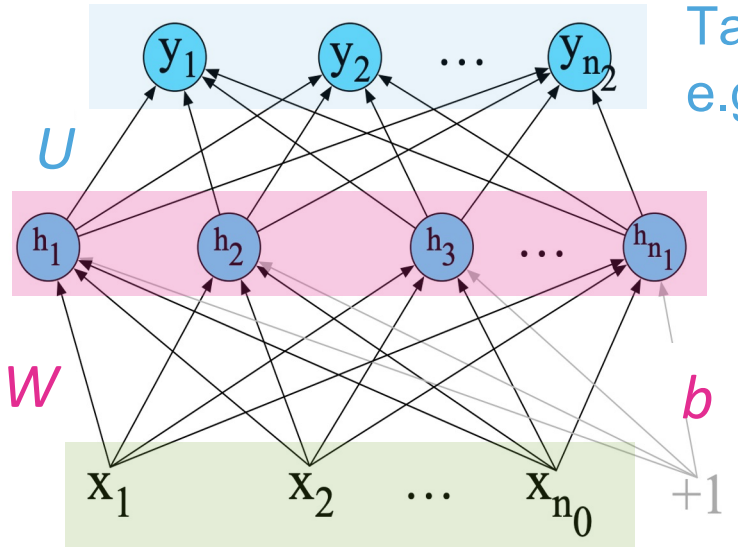




Or, in other words, the scalar $a := g(\sum_j w_j v_j)$.

Or, if we let $\mathbf{w} = [\ldots, w_j, \ldots]$ and $\mathbf{v} = [\ldots, v_j, \ldots]^T$, $a := g(\mathbf{wv})$

Feed-forward neural networks (FFNNs)



Task-specific outputs,
e.g., y_1 = P(ORG|w)

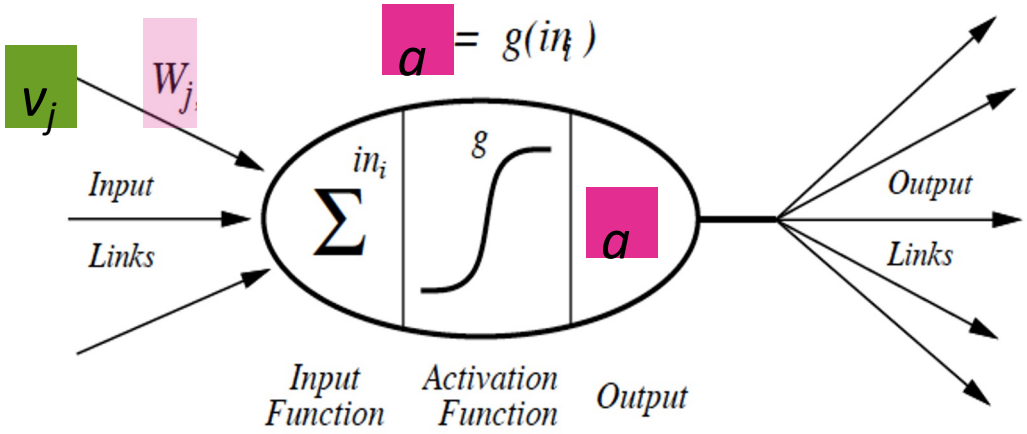
for word w, word2vec input
x = [x_1, ..., x_i, …, x_{n0}]

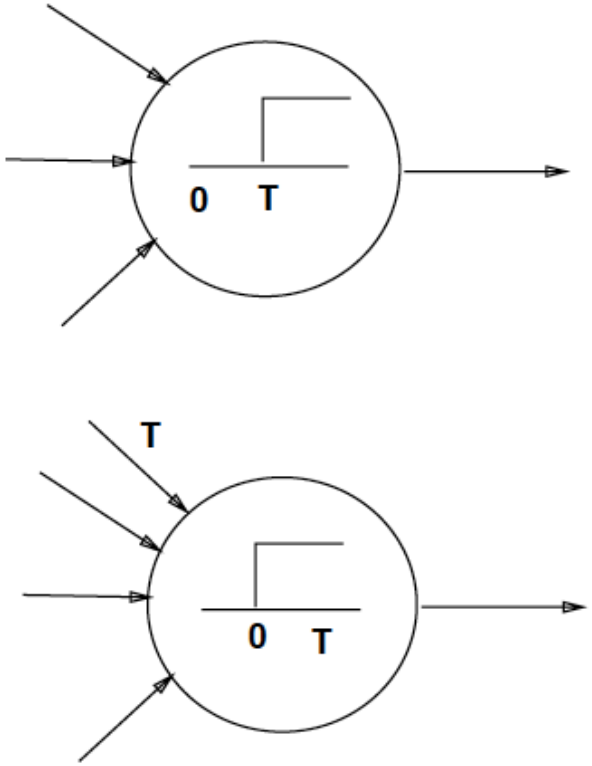
During training, all of the weights will be learned.

What is this ***b*** term and the extra input value that is always a 1?

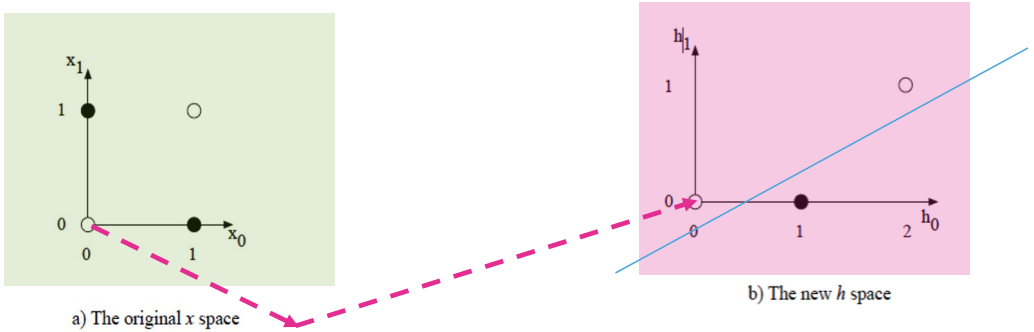
The *bias* term

Allows learning of the thresholds as though they were weights.

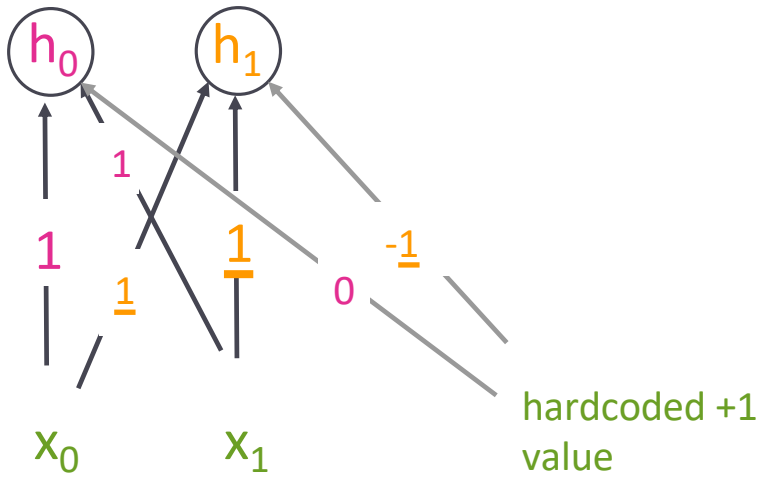




Our LOC/PER example



g = the ReLU activation function,
g(x) = max(x, 0)



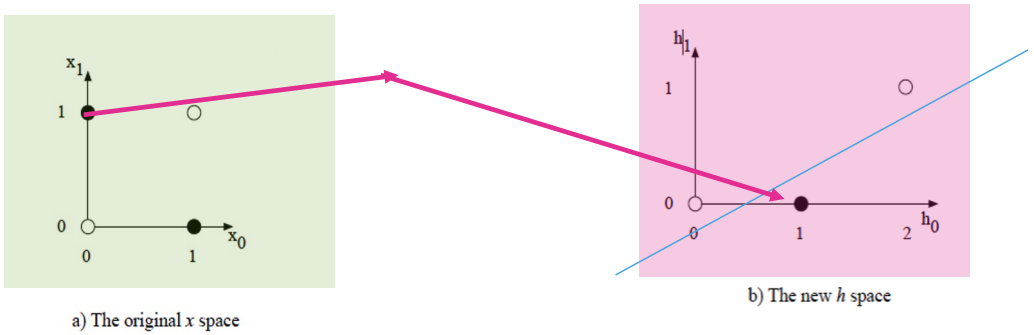
Computation for [0, 0], ○ = LOC:

$h_0 = g(\ 1(0) + 1(0) + 0(+1))$
$= g(0) = 0$

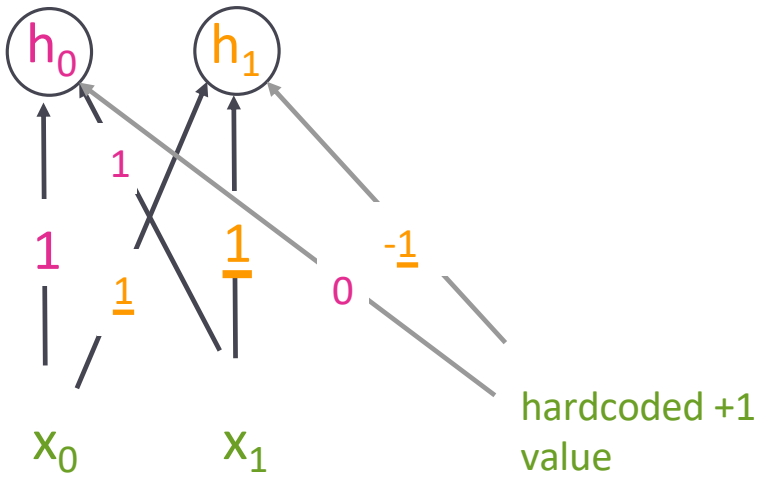
$h_1 = g(\ \underline{1}(0) + \underline{1}(0) + -1(+1))$
$= g(-1)$
$= 0$

Trick: This hardcoded +1 input allows us to treat the bias scalar b as just another weight.

Our LOC/PER example



g = the ReLU function,
g(x) = max(x, 0)

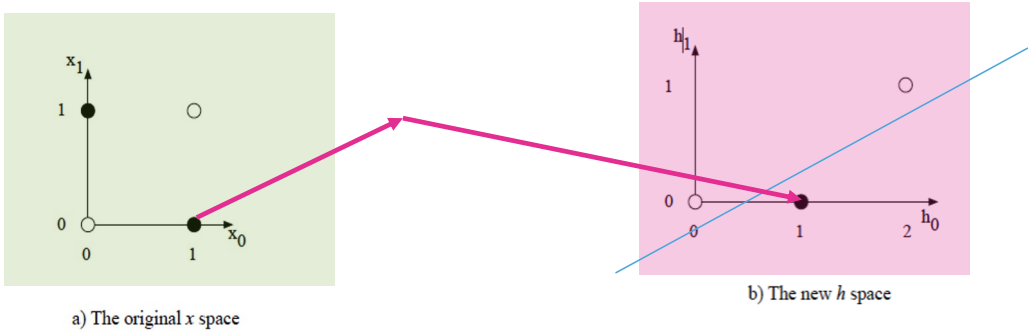


Computation for [0, 1], ⭕ = PER

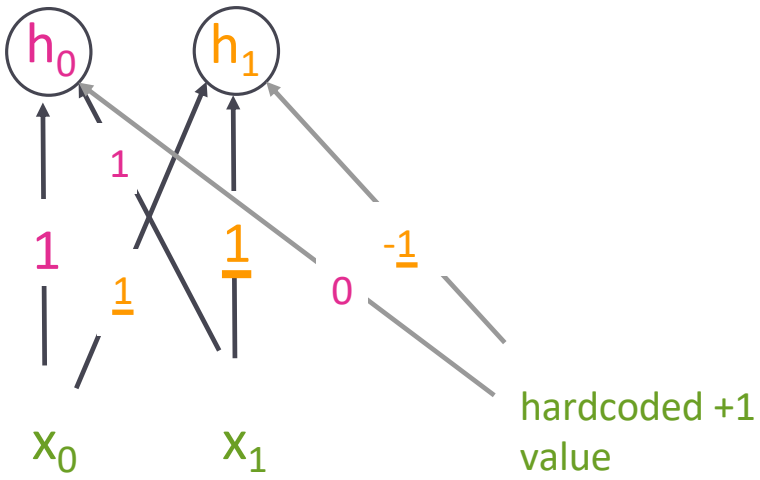
h_0=g(1(0) + 1(1) + 0(+1))
= g(1) = 1

h_1=g(1(0) + 1(1) + -1(+1))
= g(0)
= 0

Switch to matrix computations!



g = the ReLU function,
g(x) = max(x, 0)



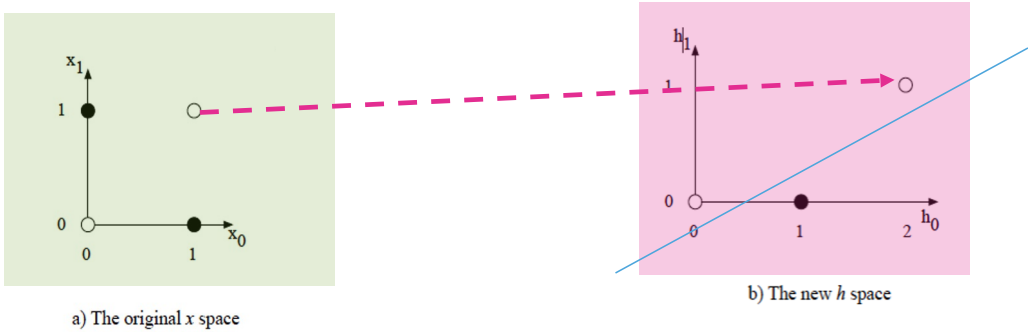
Computation for $[1, 0]^T$, 🔘 = PER

$h_0 = g([1, 1, 0]\ [1, 0, +1]^T)$
$= g(1) = 1$

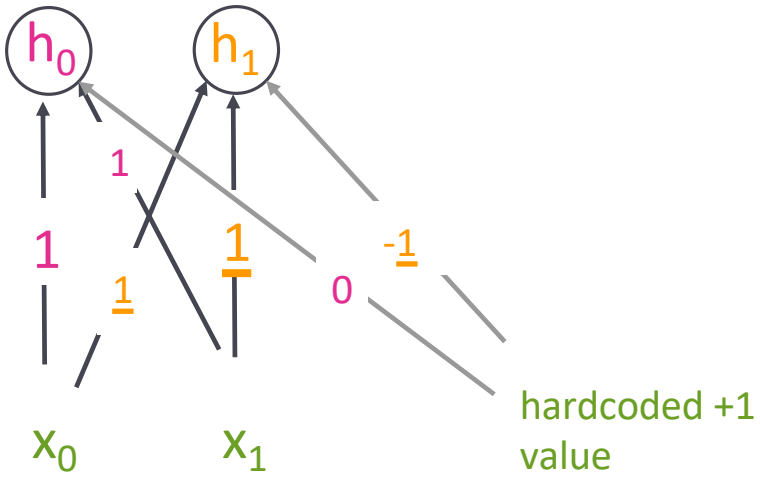
$h_1 = g([1, 1, -1]\ [1, 0, +1]^T)$
$= g(0)$
$= 0$

T = transpose: our word vectors are column vectors

Switch to matrix computations!



g = the ReLU function,
g(x) = max(x, 0)



Computation for $[1, 1]^T$, ○ = LOC

h_0 =g(1(1) + 1(1) + 0(+1))
= g(2) = 2

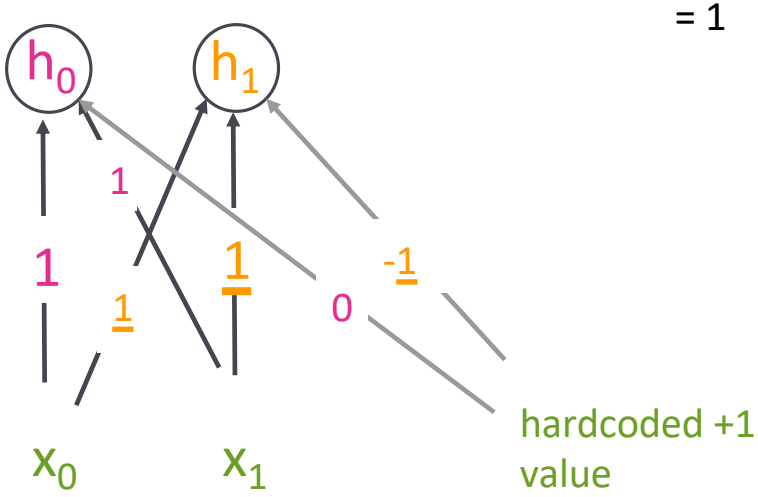
h_1=g(1(1) + 1(1) + -1(+1))
= g(1)
= 1

Matrix version (use Python packages, not for-loops)

$$g\left(\begin{bmatrix}1 & 1 & 0\\ 1 & 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 1\\ +1\end{bmatrix}\right) = g\left(\begin{bmatrix}2\\ 1\end{bmatrix}\right) = \begin{bmatrix}2\\ 1\end{bmatrix}$$

But, various computational packages want inputs as row vectors. Be flexible!

g = the ReLU function,
g(x) = max(x, 0)



Computation for $[1, 1]^T$, ○ = LOC

h_0 =g(1(1) + 1(1) + 0(+1))
= g(2) = 2

h_1=g(1(1) + 1(1) + -1(+1))
= g(1)
= 1

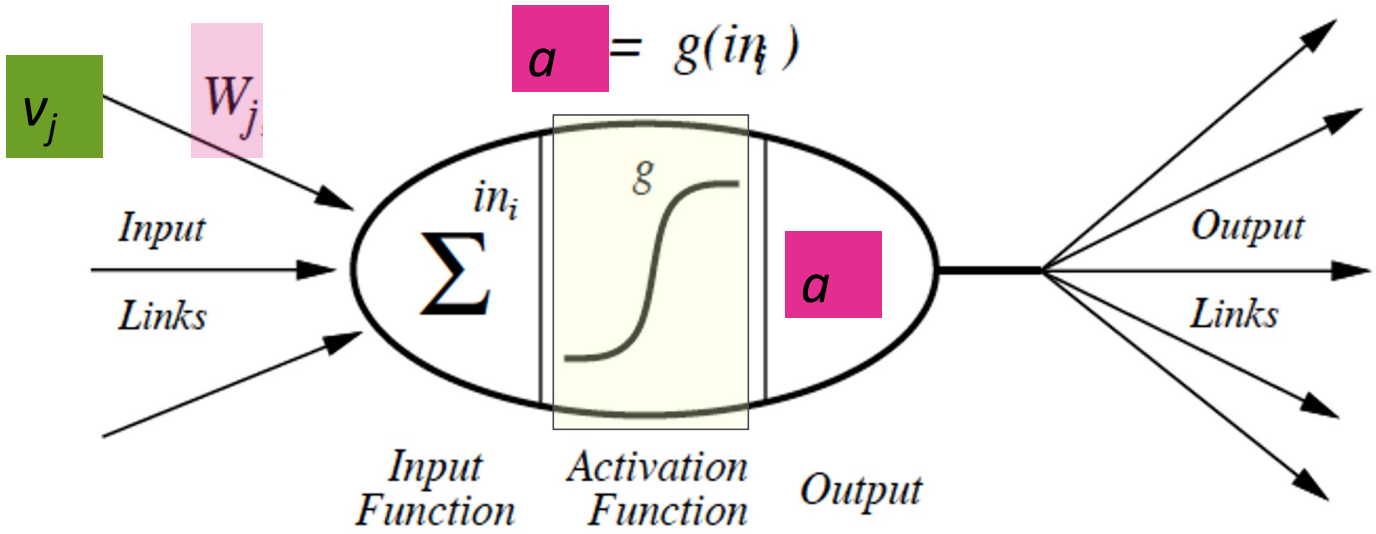
$$g\left(\begin{bmatrix}1 & 1 & 0\\ 1 & 1 & -1\end{bmatrix}\begin{bmatrix}1\\ 1\\ +1\end{bmatrix}\right) = g\left(\begin{bmatrix}2\\ 1\end{bmatrix}\right) = \begin{bmatrix}2\\ 1\end{bmatrix}$$

Is this equivalent?

row vector not column!

$$g\left(\begin{bmatrix}1 & 1 & +1\end{bmatrix}\begin{bmatrix}1 & 1\\ 1 & 1\\ 0 & -1\end{bmatrix}\right) = g\left(\begin{bmatrix}2 & 1\end{bmatrix}\right) = \begin{bmatrix}2\\ 1\end{bmatrix}^T$$

Alternate activation functions g( )



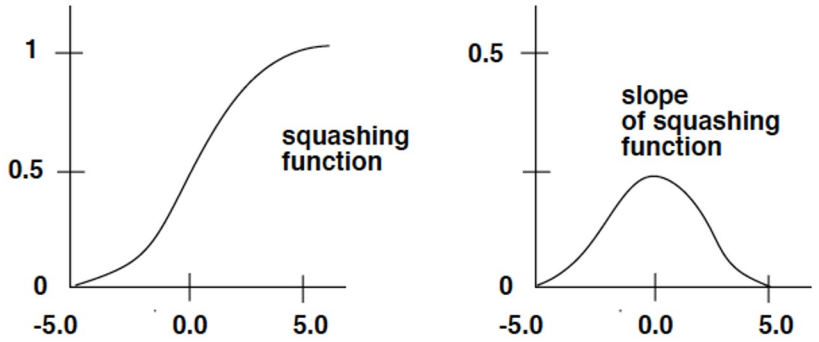
Or, in other words, the scalar $a := g(\sum_j w_j v_j)$.
Or, if we let $\mathbf{w} = [\ldots, w_j, \ldots]$ and $\mathbf{v} = [\ldots, v_j, \ldots]^T$, $a := g(\mathbf{wv})$

Sigmoid activation function

- Maps the output into the range **[0, 1]**
  - useful in squashing outliers toward 0 or 1

- Smoothly **differentiable:** the training algorithm requires a differentiable activation function

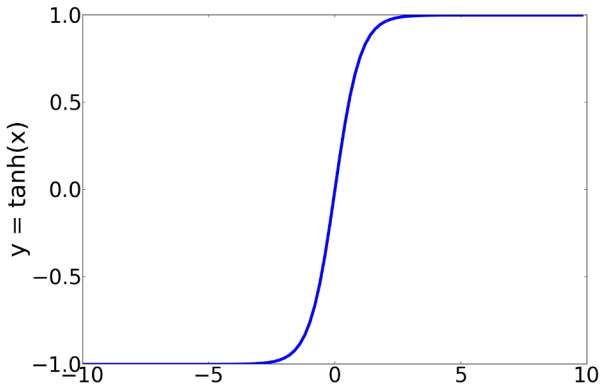
$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Other activation functions

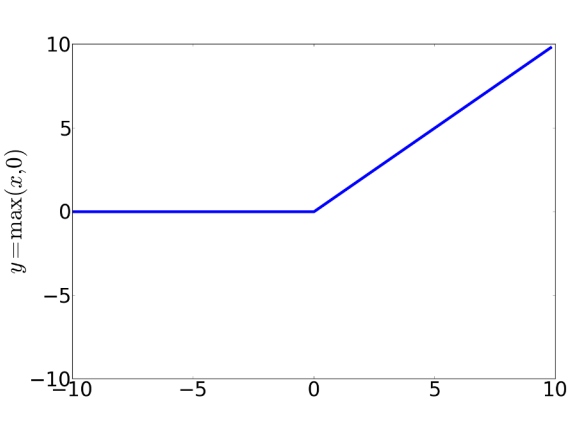
- ***tanh*** a variant of the sigmoid that ranges from -1 to +1

$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

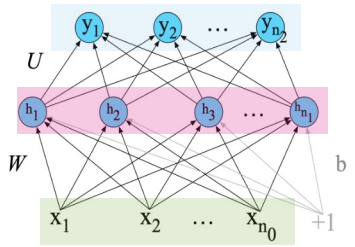


- ***ReLU*** rectified linear unit: equals x when x is positive, and 0 otherwise

$$y = max(x, 0)$$



Feedforward network



- Computation proceeds iteratively from one layer of units to the next.

- A feedforward network is a **multilayer** feedforward network in which the units are connected with **no cycles**.

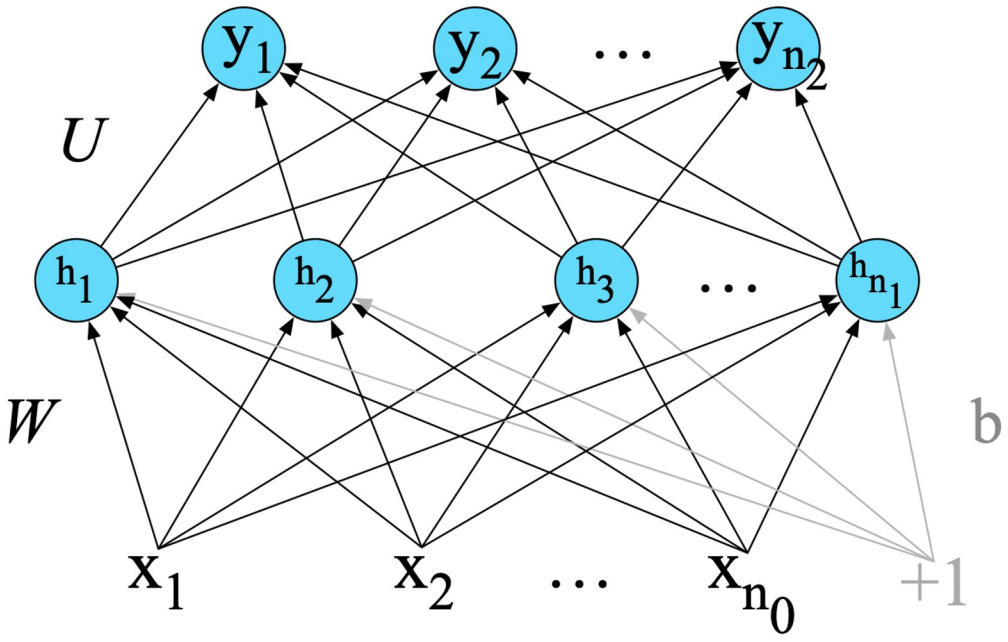
  I.e., the outputs from units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.

Feedforward network

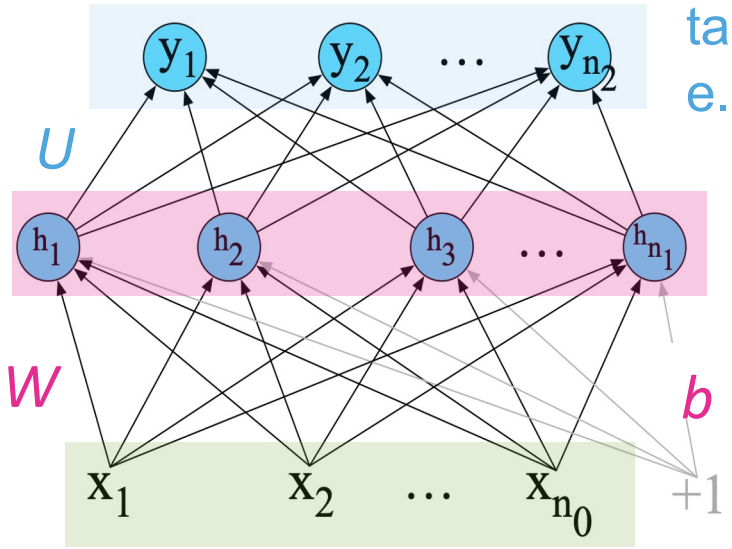
- Sometimes called **multi-layer perceptrons (MLPs)**; however, units in modern multilayer networks aren't perceptrons (linear)
  They are units with non-linear activation functions (e.g. tanh).

Fully-connected FFNN

Each unit in each layer takes as input the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers.



Back to the original picture! From x to h.



task specific outputs,
e.g., $y_1 = P(ORG|w)$

new vector rep
$\mathbf{h} = [h_1, .., h_{n1}]^T$

for word w, word2vec input
$\mathbf{x} = [x_1, ..., x_i, \ldots, x_{n0}]^T$

Matrix-multiply version: Given

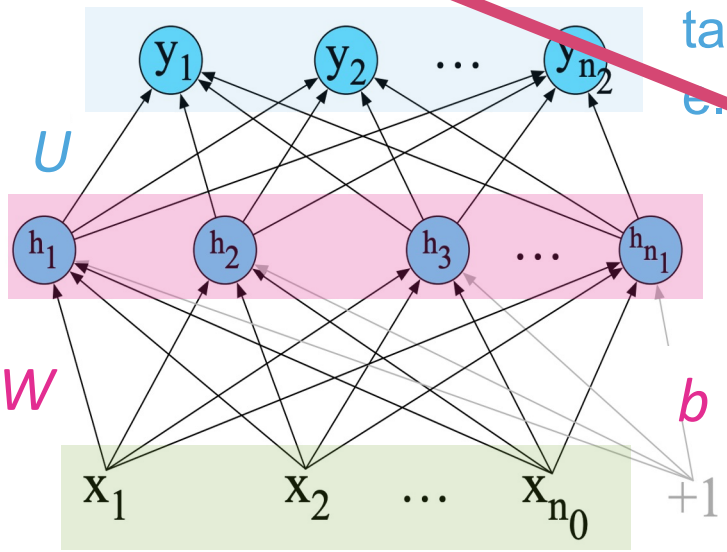
W $n_1 \times n_0$
$\mathbf{b}$ an $n_1 \times 1$ *bias* vector,
$\mathbf{x}$ $n_0 \times 1$
g an activation function that is applied elementwise to a vector.

$\mathbf{w}^{[i]}$:= row i of W is the weights applied to the x_js to help produce h_i.

$h_i = g(\mathbf{w}^{[i]}\mathbf{x} + b_i)$ or
$\mathbf{h} = g(W\mathbf{x} + \mathbf{b})$

$y_j = \text{softmax}(u^{[j]}\mathbf{h})$ or $\mathbf{y} = \text{softmax}(U\mathbf{h})$

Here, **x** is a column vector; the weights for computing h_i are row vectors.



task specific outputs,
e.g., $y_1 = P(ORG|w)$

new vector rep
$\mathbf{h} = [h_1, .., h_{n1}]^T$

a word2vec input
$\mathbf{x} = [x_1, ..., x_i, \ldots, x_{n0}]^T$

Matrix-multiply version: Given

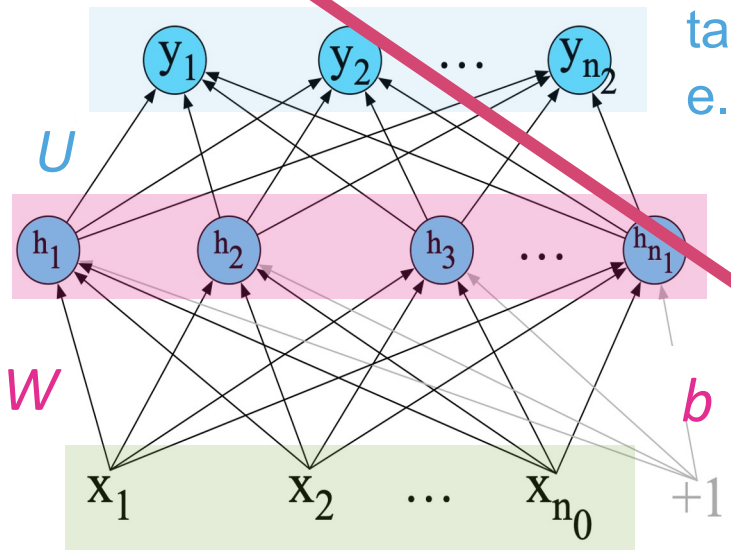
W $n_1 \times n_0$
$\mathbf{b}$ an $n_1 \times 1$ *bias* vector,
$\mathbf{x}$ $n_0 \times 1$
g an activation function that is applied elementwise to a vector.

$\mathbf{w}^{[i]}$:= row i of W is the weights applied to the x_js to help produce h_i.

$h_i = g(\mathbf{w}^{[i]}\mathbf{x} + b_i)$ or
$\mathbf{h} = g(W\mathbf{x} + \mathbf{b})$

$y_j = \text{softmax}(u^{[j]}\mathbf{h})$ or $\mathbf{y} = \text{softmax}(U\mathbf{h})$

Here, h is a column vector; the weights for computing y_j are row vectors.



task specific outputs,
e.g., y_1 = P(ORG|w)

new vector rep
$\mathbf{h} = [h_1, .., h_{n1}]^T$

word2vec vector as input
$\mathbf{x} = [x_1, ..., x_i, \ldots, x_{n0}]^T$

Matrix-multiply version: Given

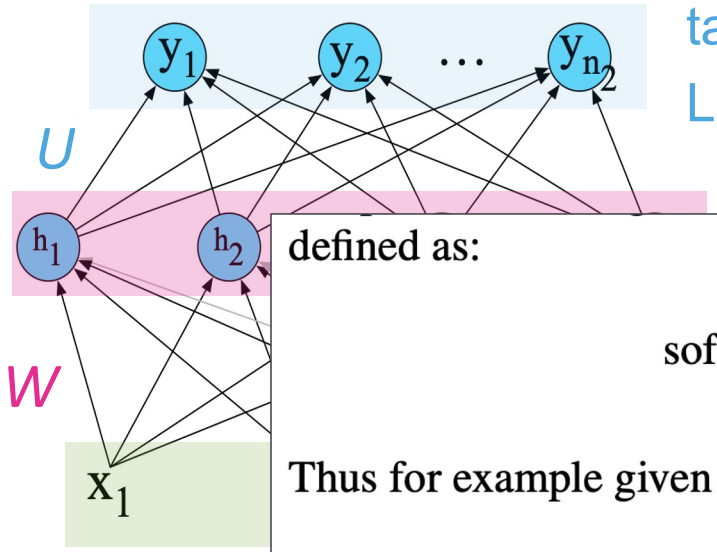
W $n_1 \times n_0$
b an $n_1 \times 1$ *bias* vector,
x $n_0 \times 1$
g an activation function that is applied elementwise to a vector.

$\mathbf{w}^{[i]}$:= row i of W is the weights applied to the x_js to help produce h_i.

$h_i = g(\mathbf{w}^{[i]}\mathbf{x} + b_i)$ or
$\mathbf{h} = g(W\mathbf{x} + \mathbf{b})$

y_j = softmax($u^{[j]}\mathbf{h}$) or **y** = softmax(U**h**)

From h to y=softmax(Uh): how do we make the y_i's into a probability distribution?



task specific outputs, e.g., y_1 = P(ORG|w).

Let **z** be **Uh**. Then, $y_i = \text{softmax}_z(z_i)$.

for word w,

x = [x_1, ..., x

defined as:

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d}\exp(\mathbf{z}_j)} \quad 1 \le i \le d \tag{7.9}$$

Thus for example given a vector

$$\mathbf{z} = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1], \tag{7.10}$$

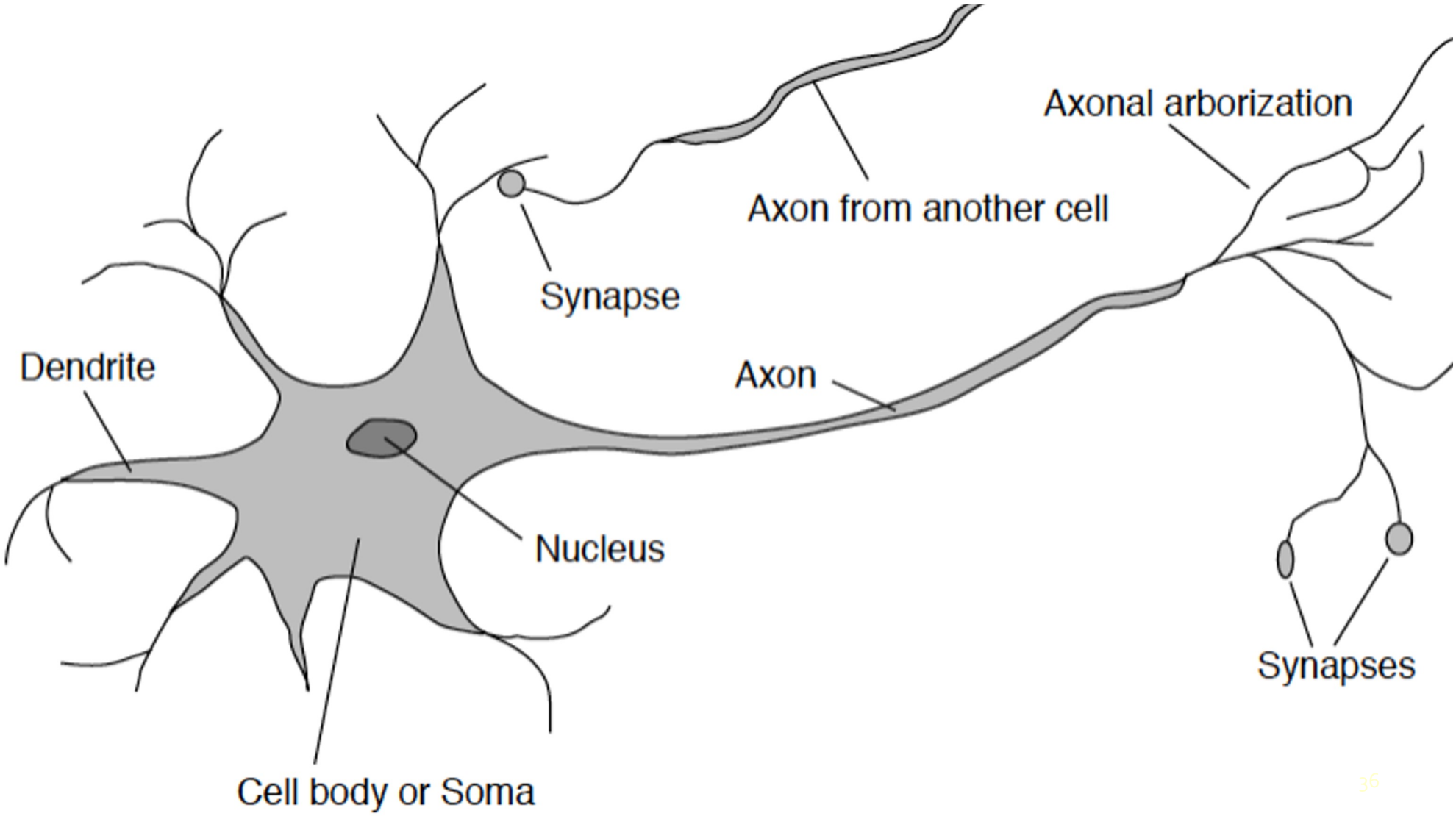
the softmax function will normalize it to a probability distribution (shown rounded):

$$\text{softmax}(\mathbf{z}) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010] \tag{7.11}$$

Historical note

- Rich history, starting in the early 1940's
  - McCulloch and Pitts, 1943
- Two views of neural nets
  - Models of the human brain
  - Representations of complex functions

Much progress on both fronts!!!


Axonal arborization
Axon from another cell
Synapse
Dendrite
Axon
Nucleus
Synapses
Cell body or Soma

So where do the weights come from?

Next class…