Queen Mary
University of London

ECS763 Natural Language Processing

Unit 6: Introduction to Deep Learning for NLP

Lecturer: Julian Hough
School of Electronic Engineering and Computer Science

OUTLINE

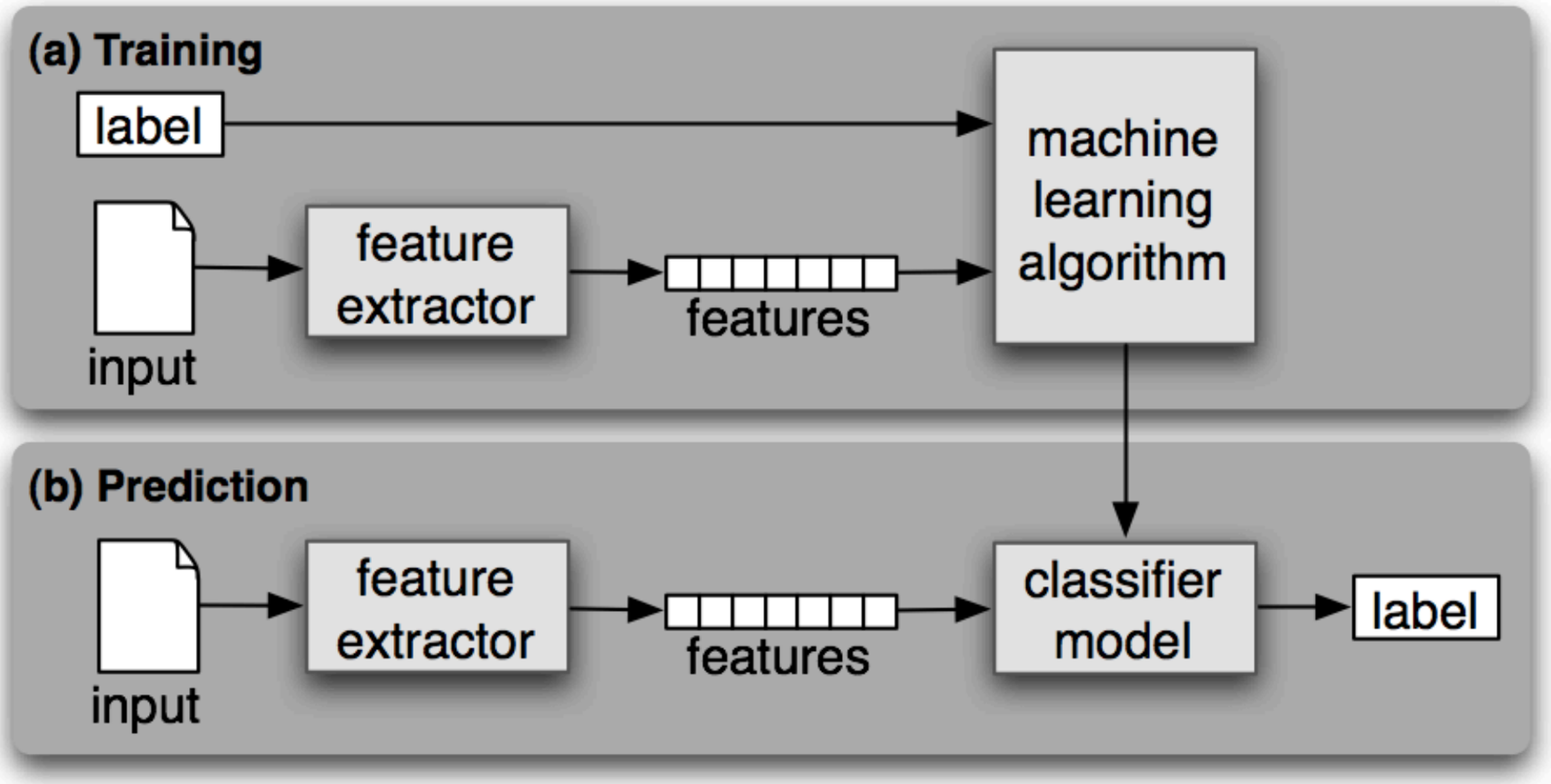
1) The renaissance of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

OUTLINE

1) The renaissance of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

‘Classical’ vs deep neural models

- The focus in this module so far has been on **‘classical’ machine learning** models being used for NLP applications. E.g.:
  - Naive Bayes, Logistic Regression and SVMs for text classification.
  - Ngram models for language/sequence modelling.
  - HMMs and CRFs for sequence classification.



‘Classical’ vs deep neural models

- Motivation away from classical approach 1: While parameters/ weights are learned automatically, all of the traditional approaches require the following:
  - (1) **Pre-processing** of raw data
  - (2) **Feature extraction engineering** (including weighting/ smoothing technique, minimum document frequency for vocab/ feature dictionary definition)
  - (3) **Hyper-parameter optimisation** (e.g. the order for ngram models, value of k for smoothing, C1, C2 for CRFs, etc.)

- All three take considerable human effort and (1) and (2) take quite a lot of linguistic expertise. What if we could automate most of (2) and to some extent (1), and instead focus on designing hyper-parameters of the model to best facilitate learning?

‘Classical’ vs deep neural models

- Motivation away from classical approach 2: Symbolic versions of meaning representation:
  - Map words -> syntax -> semantics (e.g. First Order Logic)
  - None of these layers had an elegant approach to **meaning similarity**.
  - They take massive amounts of **human labour** (designing treebanks, building parsers, defining hard-coded semantics). Scalability hard.
  - We show that **distributional** approaches go someway to modelling this by similarity measures and modelling words as vectors.

- To get a more use-based, empirical and distributional notion of word meaning similarity, and to exploit **massive unstructured text data** to get meaning more automatically, in recent years, the dominant paradigm has been **deep learning.**

  - **Deep learning** is the use of **multi-layered neural network models** for machine learning some task with a given optimisation function.

‘Classical’ vs deep neural models

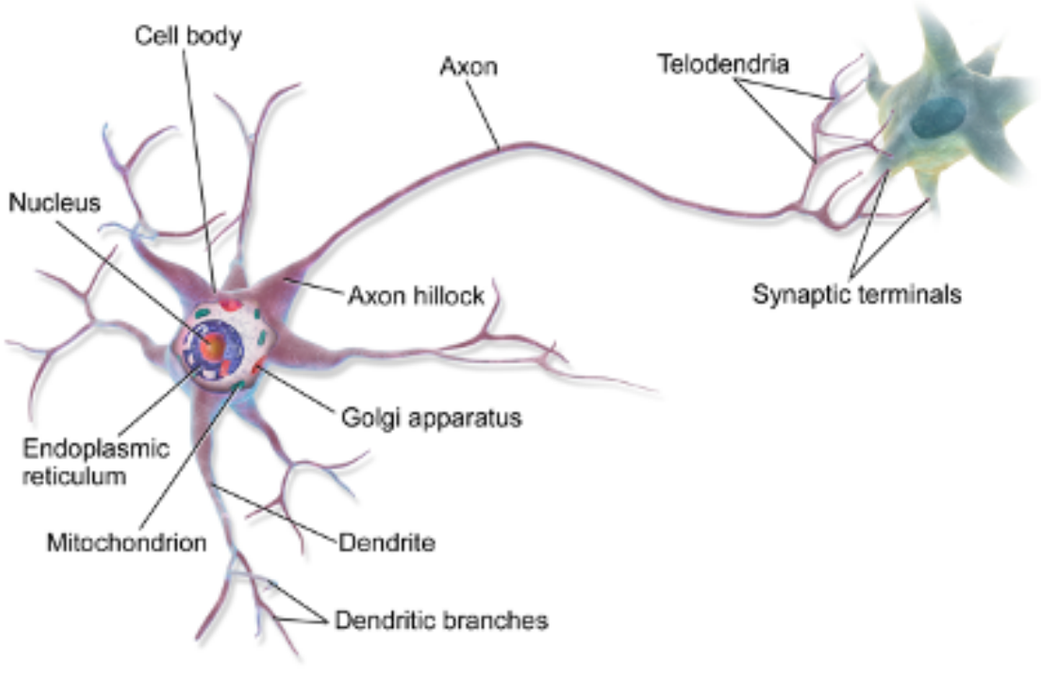
- **History**: Neural nets first emerged as a popular paradigm in the 1980s/1990s with the emergence of the **Parallel Distributed Processing (PDP)** paradigm of AI and Cognitive Science

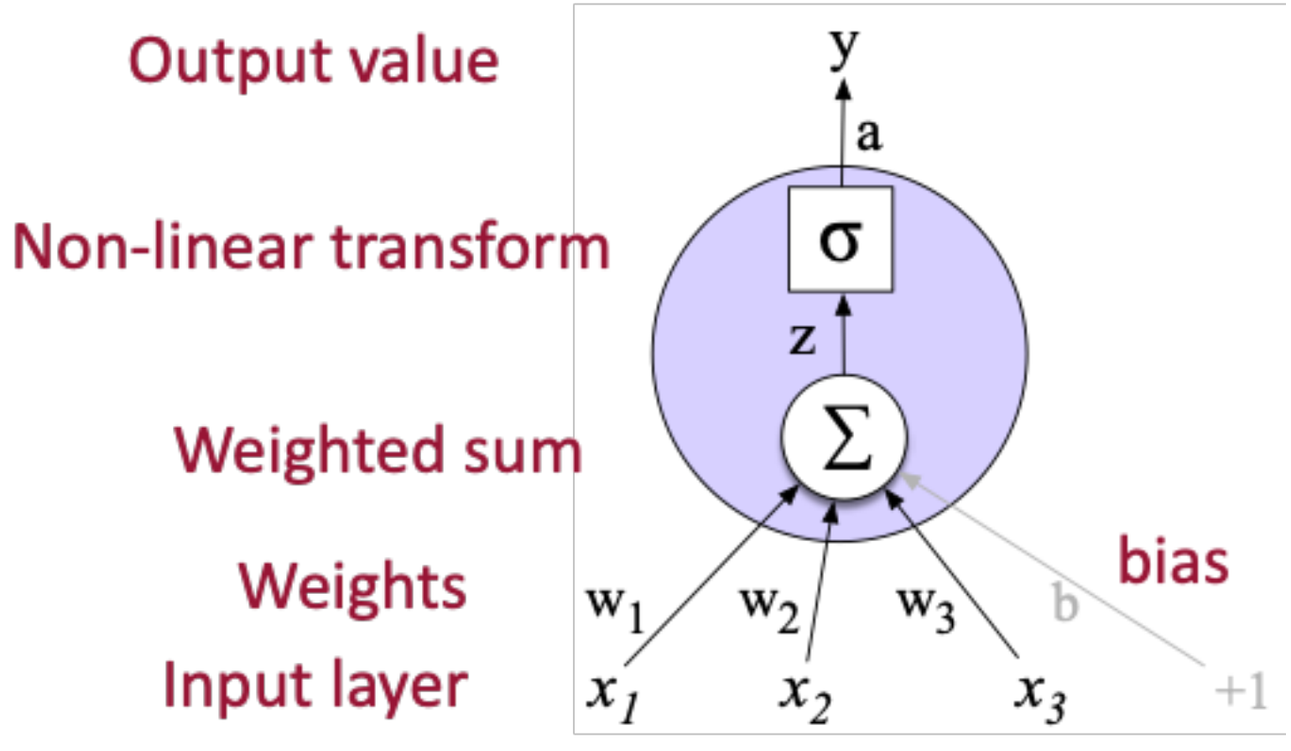
- **Rules, schmules!** The move was away from classical rule-based, symbol-orientated approached to linguistics and language acquisition and towards sub-symbolic models which use lots of **small real-valued functions (`neurons`)** to learn patterns in linguistic data.

- Originally used for fairly small tasks with a low computational cost, according to the limits of the computing power (and data) available.

‘Classical’ vs deep neural models

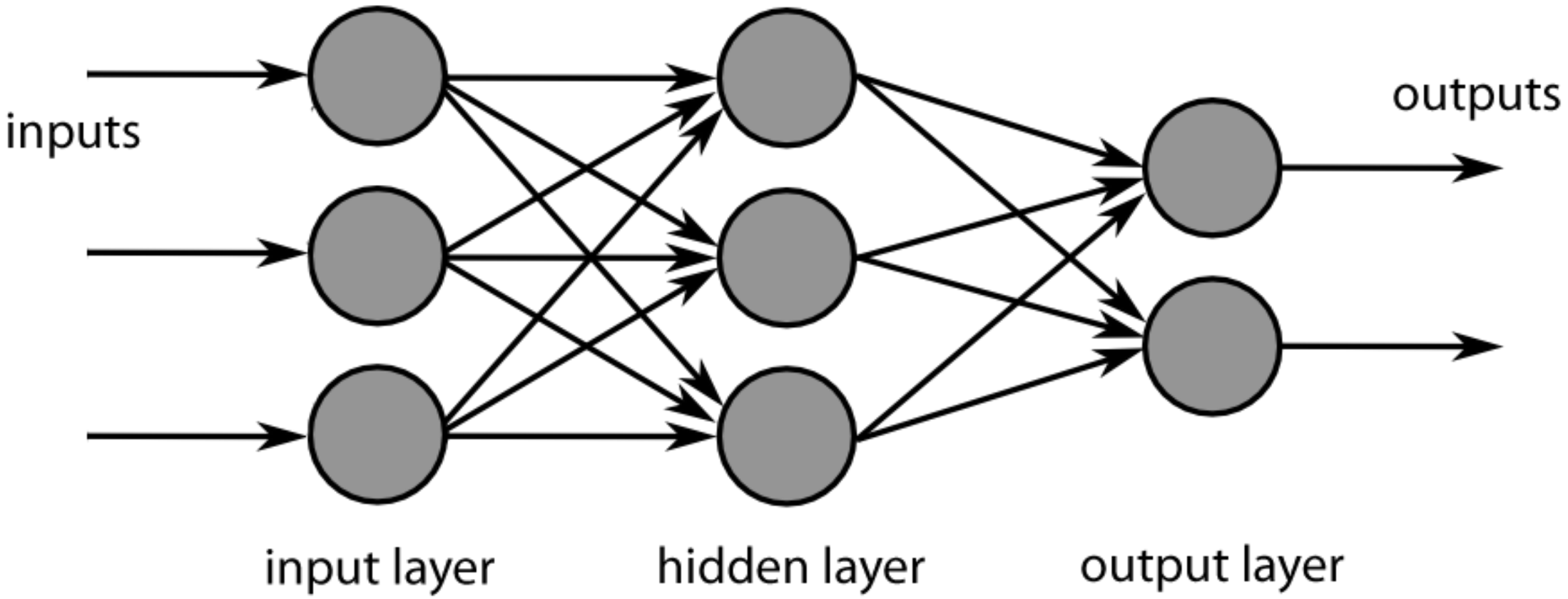
- Modelling the non-linear activation of a neuron in the brain, PDP posited a **“neuron”** or **neural unit** in their models.
- Units worked together in **layers of units** to solve complex cognitive problems.





‘Classical’ vs deep neural models

- **Feedforward neural network** (not very ‘deep’ at this point)



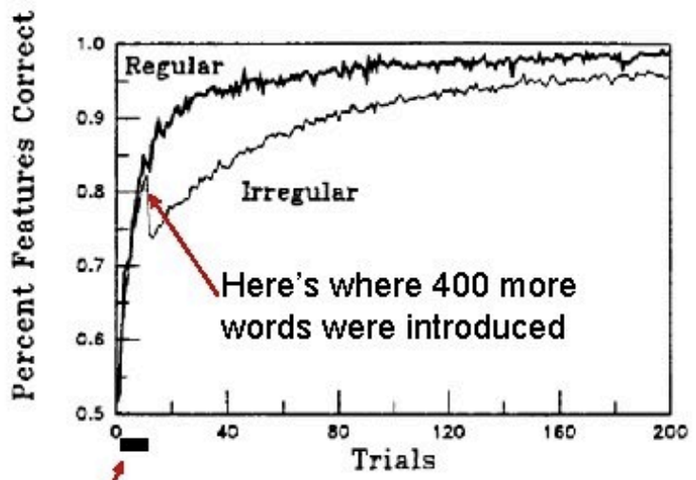
‘Classical’ vs deep neural models

- Early result in PDP: McClelland and Rumelhart (1986) used a simple **feedforward neural network** to simulate the acquisition of the past tense of English e.g.modelling the acquisition of regular vs. irregular forms to be learned, including ‘over-regularization’ errors’:
    - pick -> picked (correct **regular**, i.e. +ed ‘rule’)
    - have -> had (correct **irregular**, i.e. *ad hoc* forms)
    - have -> haved (**incorrect over-regularziation** of +ed ‘rule’)

Over-regularization errors in the RM network

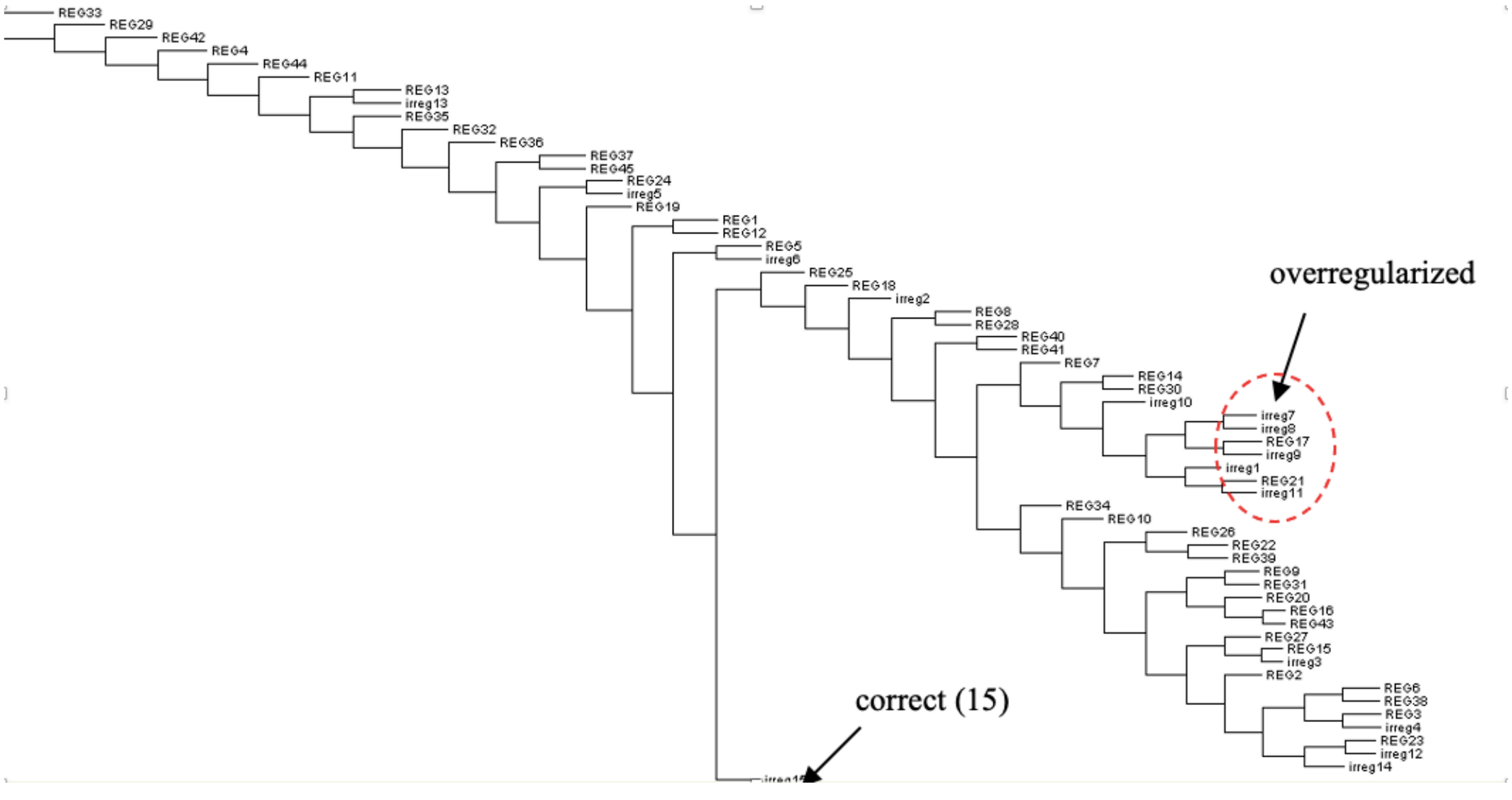
Most frequent past tenses in English:

- Felt
- Had
- Made
- Got
- Gave
- Took
- Came
- Went
- Looked
- Needed



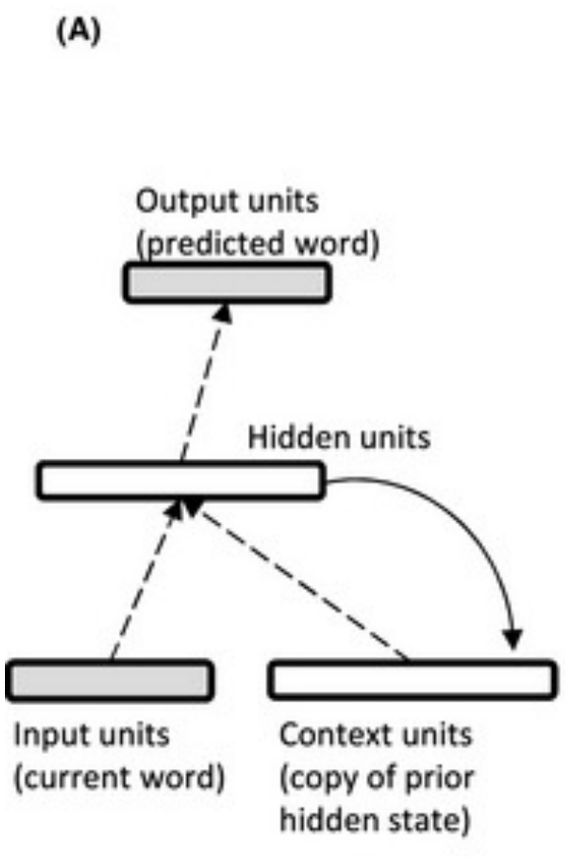
‘Classical’ vs deep neural models

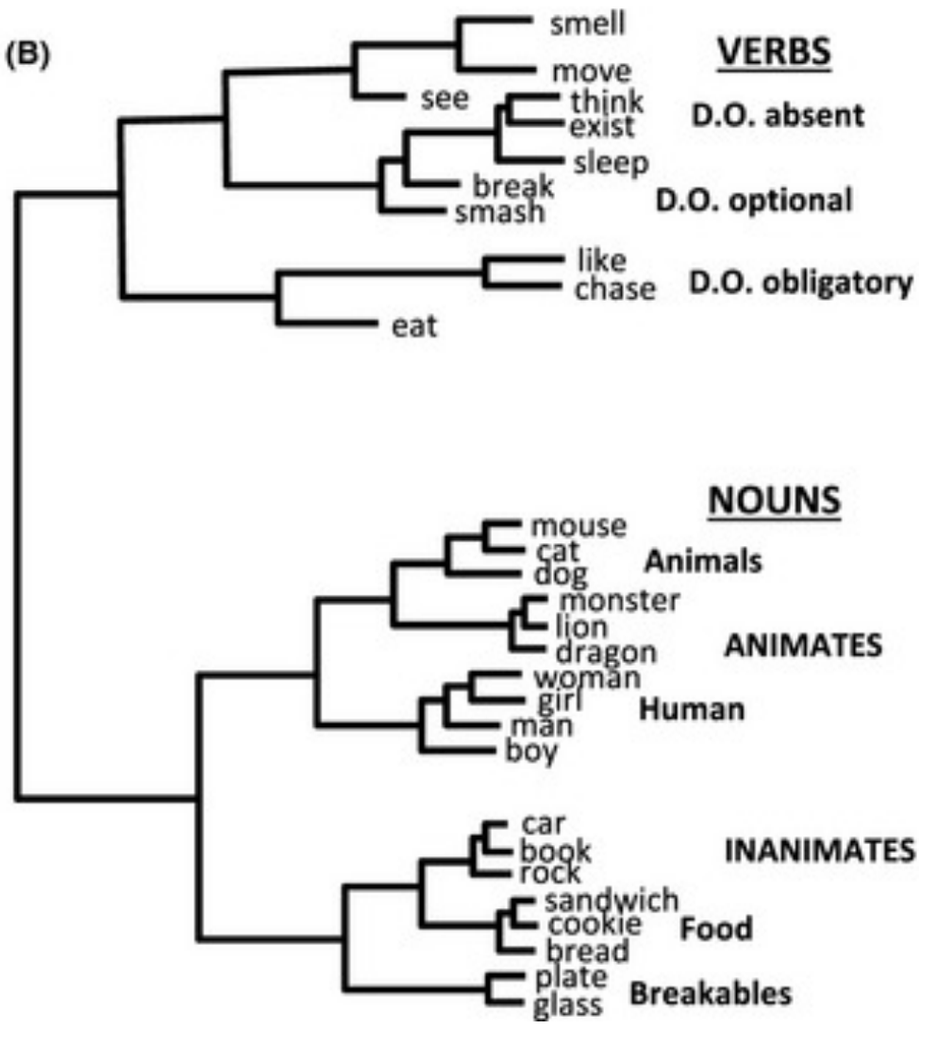
- Learning results on problems usually employing symbolic rules were surprisingly good with early neural nets.
- Looking at **activation patterns** within the network shed light on why they were doing this, e.g. PCA plots of the hidden layer activations. E.g. from the software *tlearn* (Elman et al 1996):



‘Classical’ vs deep neural models

- Elman (1990) designed a **recurrent neural net** predicting **the current word from the previous context words**, to show how verbs and nouns (and subtypes within those according to semantic or syntactic properties) clustered in similar places in PCA plots:





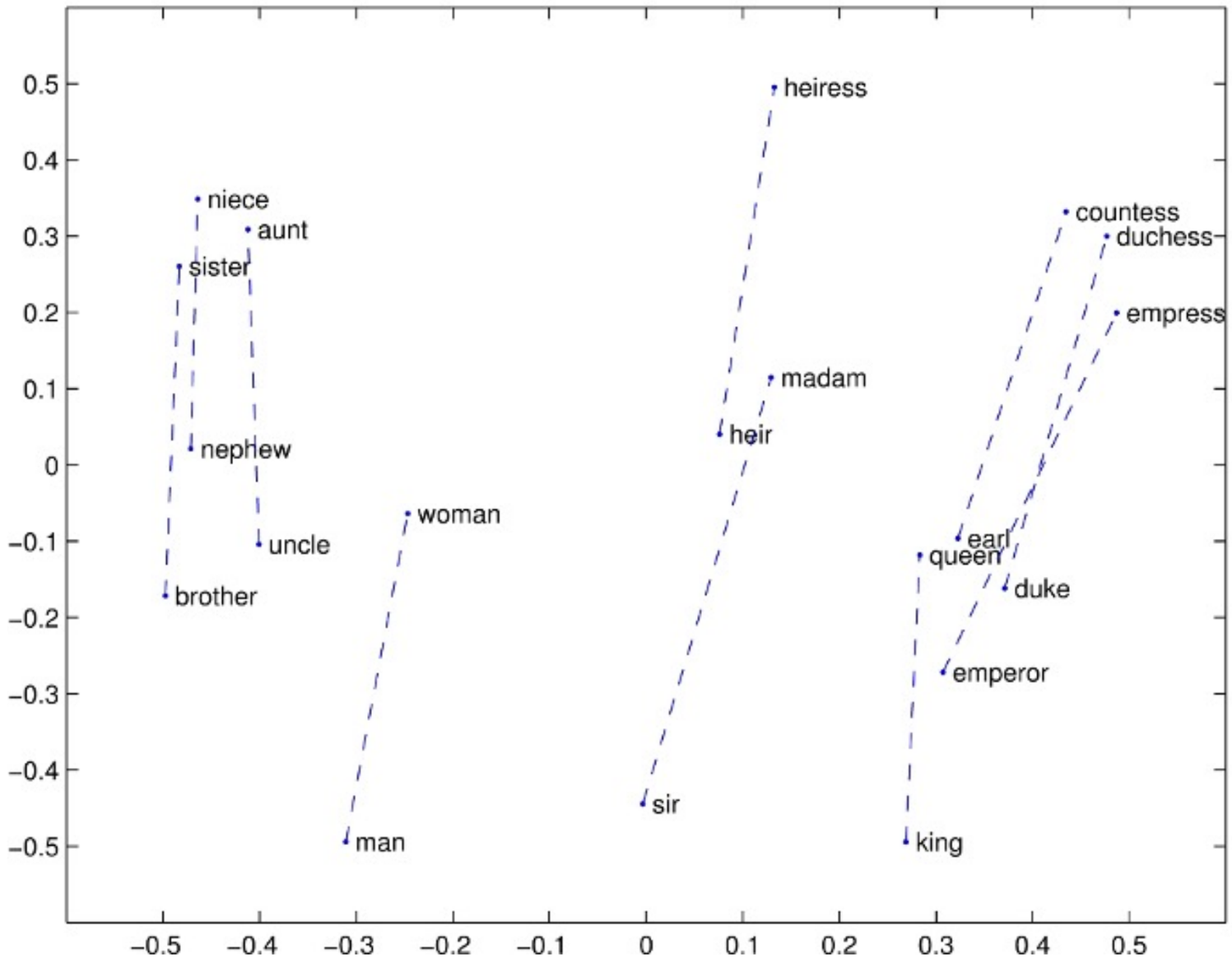
‘Classical’ vs deep neural models

- After a mild neural net ‘winter’, the early results on ‘toy’ domains paved the way for the **resurgence** of **deep (multi-layered)** nets for use in language applications in the 2010s when the computing power became available to make use of **big data** (e.g. the massive amounts of text on the internet) without hand-annotated word meanings.

- The new nets brought back into focus two ideas:
  - **Automatic associative learning** without other symbolic representations other than the data itself (**self-supervised**)
  - **Real-valued vector representations of words**, as learned by neural nets, i.e. **word embeddings**. As with the old nets these are usually derived from the activation layers of a network (**hidden layers**).

‘Classical’ vs deep neural models

- Early results were using recurrent neural nets for state-of-the-art **language modelling** (Mikolov 2011) and word2vec **word meaning representations (embeddings)** (Mikolov et al 2013).



- How do we get there?

OUTLINE

1) The renaissance of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

Logistic Regression

- **Recap** on classification with logistic regression:

$$\hat{y} = \underset{y \in Y}{\text{argmax}}\ p(y \mid x)$$

- Where x is an instance of data (e.g. a document), represented as a vector, representing, e.g. a bag of words count for that document where the element at each index x_n corresponds to the count for a particular word in the doc:

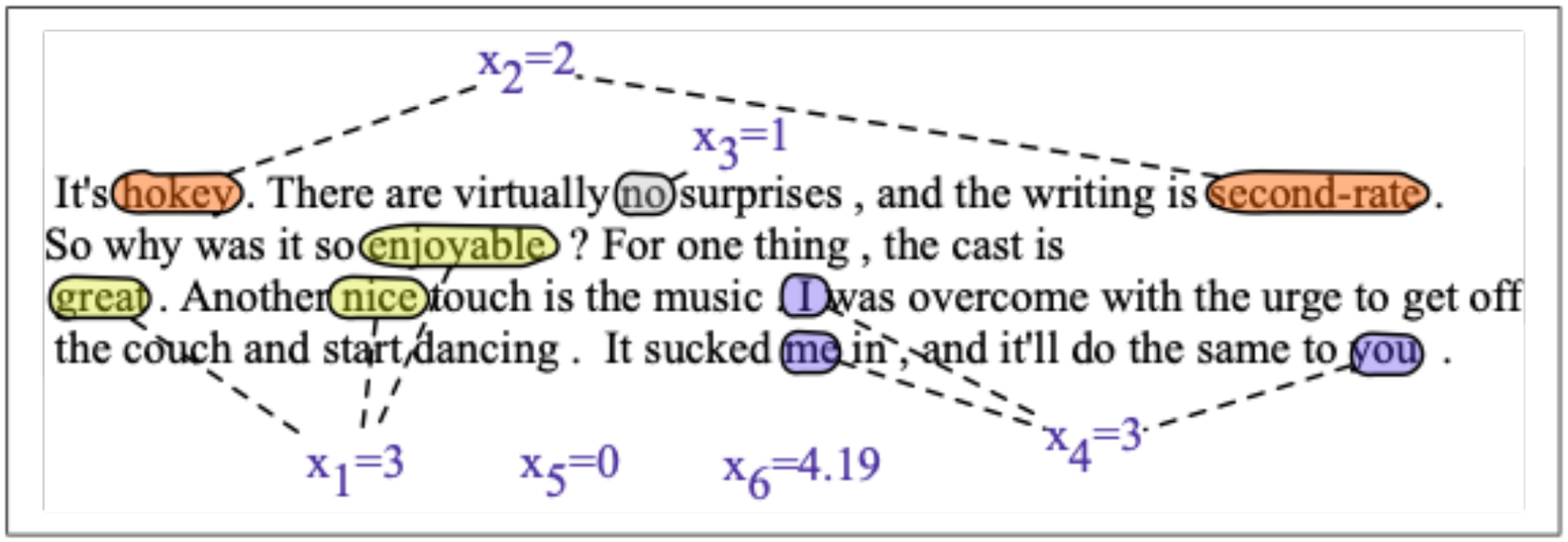
$$x = [0, 3, 0, 0, 0, \ldots, 2, 0]$$

I *Mo* *Bill* *Mary* *John* ⋮ *love* *like*

- And p(ylx) is a scalar between [0,1]

Logistic Regression

- Example: Sentiment analysis (positive + vs - negative)
- Text instance:



- Feature values x:

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if } \text{“no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

Logistic Regression

- We have some weights (a vector *w*) and a bias weight *b* learned through training which are the coefficients applied to the inputs:

- e.g. *w* = [2.5, -5.0, -1.2, 0.5, ….., 2.0, 0.7] and *b* = 0.1

- In binary LR we have the activation function Z, just the linear combination (or dot product) of the weights with the inputs + bias:

$$Z = (\sum_{i=1}^{n} w_i x_i) + b$$

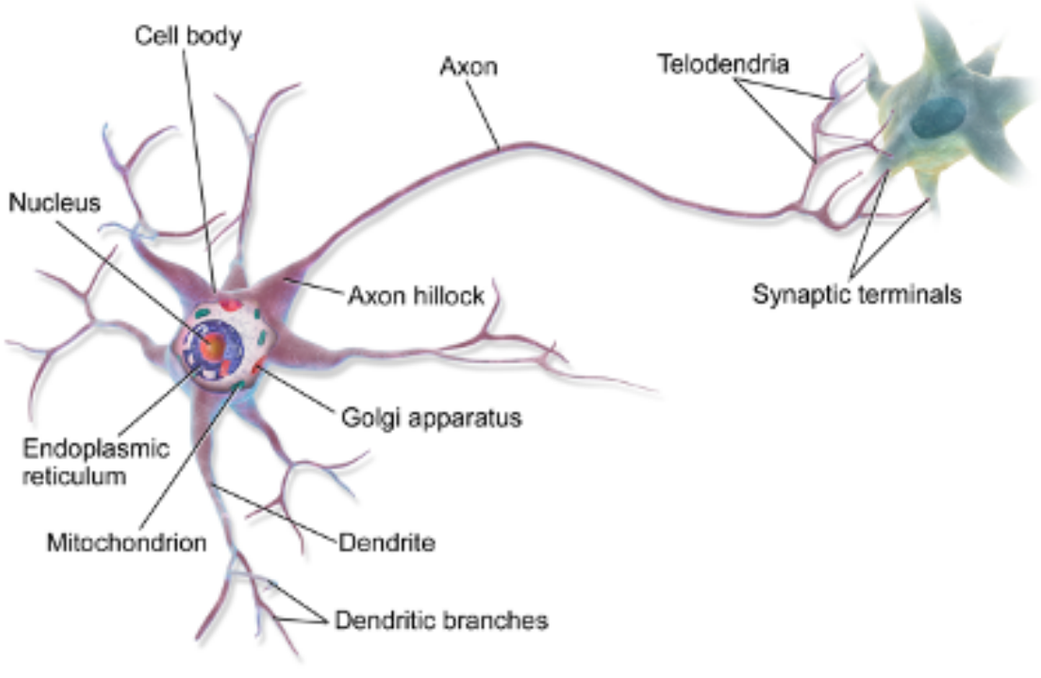
$$Z = w \cdot x + b$$

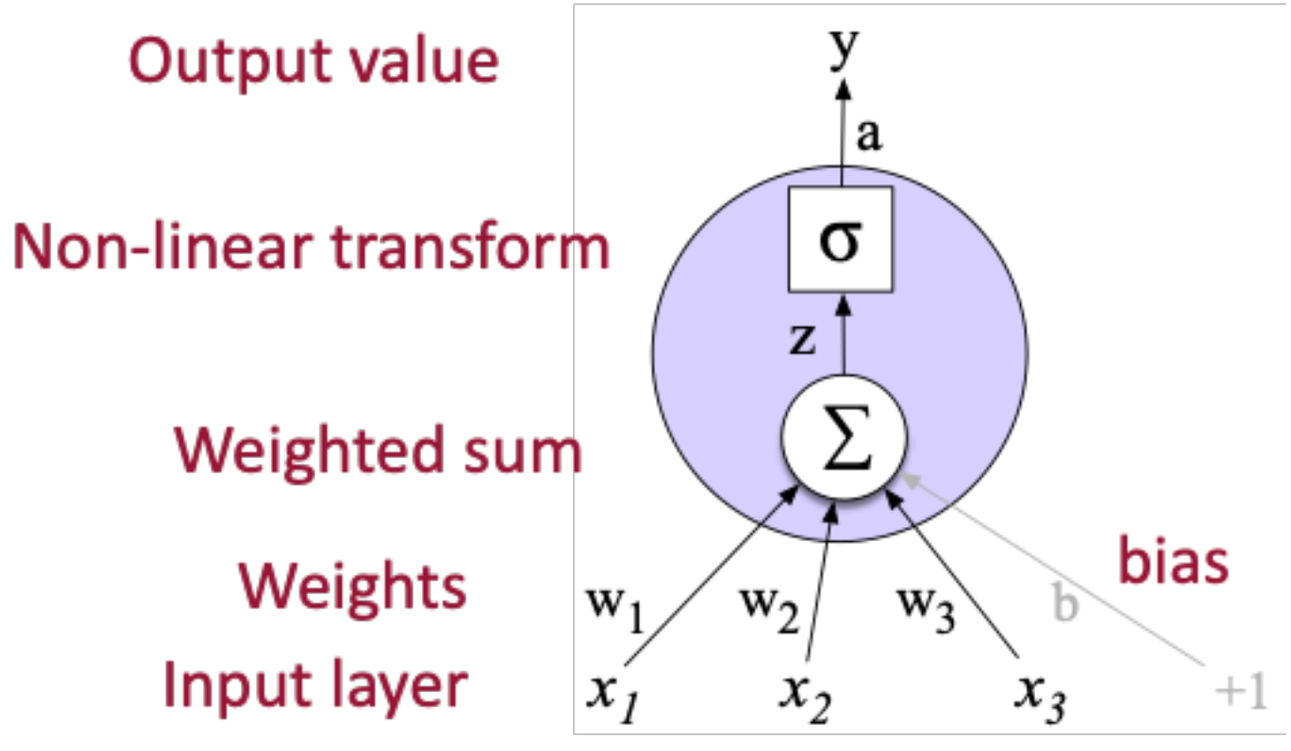
- To make it **proper probability value** use a **sigmoid** function over *z* to make it between [0,1]*:*

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

Logistic Regression to Neural Nets

- We can think of the non-linear sigmoid function as something like the non-linear activation of a neuron in the brain.
- This formulation of logistic regression gives rise to the basic building block of neural network architectures, the **neural unit**:

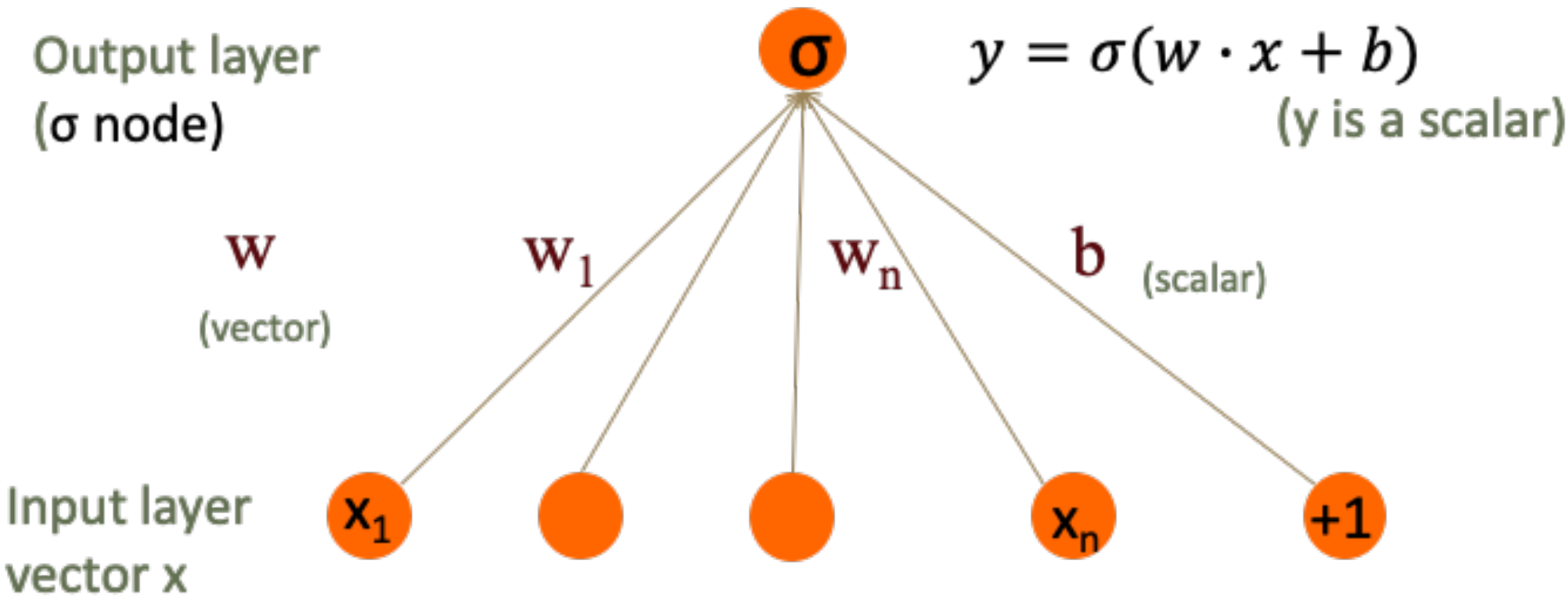




Logistic Regression to Neural Nets

Binary Logistic Regression as a 1-layer Network

(we don't count the input layer in counting layers!)



Logistic Regression to Neural Nets

- What if we want to go beyond binary classification to multi-class classification?

- We need **multinomial logistic regression.**

- The probability of everything must still sum to 1

```
P(positive|doc) + P(negative|doc) + P(neutral|doc) = 1
```

.

Logistic Regression to Neural Nets

- For multinomial LR we use a generalization of the sigmoid, **softmax:**
  - Takes a vector $z = [z_1, z_2 \ldots, z_k]$ of *k* arbitrary values (corresponding to *k* classes)
  - Outputs a probability distribution over all *k* classes
  - each value in the range [0,1].
  - all the values summing to 1.
- The probability estimation for a given class *i* is:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k}\exp(z_j)} 1 \leq i \leq k$$

- The denominator $\sum_{j=1}^{k}\exp(z_j)$ is used to normalise all the values into probabilities in a proper distribution:

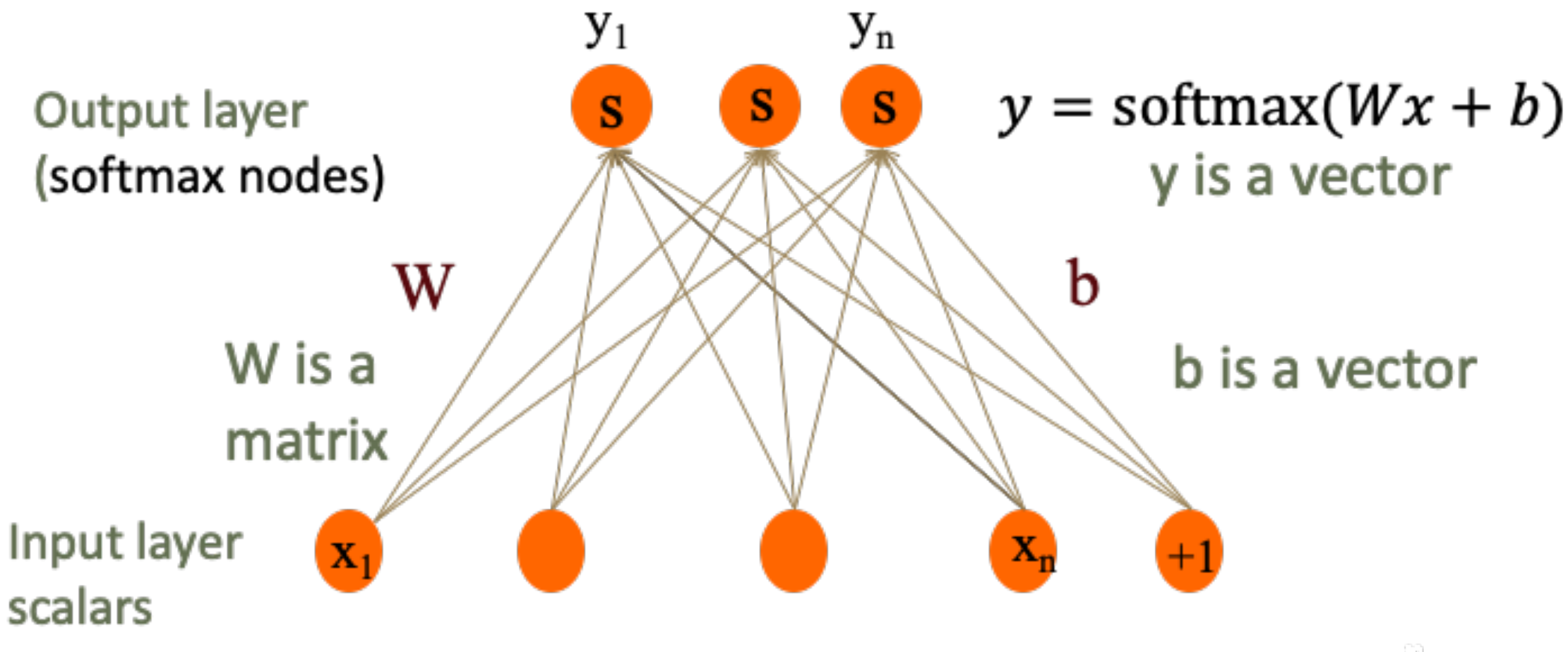
$$\text{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{j=1}^{k}\exp(z_j)}, \frac{\exp(z_2)}{\sum_{j=1}^{k}\exp(z_j)}, \ldots, \frac{\exp(z_k)}{\sum_{j=1}^{k}\exp(z_j)}\right]$$

  - It's not a hard max, because it keeps the other non-maximal values around, though exponential function gives more weight to highest z_i.

Logistic Regression to Neural Nets

Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



Logistic Regression to Neural Nets

Features in binary versus multinomial logistic regression

Binary: positive weight → y=1 neg weight → y=0

$$x_5 = \begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases} \qquad w_5 = 3.0$$

Multinominal: separate weights for each class:

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	−5.3

OUTLINE

1) The renaissance of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

Learning the weights

Where do the *w*'s come from?

Supervised classification:

- We know the correct label *y* (either 0 or 1) for each *x*.
- But what the system produces is an estimate, $\hat{y}$
- We want to set *w* and *b* to minimize the distance between our estimate $\hat{y}^{(i)}$ and the true $y^{(i)}$.
- We need a distance estimator: a **loss function** or a **cost function**
- We need an **optimization algorithm** to update *w* and *b* to minimize the loss until good enough/convergence.

Learning the weights

Where do the *w*'s come from?

- Loss function: **cross-entropy**
- Optimization algorithm: **stochastic gradient descent**

Cross-entropy loss

- Goal: maximize probability of the correct label $p(y|x)$ through the exponential function:
- Maximize:

$$p(y|x) = \hat{y}^y (1-\hat{y})^{1-y}$$

- Now flip sign to turn this into a loss: something to minimize
- **Cross-entropy loss** (because is formula for cross-entropy($\hat{y}$,y)
- Minimize:

$$L_{\text{CE}}(\hat{y}, y) = -\log p(y|x) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})]$$

- Or, plugging in definition of $\hat{y}$:

$$L_{CE}(\hat{y}, y) = -\left[y \log \sigma(w \cdot x + b) + (1-y) \log(1 - \sigma(w \cdot x + b))\right]$$

Cross-entropy loss

- Going back to the sentiment analysis example. We want loss to be:
  - smaller if the model estimate is close to correct
  - bigger if model is confused
- Let's first suppose the true label of this is y=1 (positive)

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) $\in$ doc)	3
x_2	count(negative lexicon) $\in$ doc)	2
x_3	$\begin{cases} 1 & \text{if } \text{“no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns $\in$ doc)	3
x_5	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

- What’s the cross-entropy loss for when w = [2.5, -5.0, -1.2, 0.5, 2.0, 0.7] and b = 0.1?

Cross-entropy loss

- True value is y=1. How well is the model doing?

$$
\begin{aligned}
p(+|x) = P(Y=1|x) &= \sigma(w\cdot x+b) \\
&= \sigma([2.5,-5.0,-1.2,0.5,2.0,0.7]\cdot[3,2,1,3,0,4.19]+0.1) \\
&= \sigma(.833) \\
&= 0.70 \qquad (5.6)
\end{aligned}
$$

- Pretty well. What's the loss?

$$
\begin{aligned}
L_{\text{CE}}(\hat{y},y) &= -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\
&= -[\log\sigma(w\cdot x+b)] \\
&= -\log(.70) \\
&= .36
\end{aligned}
$$

Cross-entropy loss

- Suppose, for the same model, the true value is y=0 instead (i.e. it's a negative review). How well is the model doing?

$$\begin{aligned} p(-|x) = P(Y=0|x) &= 1-\sigma(w\cdot x+b) \\ &= 0.30 \end{aligned}$$

- Not so well this time. What's the loss?

$$\begin{aligned} L_{\text{CE}}(\hat{y},y) &= -[y\log\sigma(w\cdot x+b)+(1-y)\log(1-\sigma(w\cdot x+b))] \\ &= -[\log(1-\sigma(w\cdot x+b))] \\ &= -\log(.30) \\ &= 1.2 \end{aligned}$$

- Loss is higher. More of a change is needed to the weights to predict this example correctly.

Cross-entropy loss

- For multinomial output, a **softmax cross-entropy loss** can be used
- The true output/label is a **one-hot vector**, where for a ground-truth label *i,* $y_i = 1$ and it's 0 everywhere else:

$$
\begin{aligned}
L_{CE}(\hat{y}, y) &= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \hat{y}_i \\
&= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \hat{p}(y = k|x) \\
&= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \frac{\exp(\mathbf{z}_k)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)}
\end{aligned}
$$

- The terms in the sum will be 0, except for the term corresponding to the true class, so, this can be simplified to the below, also called the **negative log loss**.

$$
L_{CE}(\hat{y}, y) = -\log \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{K} \exp(\mathbf{z}_j)}
$$

Gradient descent

- We want to change/learn the weights to **minimise loss** over the training data.

- To do this, we want to find a **loss curve** and change weights such that the predictions get to the bottom of it.

- For logistic regression, loss function is **convex**
  - A convex function has just one minimum
  - Gradient descent starting from any point is guaranteed to find the minimum
  - (Loss for deep neural networks is non-convex)

Gradient descent

- Let's make explicit that the loss function is parameterized by weights θ=(w,b)
- And we'll represent $\hat{y}$ as f(x; θ) to make the dependence on θ more obvious
- We want the weights that minimize the loss, averaged over all *m* examples in the corpus (or sub-corpus/batch):

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\mathrm{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

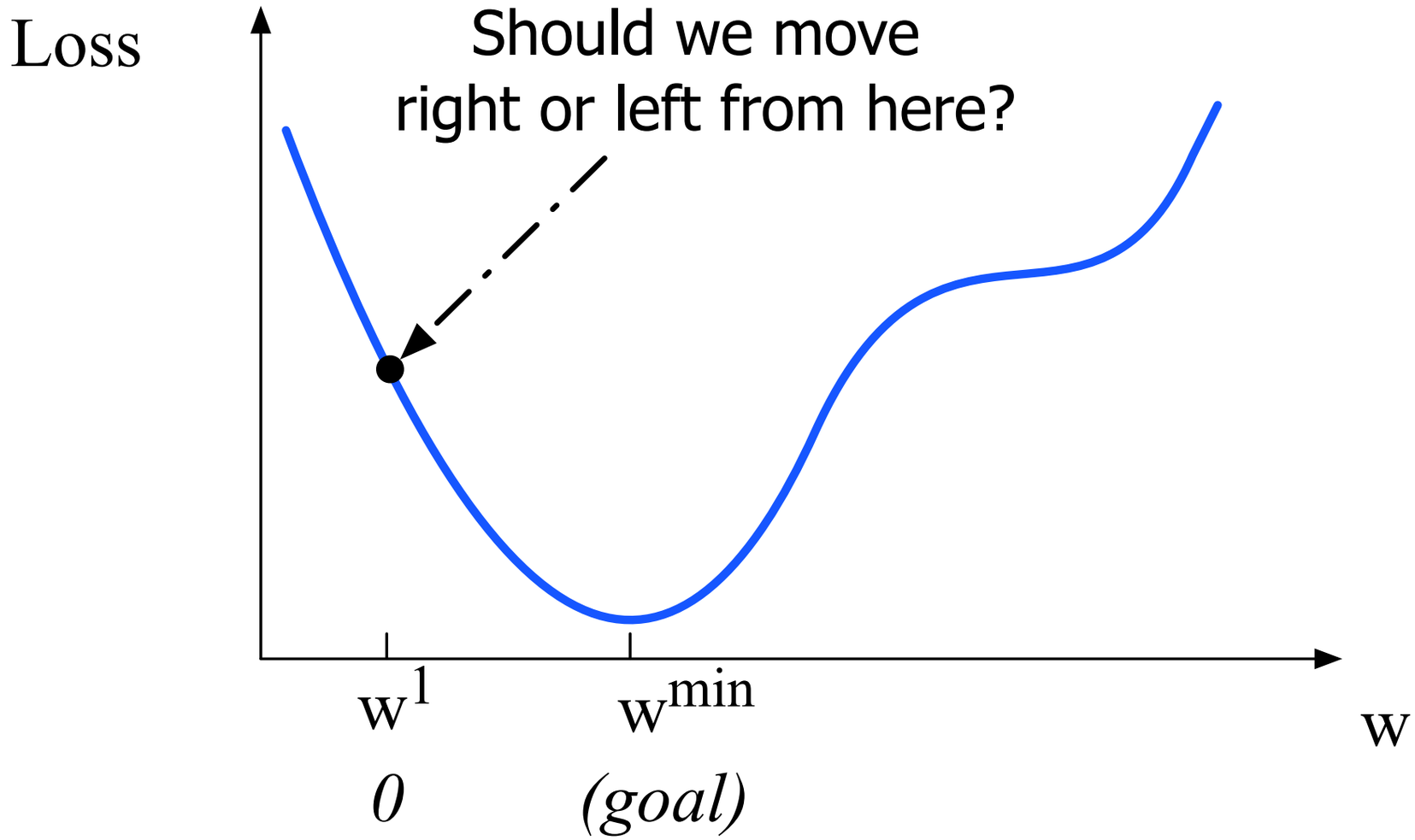
- To do this, we want to find a loss curve and change weights such that the predictions get to the bottom of it.

Gradient descent

We use a method called **gradient descent.**

Q: Given current *w*, should we make it bigger or smaller?

A: Move *w* in the reverse direction from the slope of the function



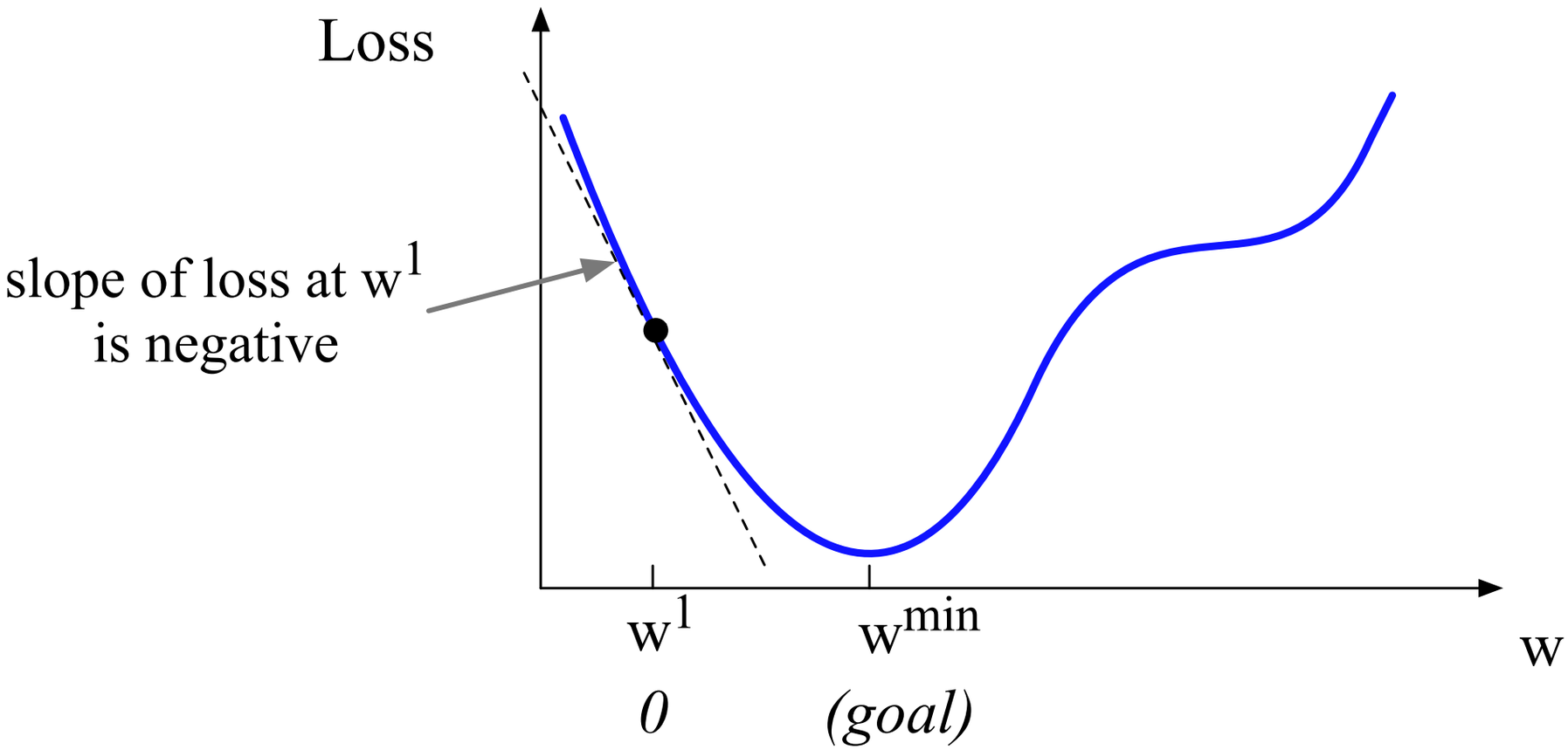
GD finds the gradient of the loss function at the current point and moves in the opposite direction (down the slope).

Gradient descent

We use a method called **gradient descent.**

Q: Given current *w*, should we make it bigger or smaller?

A: Move *w* in the reverse direction from the slope of the function



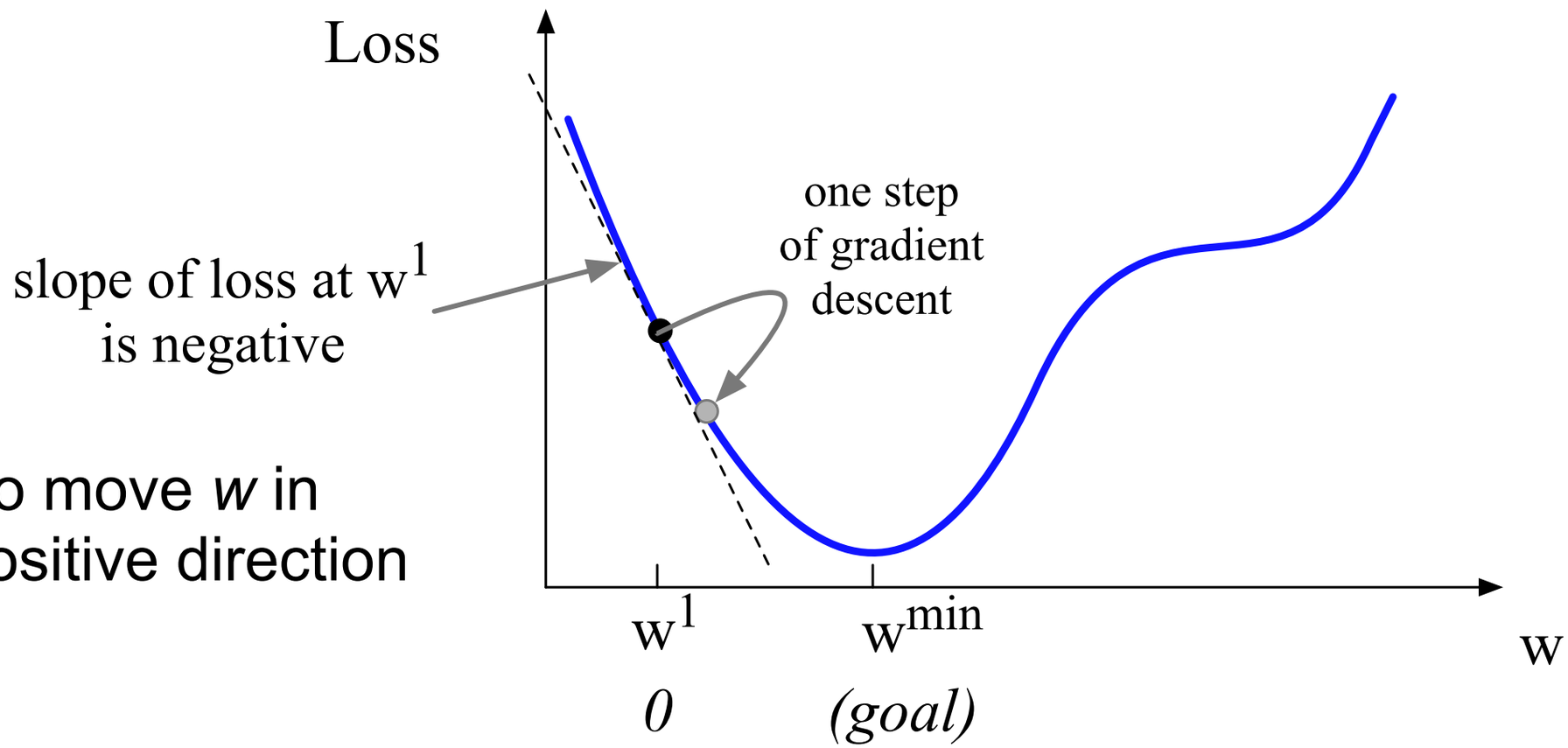
GD finds the gradient of the loss function at the current point and moves in the opposite direction (down the slope).

Gradient descent

We use a method called **gradient descent.**

Q: Given current *w*, should we make it bigger or smaller?

A: Move *w* in the reverse direction from the slope of the function



GD finds the gradient of the loss function at the current point and moves in the opposite direction (down the slope).

Gradient descent

- How do we compute how much to move a given scalar weight *w*?

- The value of the gradient (slope in our example) at a given point in time t is

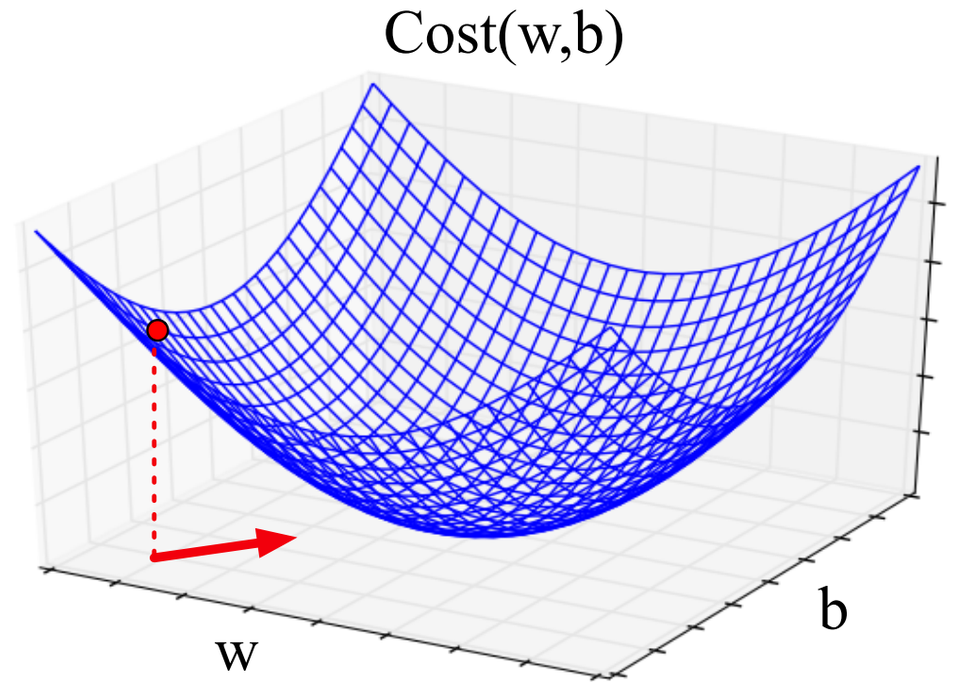
$$\frac{d}{dW}L(f(x;w),y)$$

- This is weighted by a **learning rate η**, to give the updated value for *w* at *t+1*.

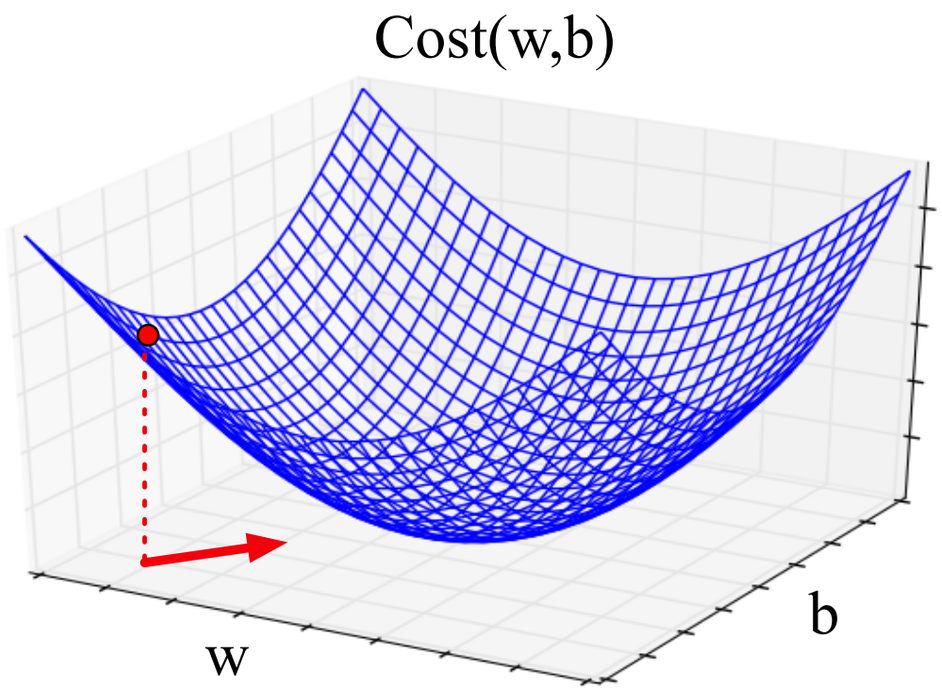
$$w^{t+1} = w^t - \eta \frac{d}{dw}L(f(x;w),y)$$

Gradient descent

- However, we want to go beyond the loss of f(x; w) for one scalar weight w, to a vector of weights θ.
- We want to know where in the N-dimensional space (of the N parameters that make up θ) we should move.
- The gradient is just such a vector; it expresses the directional components of the sharpest slope along each of the N dimensions. e.g. 2 dimensions w and b:



Gradient descent



- Real gradients are much longer; lots and lots of weights

For each dimension/variable w_i in w (plus the bias b), the gradient component i tells us the slope with respect to that variable.

- Key question: "How much would a small change in w_i influence the total loss function L?"
  - We express **the slope as a partial derivative of the loss with respect to** w_i
  - The gradient is then defined as a vector of these partials.

Gradient descent

- Rather than one partial derivative for one gradient of one line, we have a vector of partials to characterise the **gradient of the loss function**.
- We'll represent $\hat{y}$ as f (x; θ) to make the dependence on θ more clear:

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix}$$

- The final equation for updating θ based on the gradient is thus:

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

- Where learning rate η is a hyperparameter:
- too high: the learner will take big steps and overshoot
- too low: the learner will take too long

Gradient descent

- Given the loss function L(f (x; θ), y) for logistic regression is cross-entropy between the prediction and the true value:

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log (1 - \sigma(w \cdot x + b))]$$

- The derivatives (gradients) for each weight w_j in the weight vector to be computed for the **gradient** vector, turn out to be the below (check 5.8 in Jurafsky and Martin for elegant derivation):

$$\frac{\partial L_{\text{CE}}(\hat{y}, y)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

- The gradient with respect to a single weight w_j represents an intuitive value: the difference between the true y and our estimated $\hat{y}$ = σ (w · x + b) for that observation, multiplied by the corresponding input value x_j.

- Softmax output gradient w.r.t. weight w_k for ground-truth *i* uses a 1-hot vector from ground truth:

$$\begin{aligned}\frac{\partial L_{CE}}{\partial w_k} &= -(\mathbb{1}\{y = k\} - p(y = k|x))\mathbf{x}_k \\ &= -\left(\mathbb{1}\{y = k\} - \frac{\exp(\mathbf{w}_k \cdot \mathbf{x} + \mathbf{b}_k)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + \mathbf{b}_j)}\right)\mathbf{x}_k\end{aligned}$$

Gradient descent

- Now we can calculate the derivative of the loss function for each parameter/weight, we have the gradient which we can move iteratively from its original value to minimise the loss by the following:

Step 1: Initialize the weights w with random values

- Then, repeat the following until the step size becomes very small, or after a fixed number of iterations:

Step 2: Calculate the weight values w in terms of the derivative of the loss function for each separate w_j.

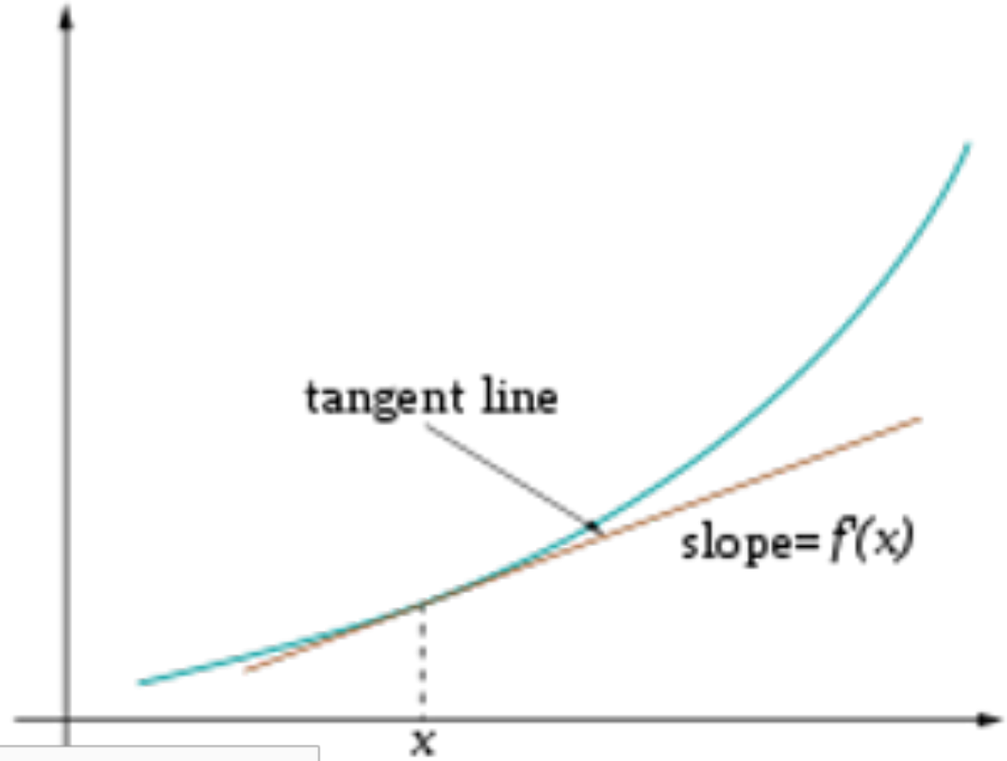
Step 3: Calculate the step sizes: step size = slope x learning weight

Step 4: Calculate the new parameters, as:
New parameter = old parameter - step size

Descending by using derivatives

We will minimize a cost function by gradient descent

Trivial example: (from Wikipedia)
Find a local minimum of the function
$f(x) = x^4 - 3x^3 + 2$,
with derivative $f'(x) = 4x^3 - 9x^2$




```
x_old = 0
x_new = 6 # The algorithm starts at x=6
eps = 0.01 # step size
precision = 0.00001

def f_derivative(x):
    return 4 * x**3 - 9 * x**2

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new = x_old - eps * f_derivative(x_old)

print("Local minimum occurs at", x_new)
```

Subtracting a fraction of the gradient moves you towards the minimum!

Stochastic Gradient descent

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}$, $x^{(2)}$,..., $x^{(m)}$
y is the set of training outputs (labels) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow 0$
repeat til done # see caption
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
    1. Optional (for reporting): # How are we doing on this tuple?
      Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
      Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$) from the true output $y^{(i)}$?
    2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
    3. $\theta \leftarrow \theta - \eta\ g$ # Go the other way instead
return θ

The algorithm terminates when it **converges** (when the gradient norm < ε), or when progress halts (for example when the loss starts going up on a held-out set).

Overfitting and regularisation

- A model that perfectly matches the training data has a problem.
- It will overfit to the data, modelling **noise**:
  - A random word that perfectly predicts y (it happens to only occur in one class) will get a very high weight.
  - Failing to generalise to a test set without this word.
- A good model should be able to generalise.

Overfitting and regularisation

+

This movie drew me in, and it'll do the same to you.

'harmless' features:

x1= "this"
x2= "movie"
x3= "hated"
x4="drew me in"

−

I can't tell you how much I hated this movie. It sucked.

4-gram features just 'memorise' the training set and might cause problems in testing:

x5= "the same to you"
x7= "tell me how much"

Overfitting and regularisation

- 4-gram model on tiny data will just memorise the data
  - 100% accuracy on the training set
  - But it will be surprised by the novel 4-grams in the test data
  - Low accuracy on test set
- Models that are too powerful can overfit the data
- Fitting the details of the training data so exactly that the model doesn't generalize well to the test set
- How to avoid overfitting?
  - **Regularisation** in logistic regression
  - **Dropout** of neurons in neural networks

Overfitting and regularisation

- Add a **regularisation term** R(θ) to the **loss function**
- Idea: choose an R(θ) that penalizes large weights.
- Fitting the data well with lots of big weights not as good as fitting the data a little less well, with small weights

$$\hat{\theta} = \underset{\theta}{argmin} \left[\frac{1}{m} \sum_{i=1}^{m} L_{CE}(f(x^{(i)}; \theta), y^{(i)}) \right] + \alpha R(\theta)$$

Overfitting and regularisation

L1 regularisation (**lasso regression**) (Tibshirani, 1996)

- The **sum of the (absolute value of the) weights**
- Named after the L1 norm $||W||_1$, = sum of the absolute values of the weights, = Manhattan distance:

$$R(\theta) = \|\theta\|_1 = \sum_{i=1}^{n} |\theta_i|$$

(note the single vertical bars here mean absolute value, not set cardinality)

- L1 regularised loss function:

$$\hat{\theta} = \underset{\theta}{argmin} \left[\frac{1}{m} \sum_{i=1}^{m} L_{CE}(f(x^{(i)}; \theta), y^{(i)}) \right] + \alpha \sum_{i=1}^{n} |\theta_i|$$

Overfitting and regularisation

L2 regularisation (ridge regression):

- The **sum of the squares of the weights**
- The name is because this is the (square of the) L2 norm $||\theta||_2$, = Euclidean distance of θ to the origin.

$$R(\theta) = \|\theta\|_2^2 = \sum_{i=1}^{n} \theta_i^2$$

- L2 regularised loss function:

$$\hat{\theta} = \underset{\theta}{argmin} \left[\frac{1}{m} \sum_{i=1}^{m} L_{CE}(f(x^{(i)}; \theta), y^{(i)}) \right] + \alpha \sum_{i=1}^{n} \theta_i^2$$

- L2 prefers weight vectors with many small weights, L1 prefers sparse solutions with some larger weights but many more weights set to zero. Thus L1 regularization leads to much sparser weight vectors, that is, far fewer features than L2.

Logistic Regression to Neural Nets

- That's all well and good, but why is this useful? We can just recast logistic regression as a neural net (a **single-layered perceptron**), but its performance and limitations don't change.

- On its own, a single layered perceptron can't model the interaction **between features** in the input - in fact, it can't even model exclusive OR (**XOR**), but a **multi-layered network of multiple logistic units** can (Minsky and Papert, 1967).

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

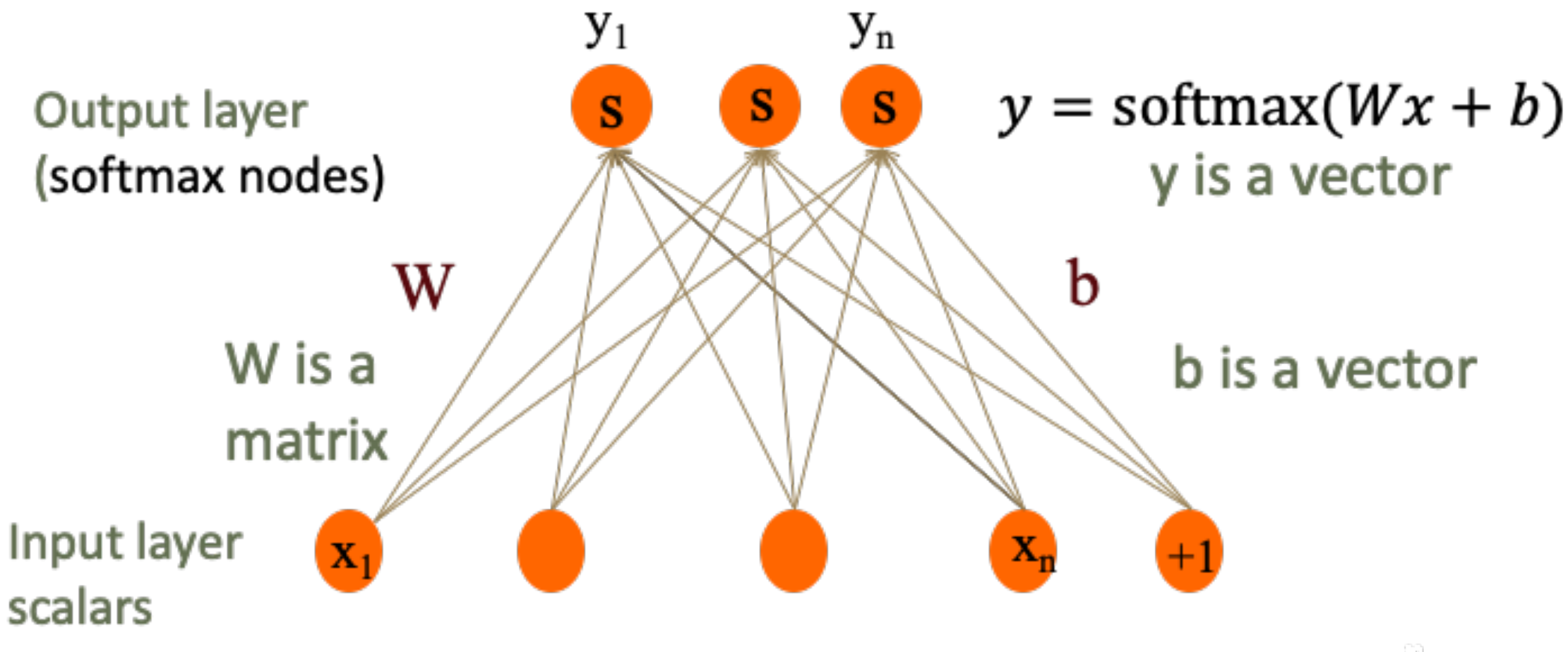
OUTLINE

1) The renaissance of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

Logistic Regression to Neural Nets

Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



What's the difference between logistic regression and a deep net?

- Similarities:
  - **Input layer** and **output layer** of **neural units**, with layers represented by vectors. Models are functions from input vector to output scalar or vector via multiplication by weight vector(s). LR or **perceptron** is a one-layered neural net.
  - **Non-linear activation functions** of (non-input) neural units.
  - **Softmax** used to squash output layer into a probabilistic function for multinomial LR/perceptrons).
  - A **loss function** such as cross-entropy calculates the difference between predictions in output layer and ground-truth outputs.
  - Weights learned/optimised by **stochastic gradient descent** based on the loss function (though note global minimum guaranteed to be found in LR, not in NNs).
  - **Regularisation** on loss function (e.g. L1, L2) is used to stop overfitting.

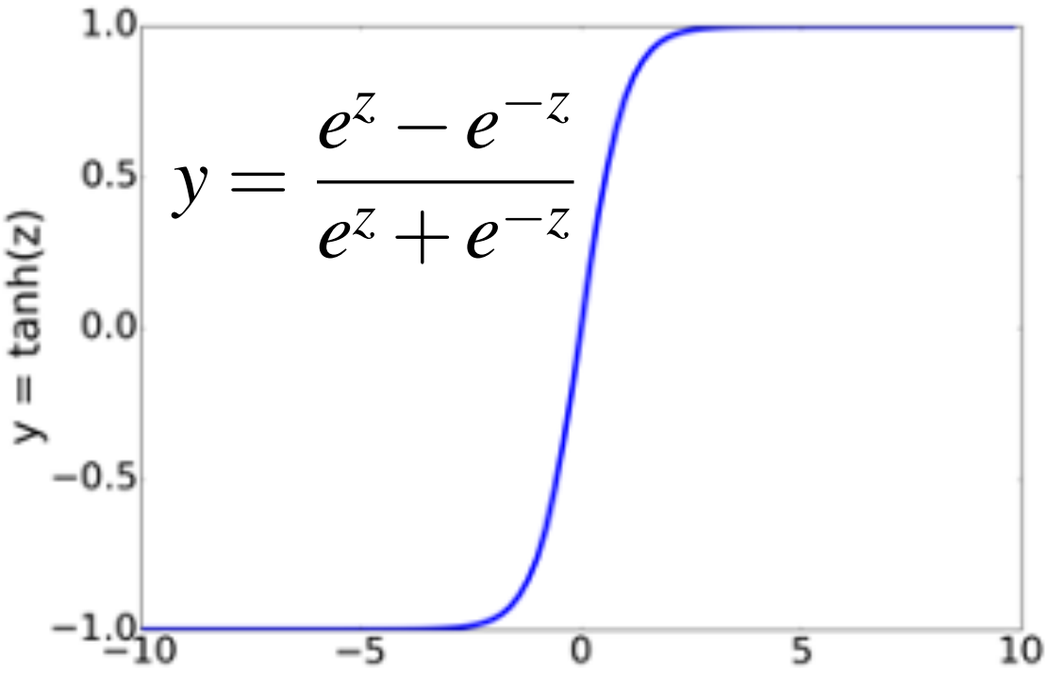
What's the difference between logistic regression and a deep net?

- Differences:
  1. Different types of **non-linear activation function** can be used for non-input layers other than the logistic (**sigmoid**) function: **tanh**, **ReLU**
  2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing **non-linear functions** to be learned (e.g. XOR).
  3. During learning, calculation of gradients for each weight more complicated due to having to attribute blame further back than previous layer: **backpropagation**
  4. In NLP applications, projection layers for each word input consist of **embeddings** (vectors representing the word).

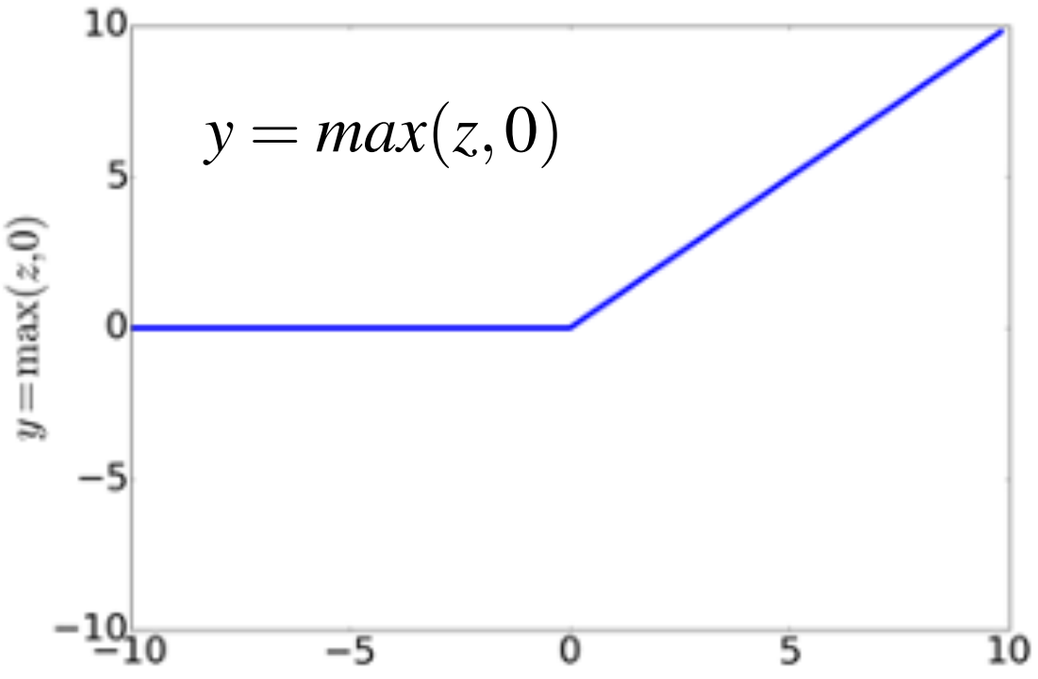
Non-linear activation functions

1. Different types of **non-linear activation function** can be used for non-input layers other than the logistic (**sigmoid**) function.

- **tanh**



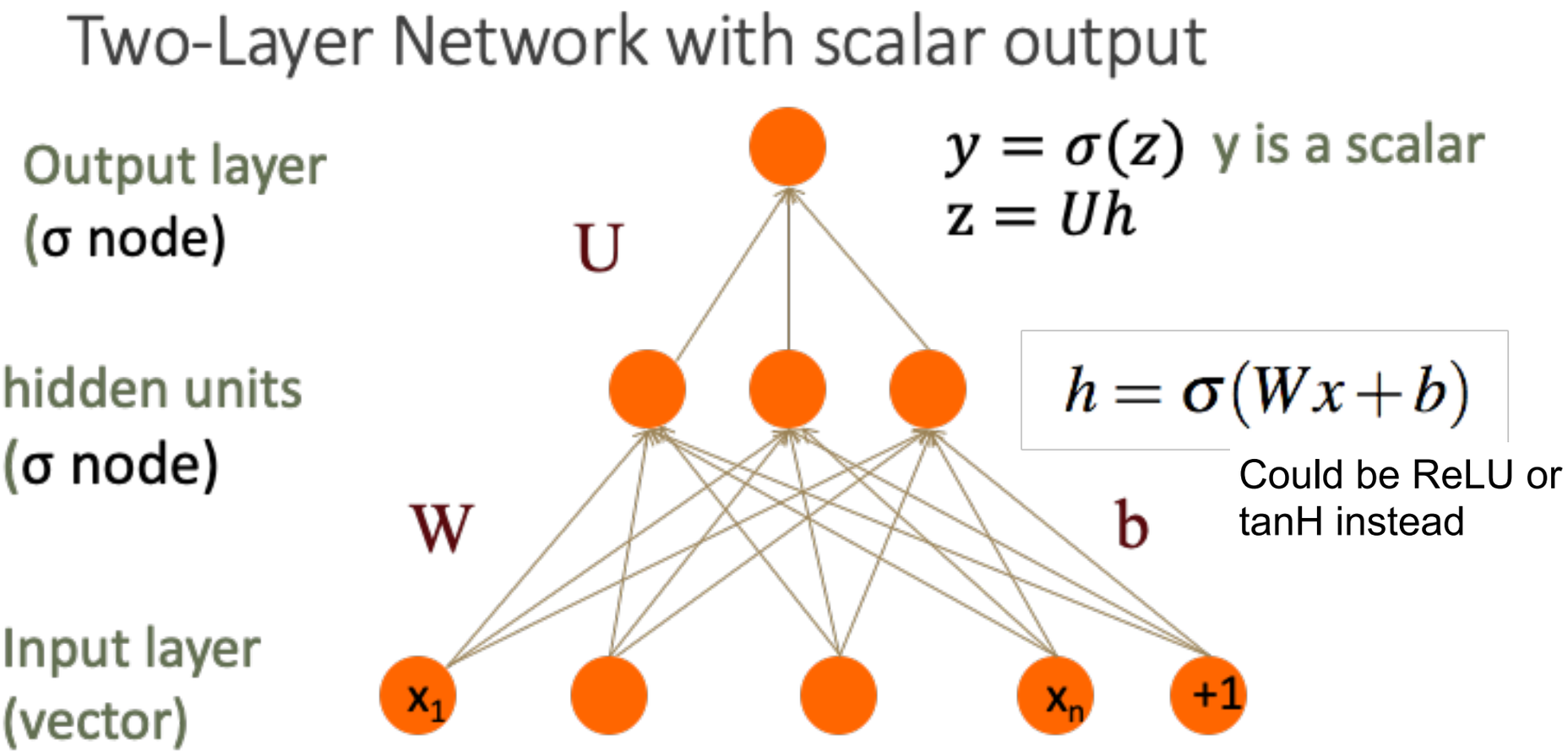
- **ReLU** (Rectified Linear Unit)
- (most common)



- tanh function has the nice properties of being smoothly differentiable and mapping outlier values **toward the mean**.
- ReLU is linear when z>0, and overcomes the **vanishing gradient problem** of sigmoid and tanh, since the derivative of ReLU for high values of z is 1 rather than very close to 0.

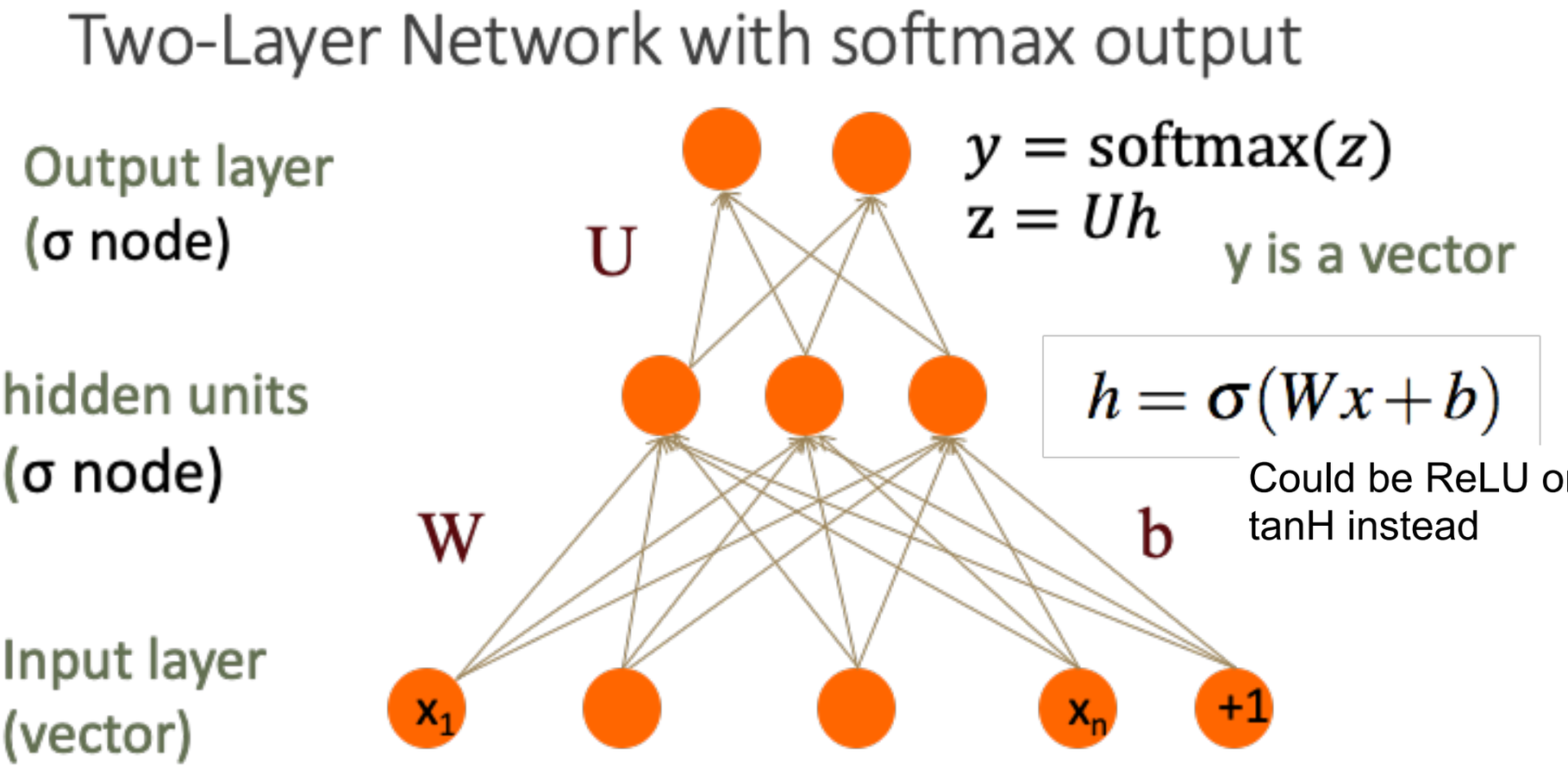
Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.



Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.



- For a two-layered network, you don't just have one set of weights W, but two, *W* and *U* where *W* maps from the input layer to a hidden layer, and *U* maps from the hidden layer to the output layer, over which sigmoid (for binary task) or softmax (for multiple outputs) is applied.

Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- A perceptron (single layered neural net i.e. a logistic/non-linear regression classifier) **can learn linear functions**, as while there's a logistic function applied, it is is still a product of a simple summation of input features.
- Perceptron equation given x_1 and x_2 with weights w_1 and w_2 is the equation of a line

$$w_1 x_1 + w_2 x_2 + b = 0$$

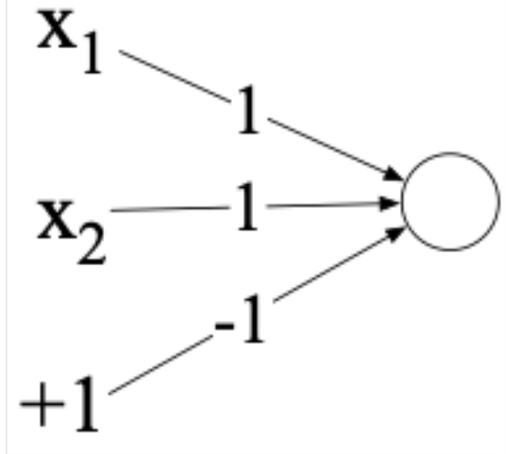
- This line acts as a decision boundary:
  - 0 if input is on one side of the line
  - 1 if on the other side of the line

Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

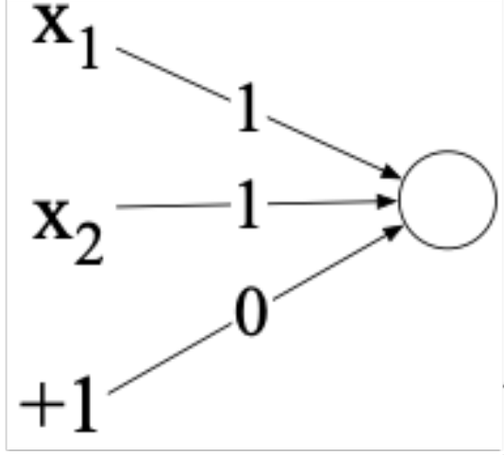
- A perceptron can compute and learn AND and OR functions with the appropriate weights:

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND

AND		
x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



OR

OR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

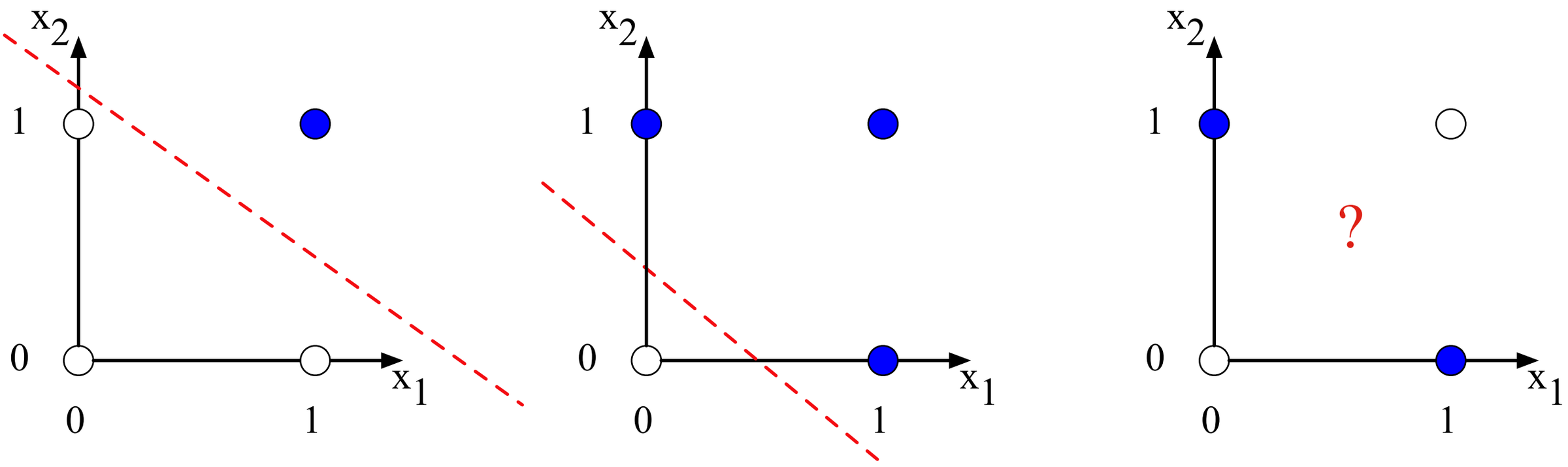
??

XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- Perceptron can find a separating line to compute AND and OR functions:



a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

- But **not XOR (exclusive OR)** - no linear separation possible with just one weight per input.

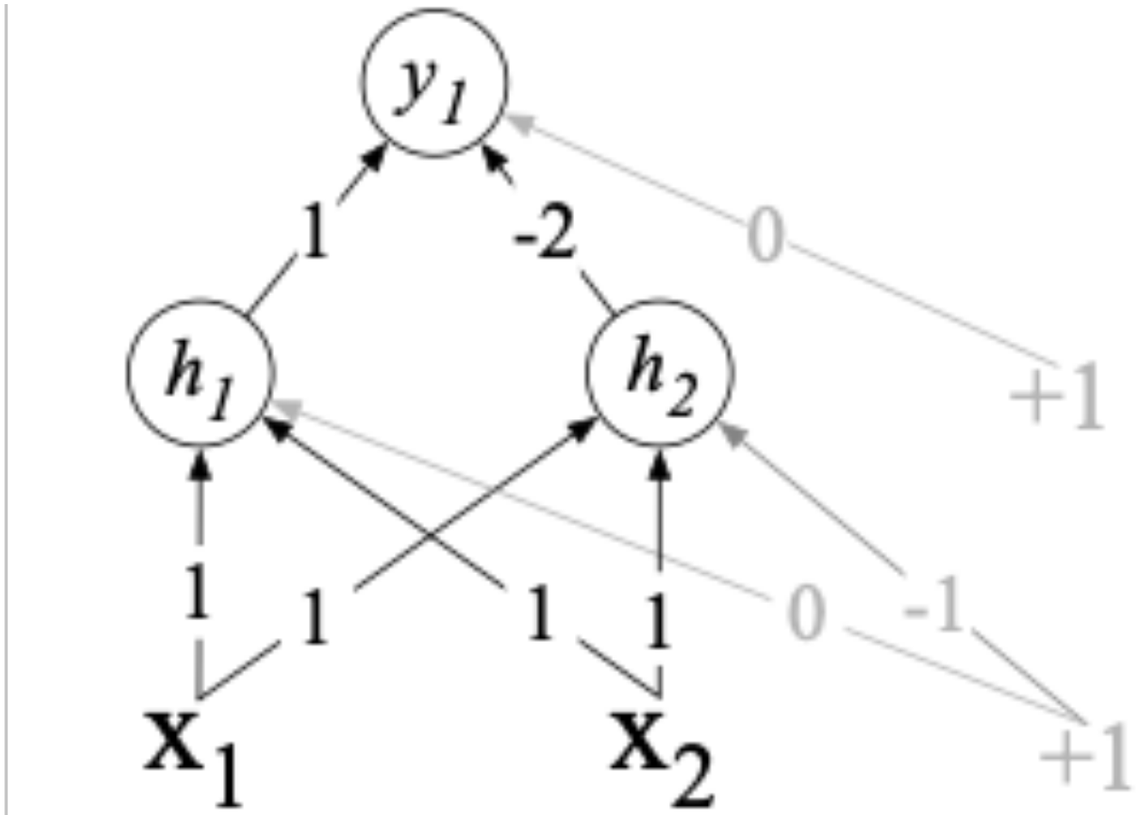
Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- XOR can be computed by a 2-layered net with one hidden layer:

XOR

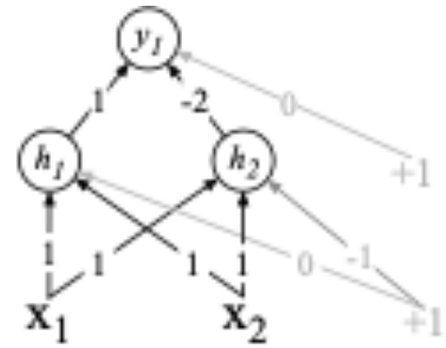
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

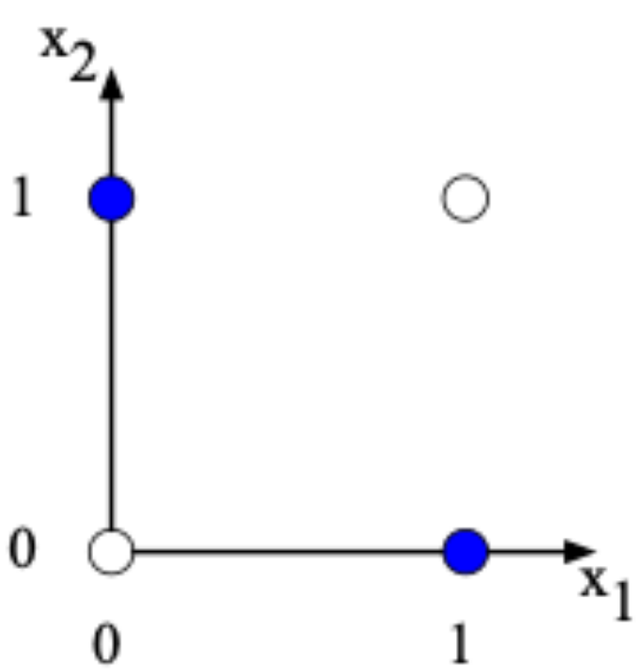


Going deeper

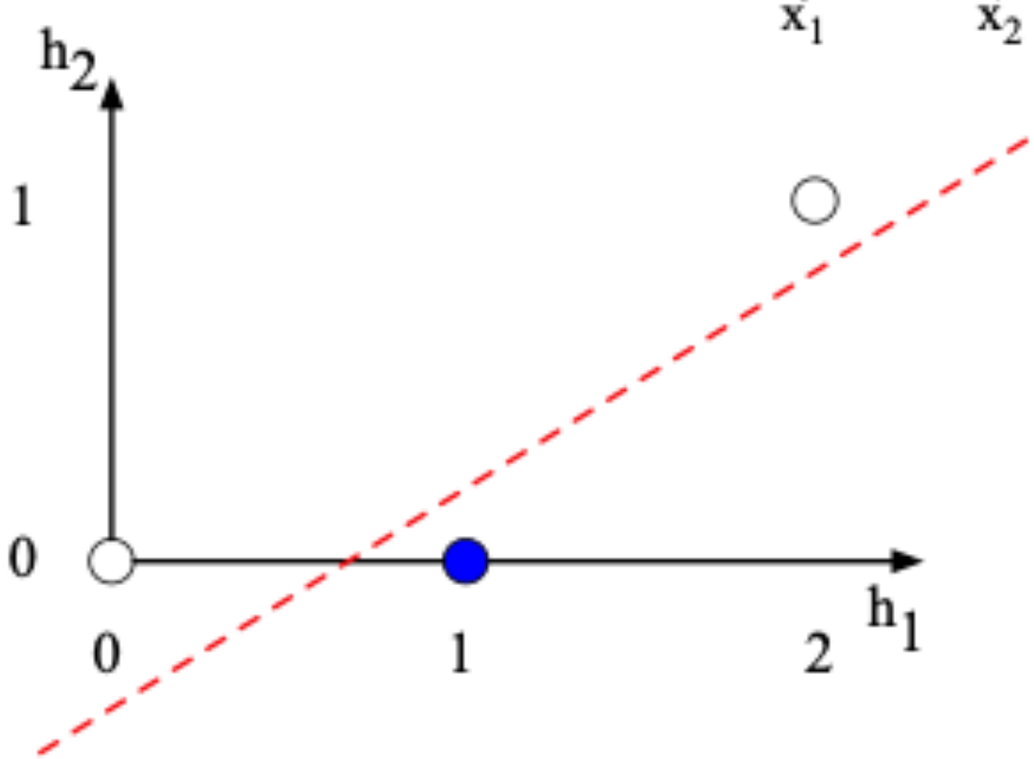
2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- The hidden layer representation makes XOR possible/a linearly separable problem. Note it has squashed the two blue inputs into one point in its hidden layer:





a) The original *x* space



b) The new (linearly separable) *h* space

Going deeper

2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- For the purposes of NLP applications, we don't have to manually define **feature interactions** any more: a deep net, if of the right architecture and trained properly, will be able to model those interactions for a given task (e.g. classification) and therefore get better performance.

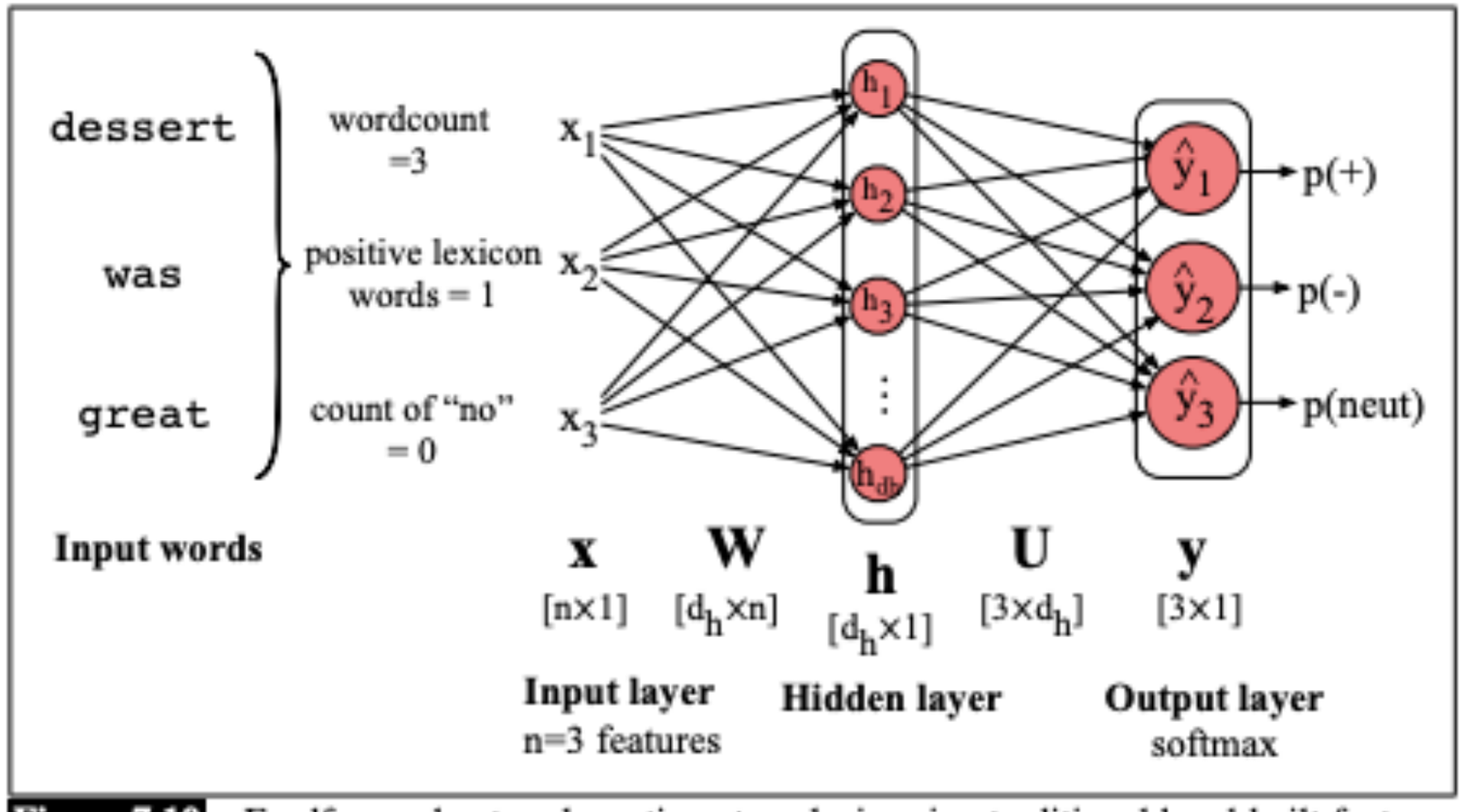


Figure 7.10 Feedforward network sentiment analysis using traditional hand-built features of the input text.

Going deeper

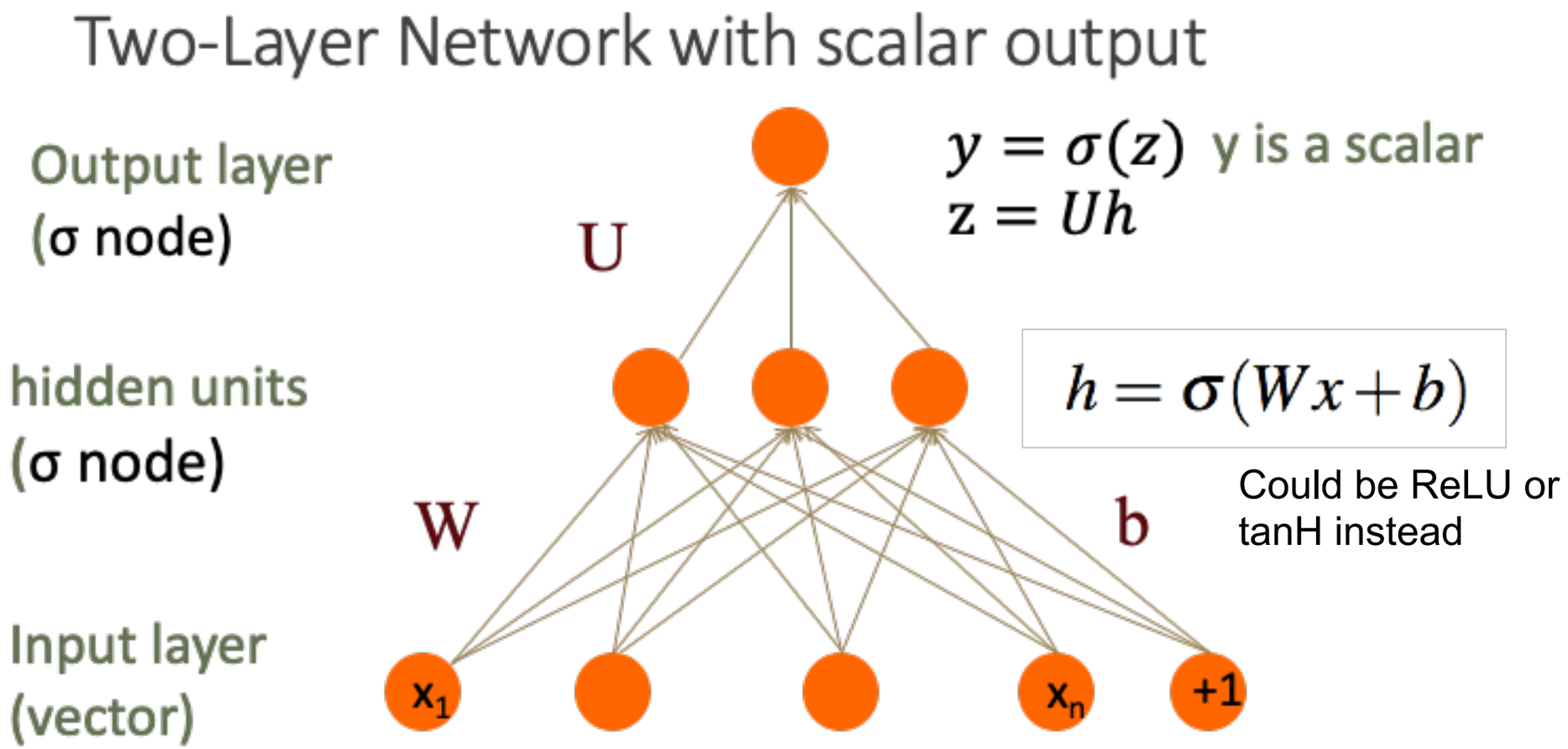
2. **Multiple layers** of neurons are used (i.e. a deep net with '**hidden**' layers), allowing non-linear functions to be learned.

- This means we have many more parameters to train than in the single layered case, which all interact according to the network's architecture- how do we do that?

Backpropagation for training

3. During learning, calculation of gradients for each weight more complicated due to having to attribute blame further back than previous layer: **backpropagation.**

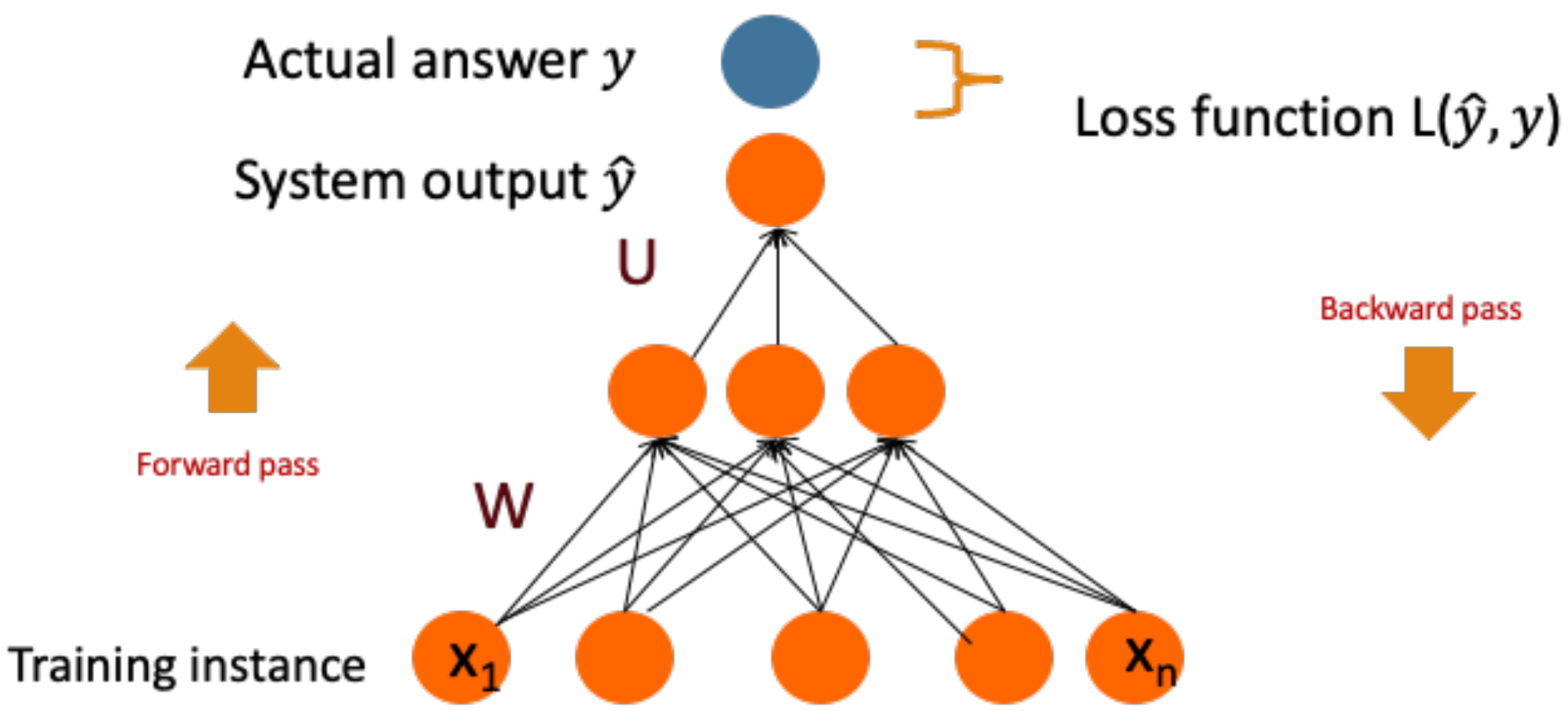
- E.g. a two-layered feedforward neural net with scalar output, sigmoid over input -> hidden and sigmoid over hidden -> output:



Backpropagation for training

3. During learning, calculation of gradients for each weight more complicated due to having to attribute blame further back than previous layer: **backpropagation.**

- E.g. a two-layered feedforward neural net with scalar output, sigmoid over input -> hidden and sigmoid over hidden -> output:



Backpropagation for training

3. During learning, calculation of gradients for each weight more complicated due to having to attribute blame further back than previous layer: **backpropagation.**

For every training tuple (x,y):

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight U_k from hidden layer to the output layer
      - Update the weight
  - For every hidden node
    - Assess how much blame it deserves for the current answer
    - For every weight W_j from input layer to the hidden layer
      - Update the weight

Backpropagation for training

For every training tuple (x,y):

- Run forward computation to find our estimate $\hat{y}$
  - $h = \sigma(W \cdot x + b)$
  - $z = U \cdot h$
  - $\hat{y} = \sigma(z)$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$

      $L_{CE}(\hat{y}, y) = -\left[y \log z + (1-y)\log(1-z)\right]$
    - For every weight U_k from hidden layer to the output layer
      - Update the weight according to gradient of how change in U_k affects $L_{CE}(\hat{y}, y)$ with node h_k 's current activation:

        $$\frac{\partial L_{CE}(\hat{y}, y)}{\partial U_k} = \left[\sigma(U \cdot h) - y\right] h_k$$
      - $U_t := U_{t-1} - \eta \nabla L_{CE}(\hat{y}, y)$
  - For every hidden node
    - ….

Backpropagation for training

For every training tuple (x,y):

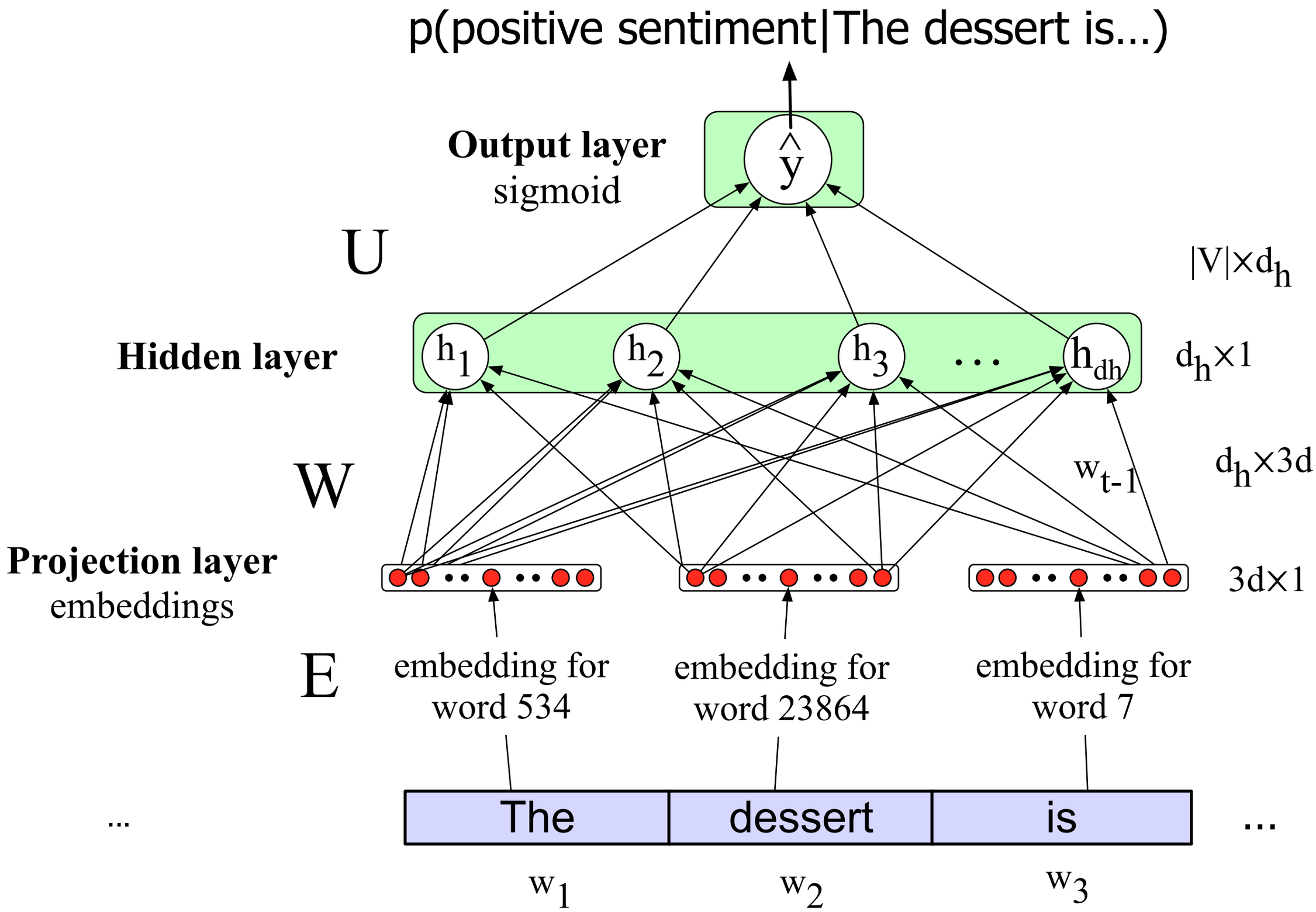
- Run backward computation to update weights:
  - …
  - For every hidden node
    - Assess how much blame it deserves for the current answer
      - For this we need the **chain rule from calculus** to get the relevant **partial derivatives** according to the network (graph of the whole NN as a function). i.e. how much does a change in W_j affect $L_{CE}(\hat{y}, y)$?

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial W_j} = \frac{\partial L_{CE}(\hat{y}, y)}{\partial z} \times \frac{\partial z}{\partial h} \times \frac{\partial h}{\partial W_j}$$

      - (Note we are not going through the details of calculating the partials to get example values for $\nabla L_{CE}(\hat{y}, y)$- it is quite involved, and not necessary here)
- For every weight W_j from input layer to the hidden layer
  - Update the weight
  - Get step size according to the gradient and learning rate:

$$W_t := W_{t-1} - \eta \nabla L_{CE}(\hat{y}, y)$$

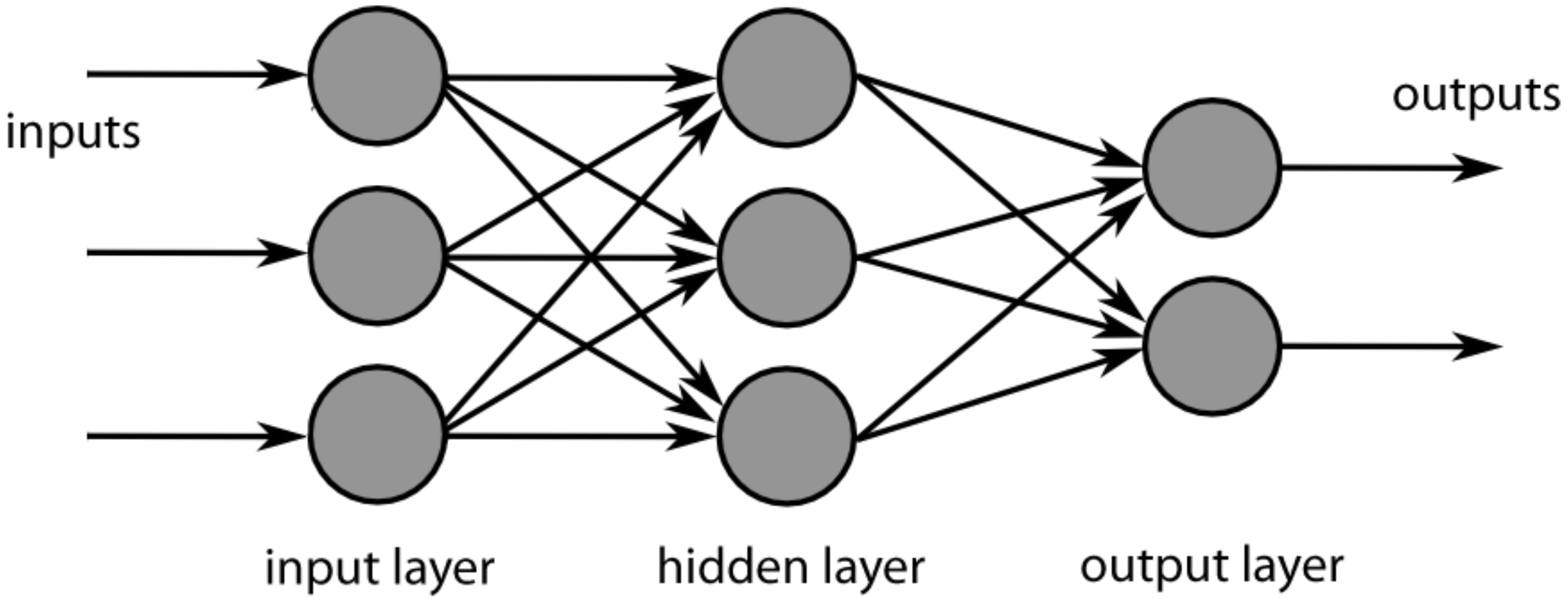
4. In NLP applications, projection layers for each word input consist of **embeddings** (vectors representing the word).



OUTLINE

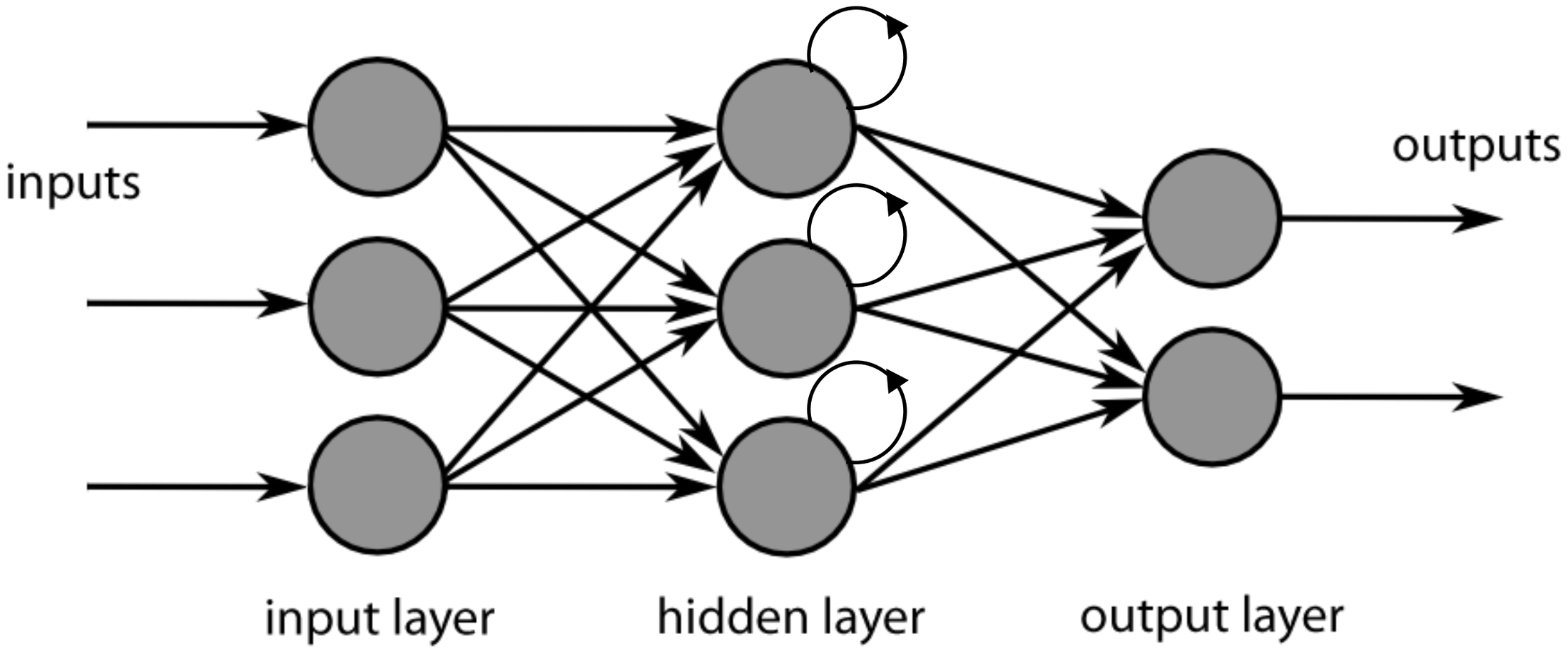
1) The arrival of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

Feedforward Neural Nets



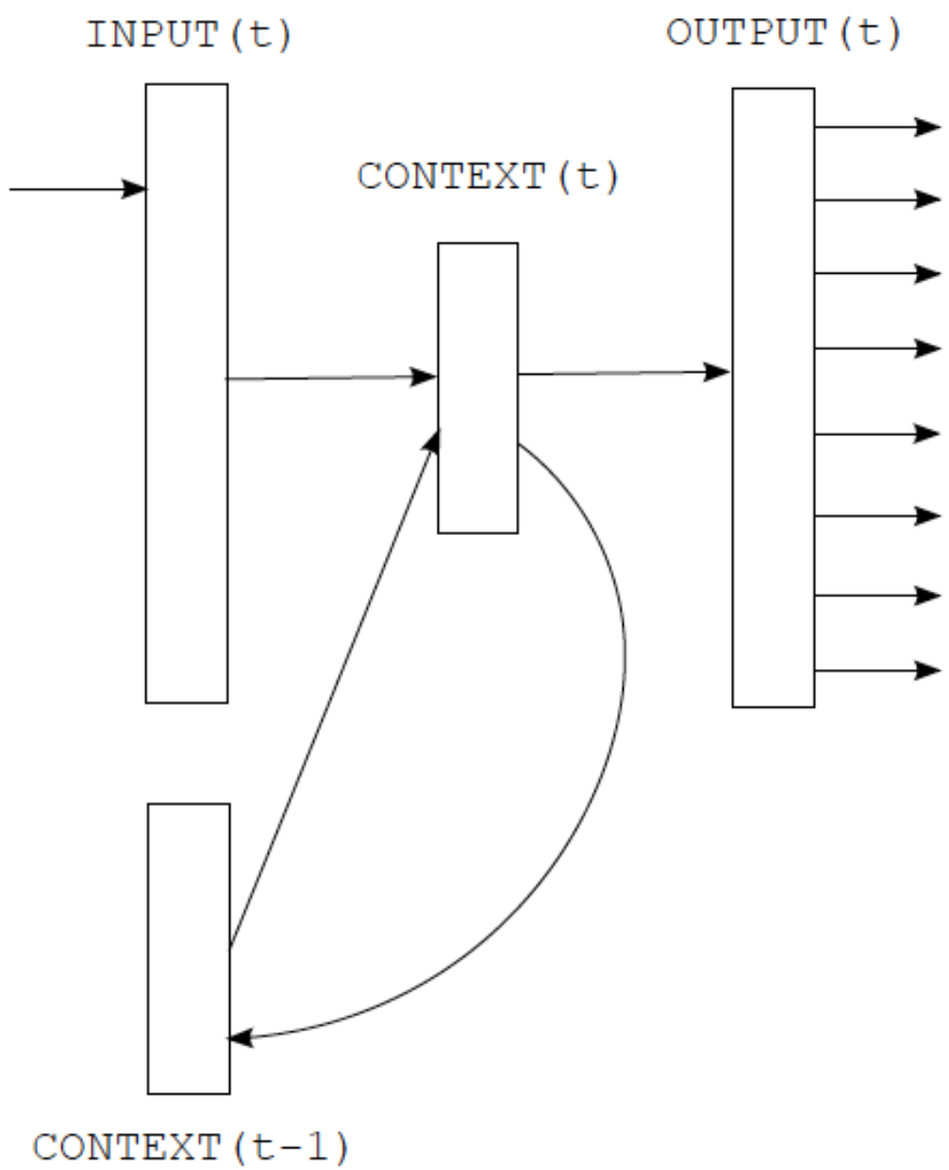
- Feed-forward only
- Context/input limited by the network structure, fixed in advance (e.g. only three input words for each instance).

Recurrent Neural Nets



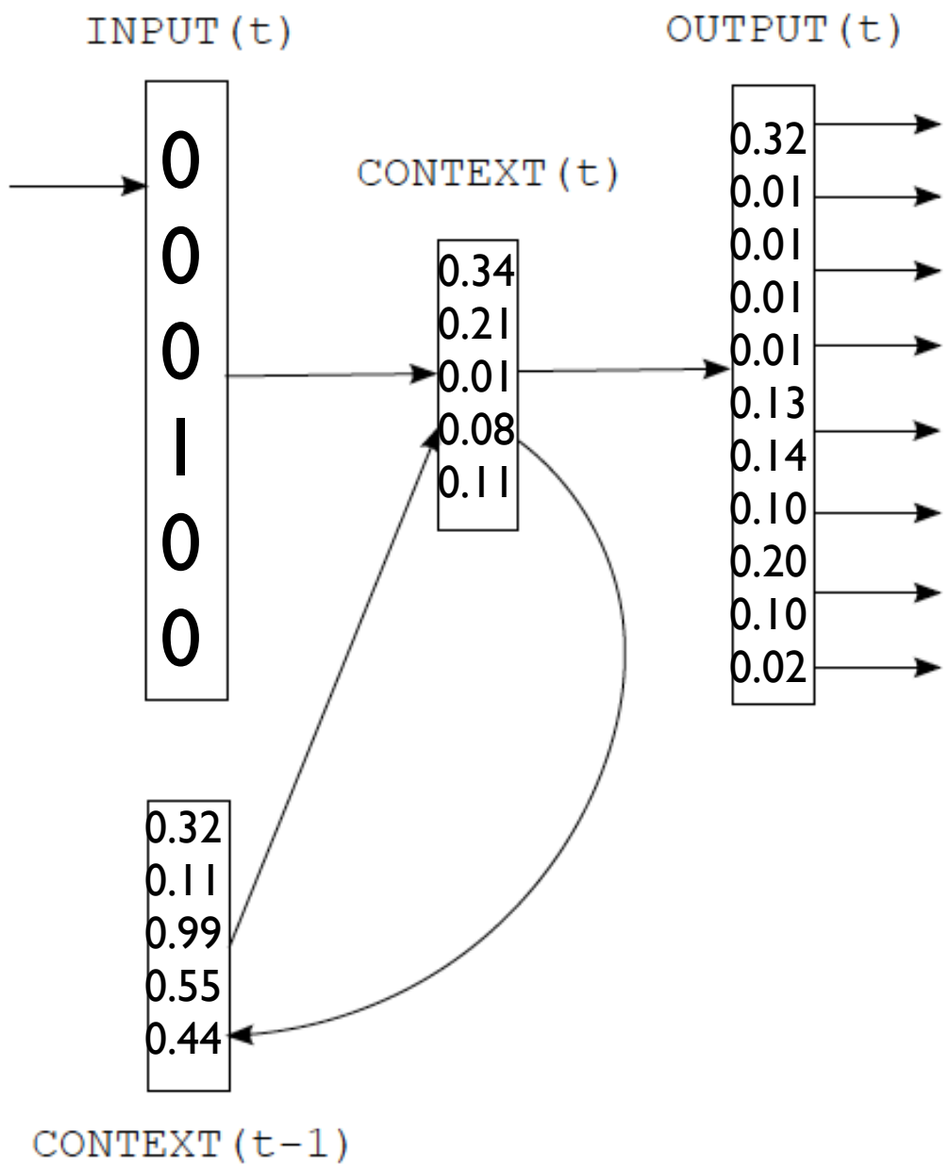
- Use a ‘working memory’ by making a copy of the hidden layer for the next time step.
- Unlimited context available!
- Should make better decisions with more ‘working memory’
- They are even ‘deeper’ if you unroll them out over time

Recurrent Neural Nets



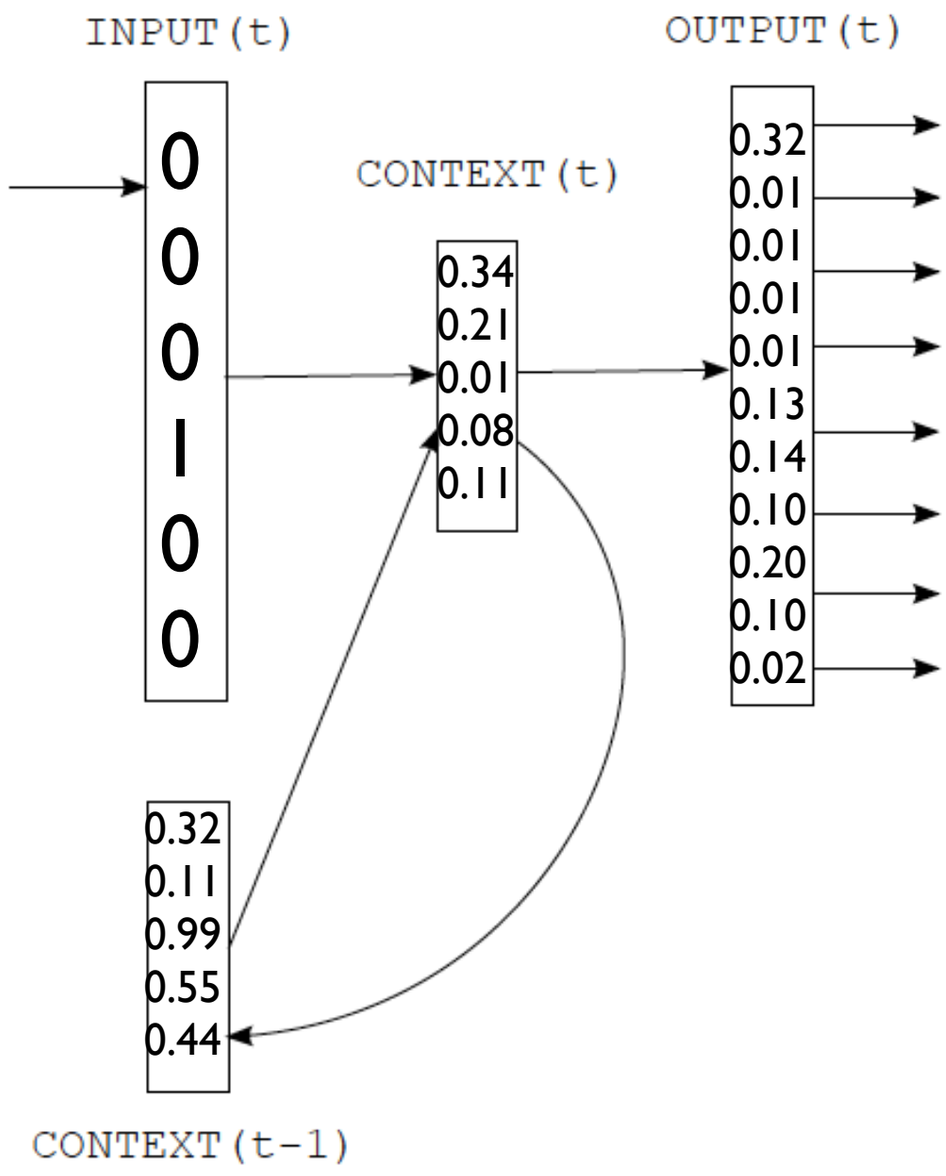
- A copy of time *t-1*'s context/hidden nodes stored for use at time *t*
- How do we do this clever (recurrent) computation?

Recurrent Neural Nets



- A copy of time *t-1*'s context/hidden nodes stored for use at time *t*
- How do we do this clever (recurrent) computation?
- In reality, these can be done as vector/matrix operations

Recurrent Neural Nets



(1)
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H$$

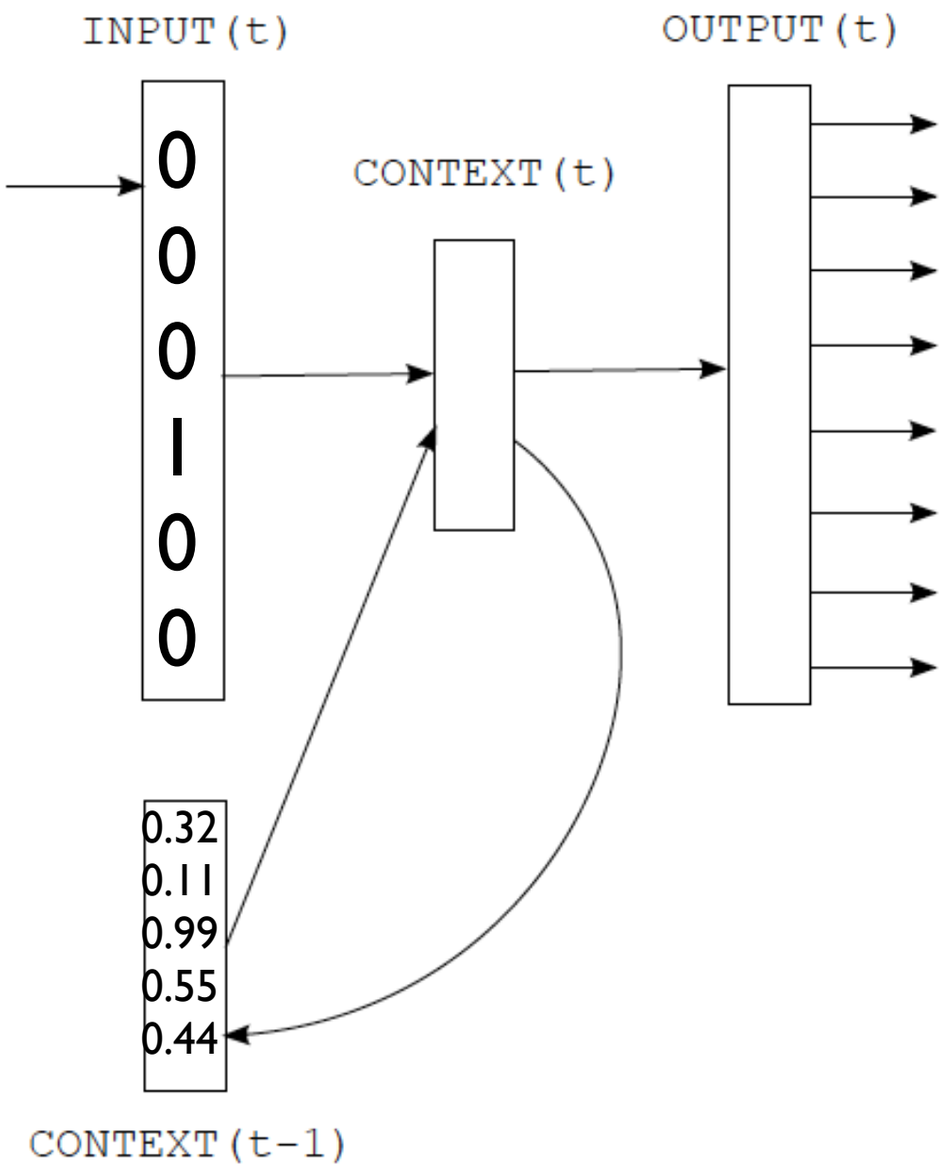
(2)
$$h_j(t) = F(a_j(t)), \quad j = 1, \ldots, n_H$$

(3)
$$b_j(t) = \sum_{i=1}^{n_H} \beta_{ji}\, h_i(t), \quad j = 1, \ldots, n_J$$

(4)
$$o_j(t) = G(b_j(t)), \quad j = 1, \ldots, n_J$$

Decoding:
4 easy steps!

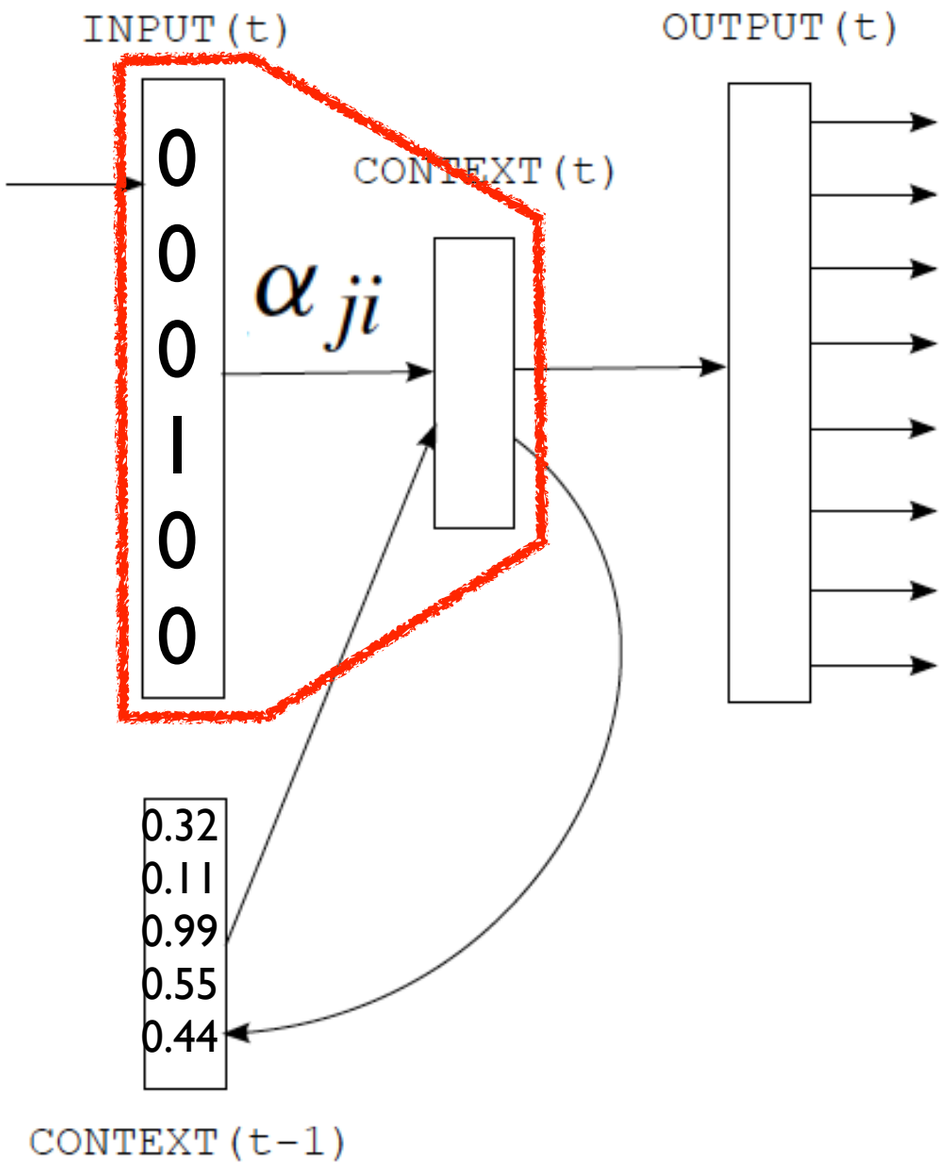
Recurrent Neural Nets



(1)
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H$$

(1) Calculate input to the hidden layer based on:

Recurrent Neural Nets

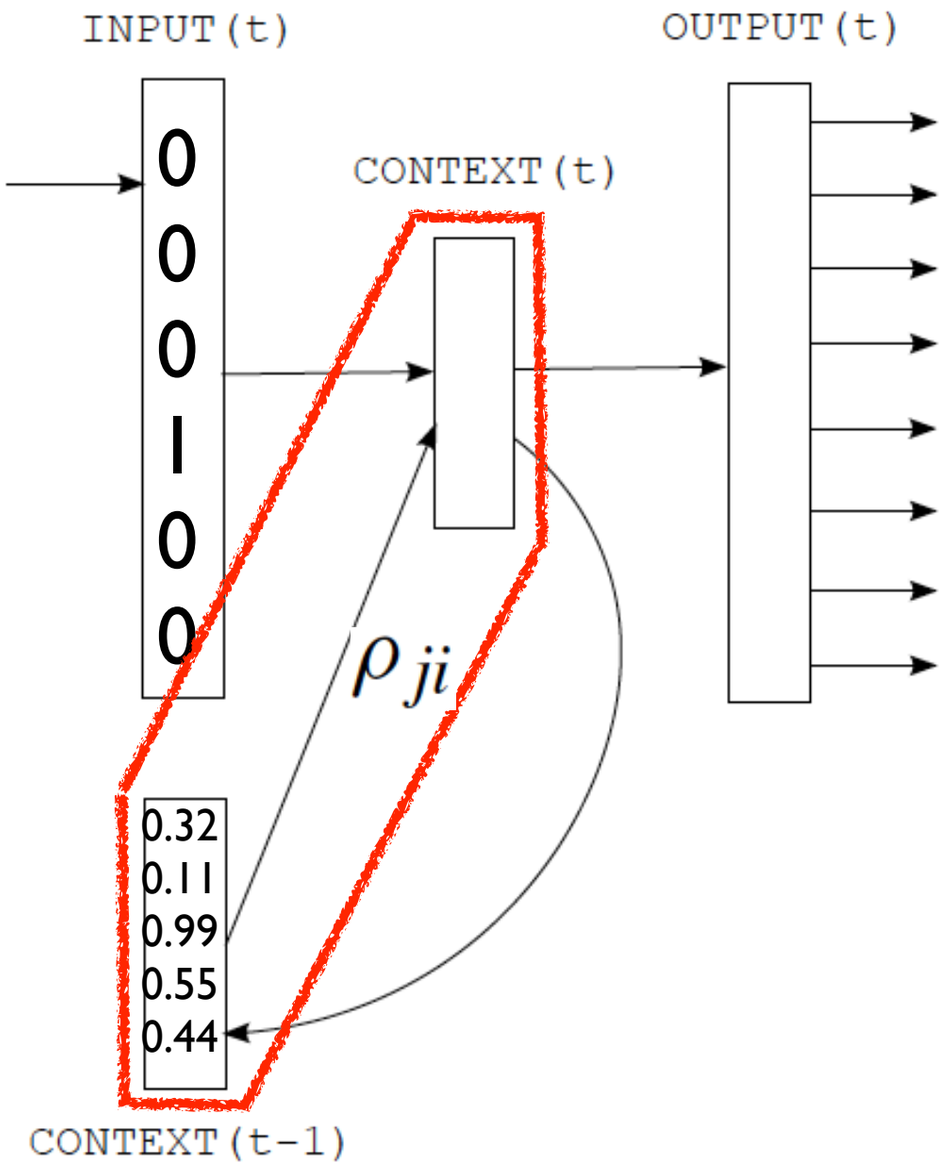


$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H \tag{1}$$

(1) Calculate input to the hidden layer based on:

i) input layer and weights between input and hidden layer

Recurrent Neural Nets



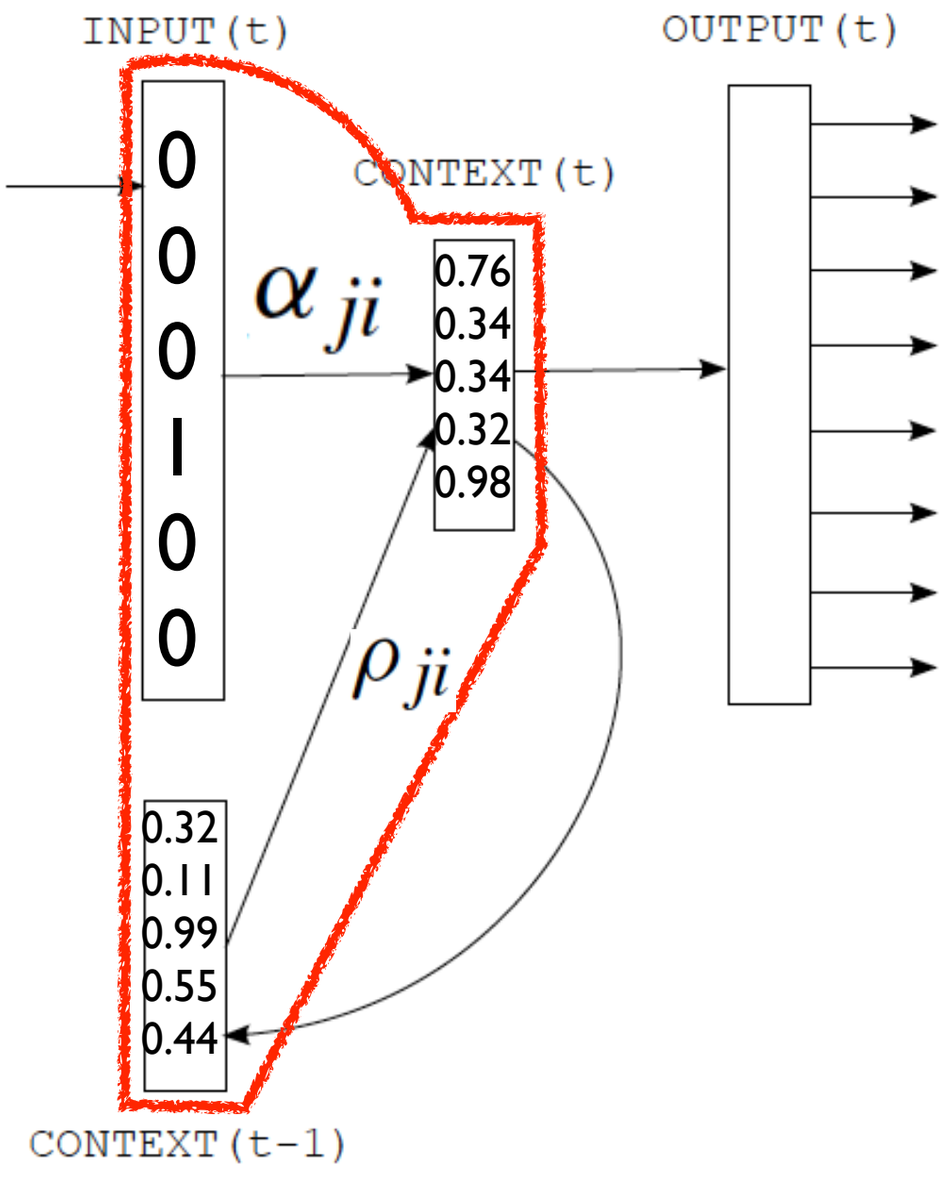
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H \tag{1}$$

(1) Calculate input to the hidden layer based on:

i) input layer and weights between input and hidden layer

ii) context (t-1) nodes and weights between context (t-1) nodes and hidden layer

Recurrent Neural Nets



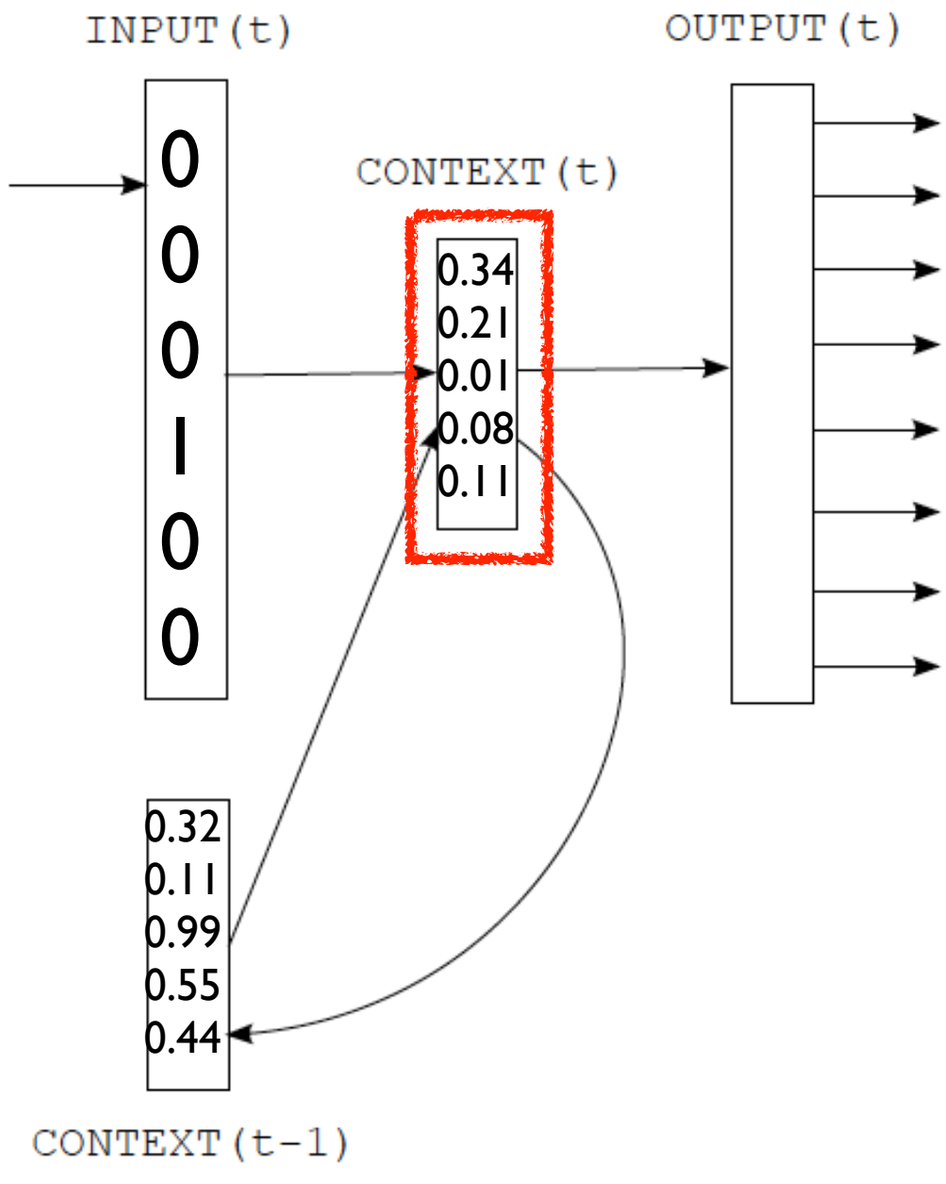
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H \tag{1}$$

(1) Calculate input to the hidden layer based on:

i) input layer and weights between input and hidden layer

ii) context (t-1) nodes and weights between context (t-1) nodes and hidden layer

Recurrent Neural Nets

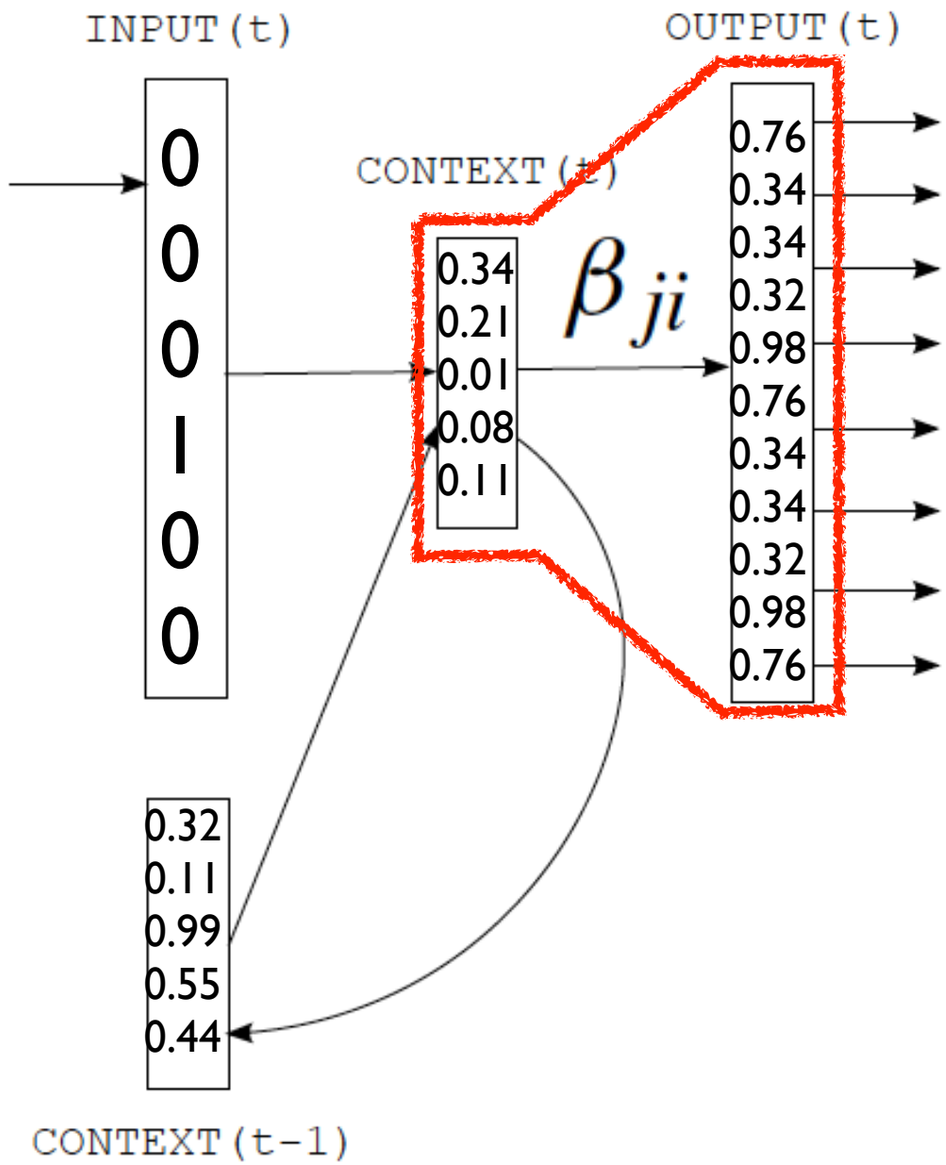


(2)
$$h_j(t) = F(a_j(t)), \quad j = 1, \ldots, n_H$$

(2) Calculate the activation of the hidden layer's nodes.

F is some non-linear function, usually the *sigmoid*

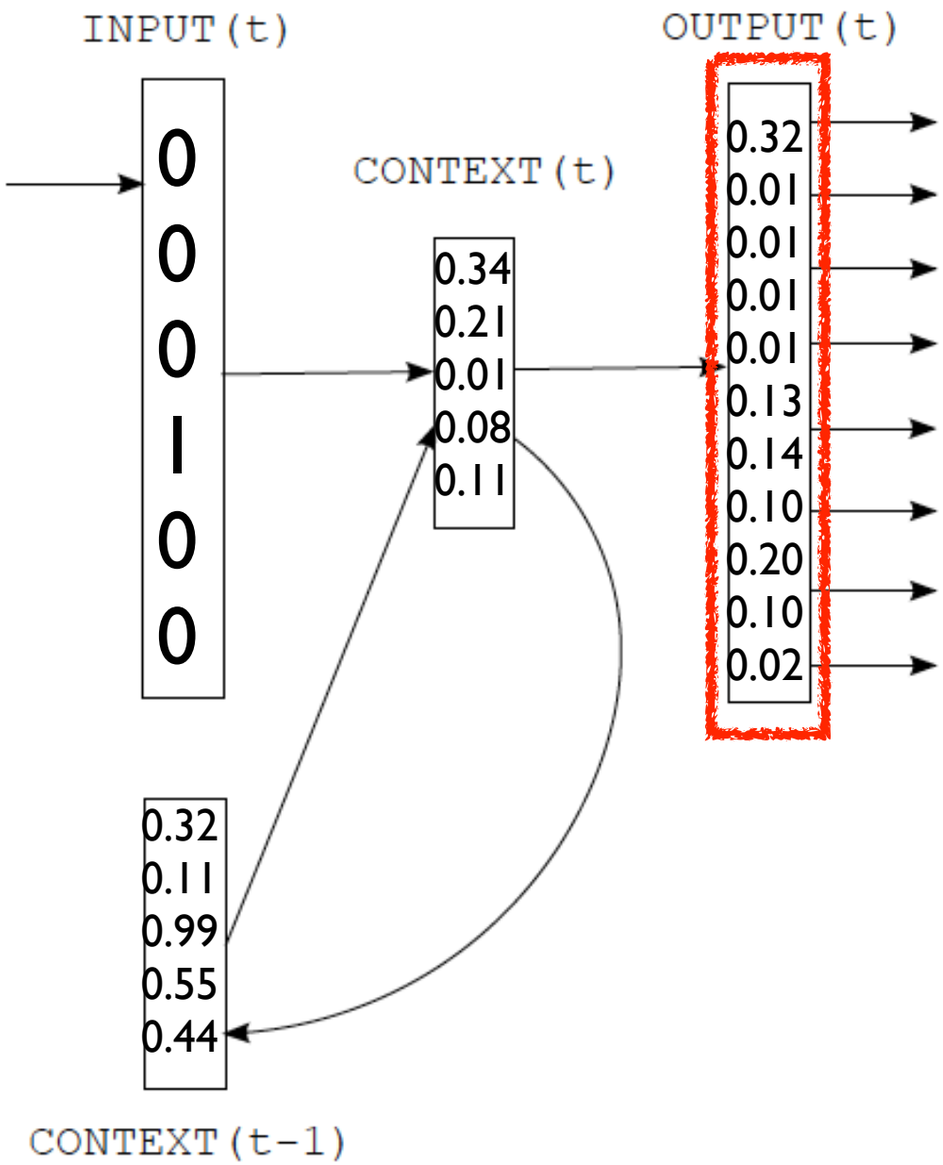
Recurrent Neural Nets



$$b_j(t) = \sum_{i=1}^{n_H} \beta_{ji}\, h_i(t), \quad j = 1, \ldots, n_J \tag{3}$$

(3) Calculate input to the output layer, based on new hidden layer's activations and weights from hidden layer to output layer

Recurrent Neural Nets

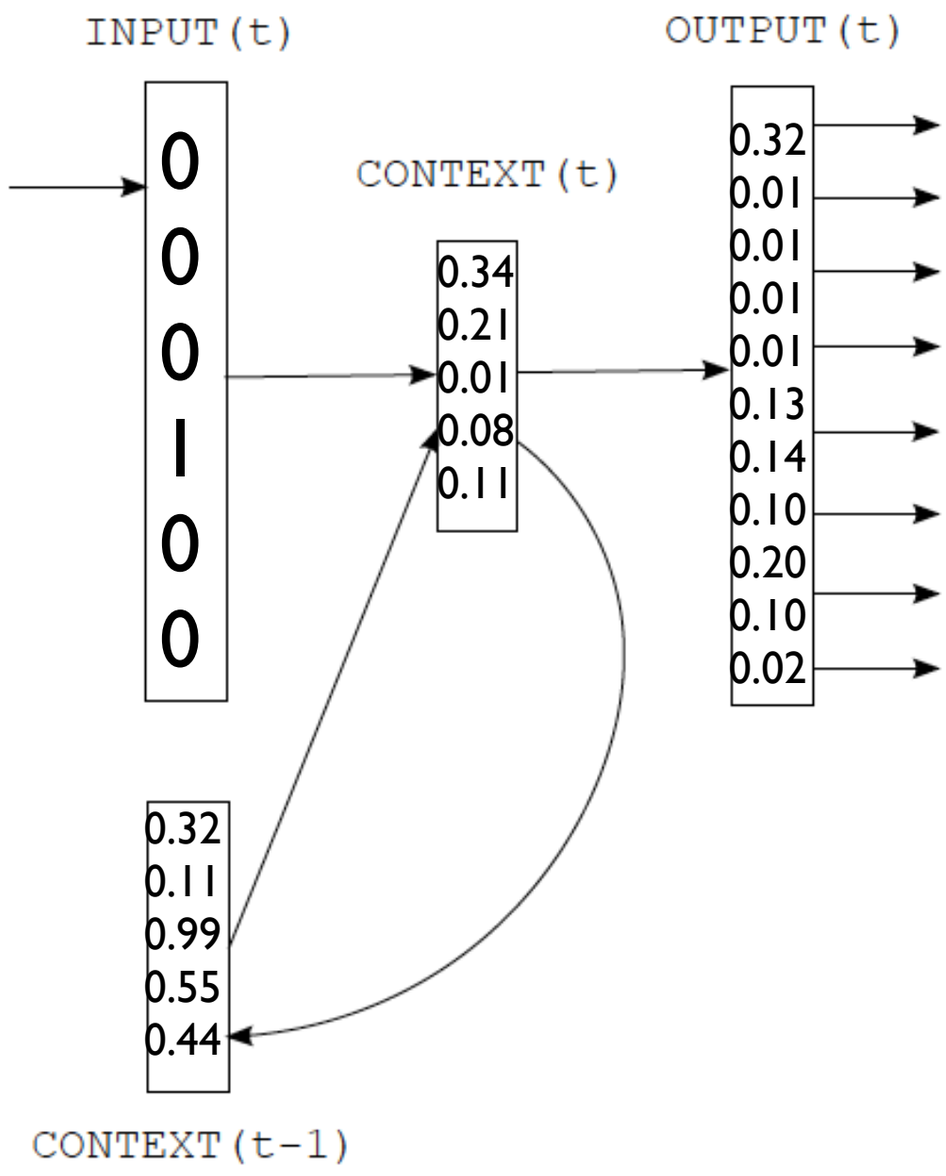


$$o_j^{(4)}(t) = G(b_j(t)), \quad j = 1, \ldots, n_J$$

(4) Calculate activation of the output layer's nodes.

G is some function, often the *softmax*, which compresses real valued vector into probability distribution

Recurrent Neural Nets



(1)
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H$$

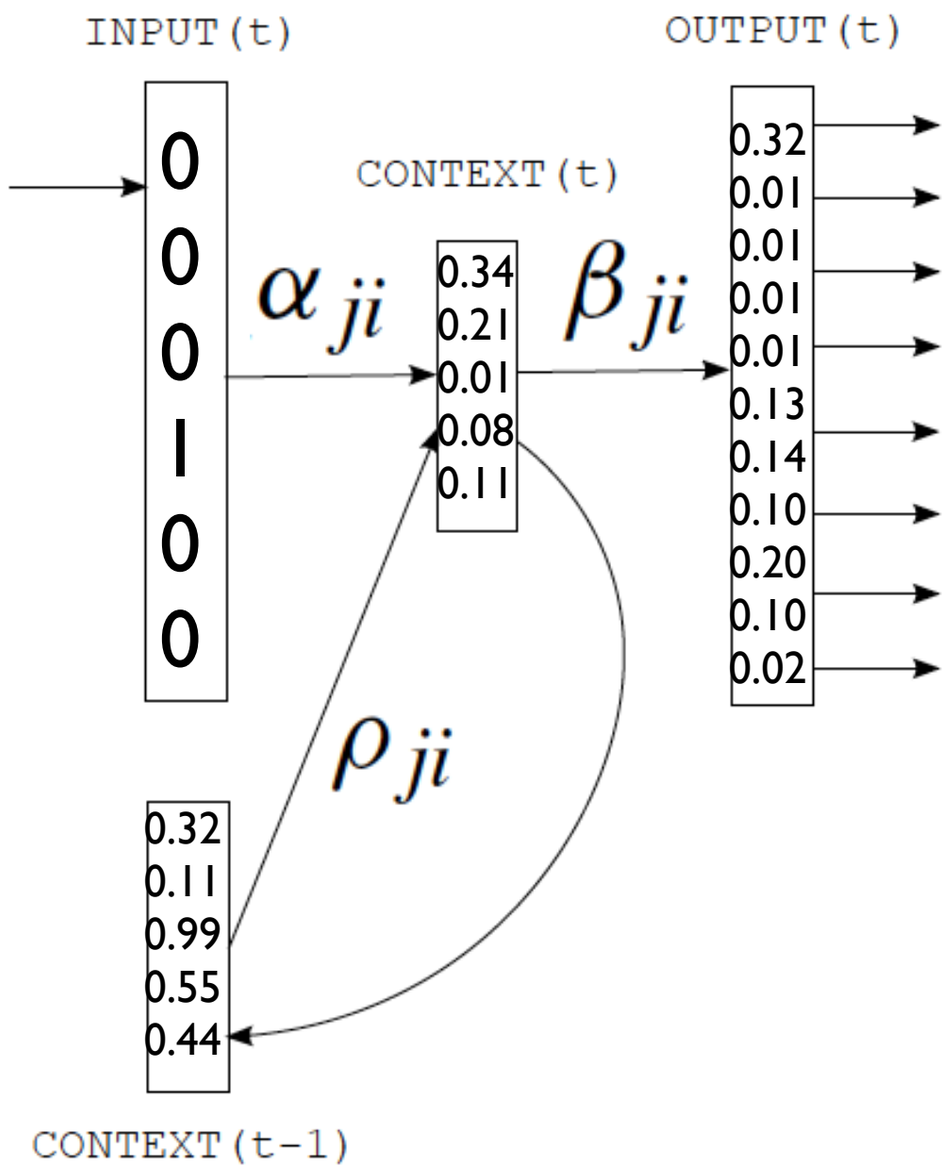
(2)
$$h_j(t) = F(a_j(t)), \quad j = 1, \ldots, n_H$$

(3)
$$b_j(t) = \sum_{i=1}^{n_H} \beta_{ji}\, h_i(t), \quad j = 1, \ldots, n_J$$

(4)
$$o_j(t) = G(b_j(t)), \quad j = 1, \ldots, n_J$$

Decoding:
4 easy steps!

Recurrent Neural Nets



$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji}\, x_i(t) + \sum_{i=1}^{n_H} \rho_{ji}\, h_i(t-1), \quad j = 1, \ldots, n_H \tag{1}$$

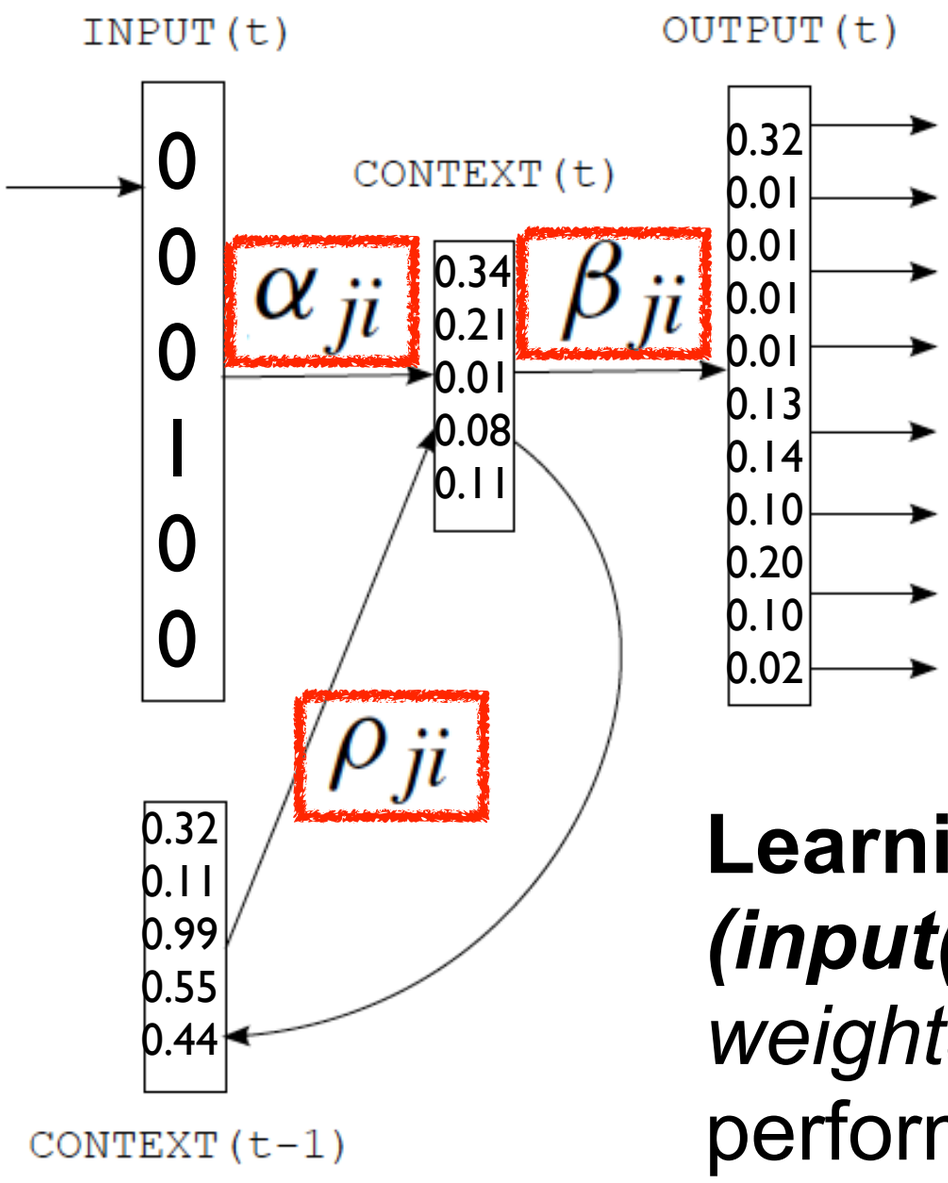
$$h_j(t) = F(a_j(t)), \quad j = 1, \ldots, n_H \tag{2}$$

$$b_j(t) = \sum_{i=1}^{n_H} \beta_{ji}\, h_i(t), \quad j = 1, \ldots, n_J \tag{3}$$

$$o_j(t) = G(b_j(t)), \quad j = 1, \ldots, n_J \tag{4}$$

Learning: given training data pairs ***(input(t),target(t))*** learn the *three sets of weights* which maximise decoding performance on some task.

Recurrent Neural Nets



(1)
$$a_j(t) = \sum_{i=1}^{n_I} \alpha_{ji} x_i(t) + \sum_{i=1}^{n_H} \rho_{ji} h_i(t-1), \quad j = 1, \ldots, n_H$$

(2)
$$h_j(t) = F(a_j(t)), \quad j = 1, \ldots, n_H$$

(3)
$$b_j(t) = \sum_{i=1}^{n_H} \beta_{ji} h_i(t), \quad j = 1, \ldots, n_J$$

(4)
$$o_j(t) = G(b_j(t)), \quad j = 1, \ldots, n_J$$

Learning: given training data pairs ***(input(t),target(t))*** learn the *three sets of weights* which maximise decoding performance, usually by minimising **negative log loss (softmax cross-entropy)**.

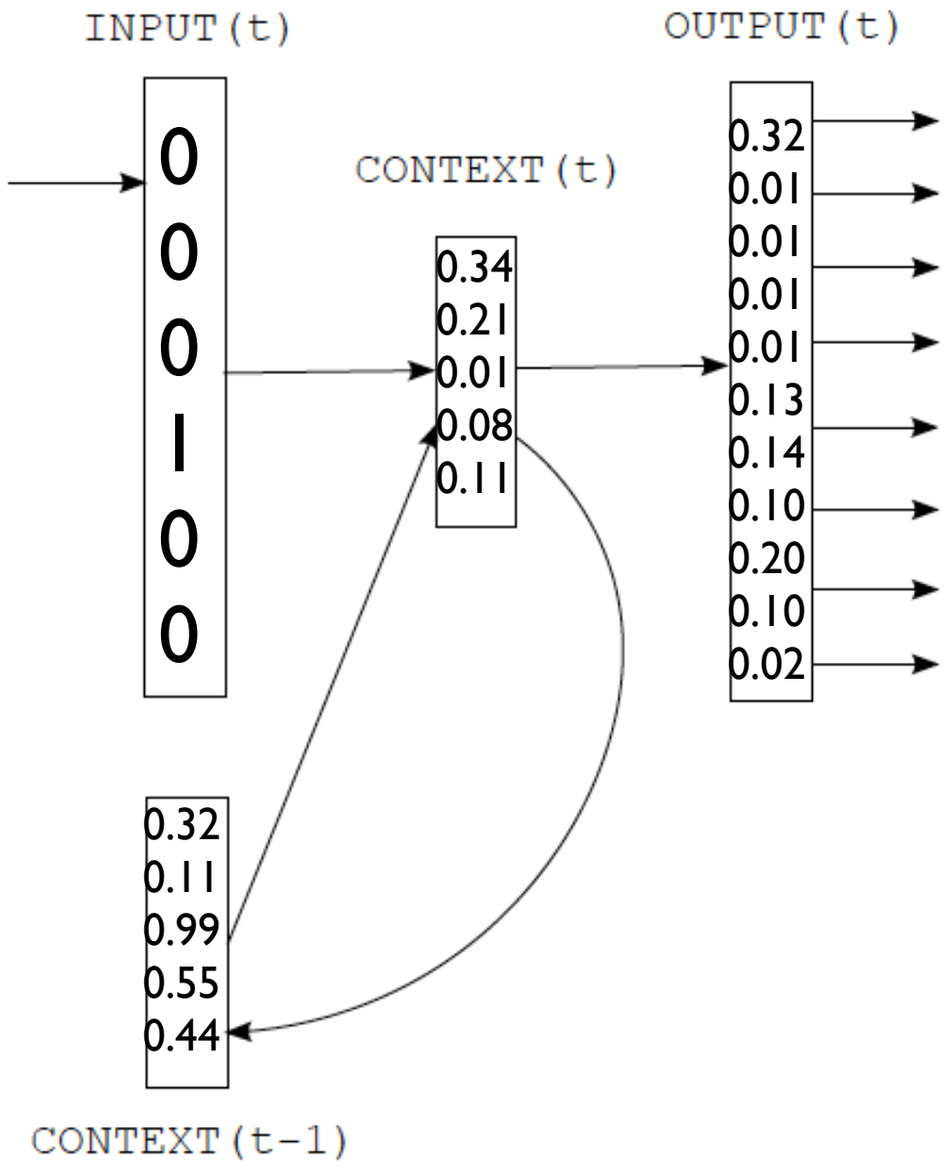
OUTLINE

1) The arrival of deep neural networks in NLP
2) Logistic regression as a one-layered neural net (perceptron)
3) Learning weights with stochastic gradient descent
4) Adding layers for ‘deep learning’ with backpropagation
5) Recurrent neural nets for sequences
6) Application: language models and sequence tagging

RNNS for Language Models

john

w_t



likes

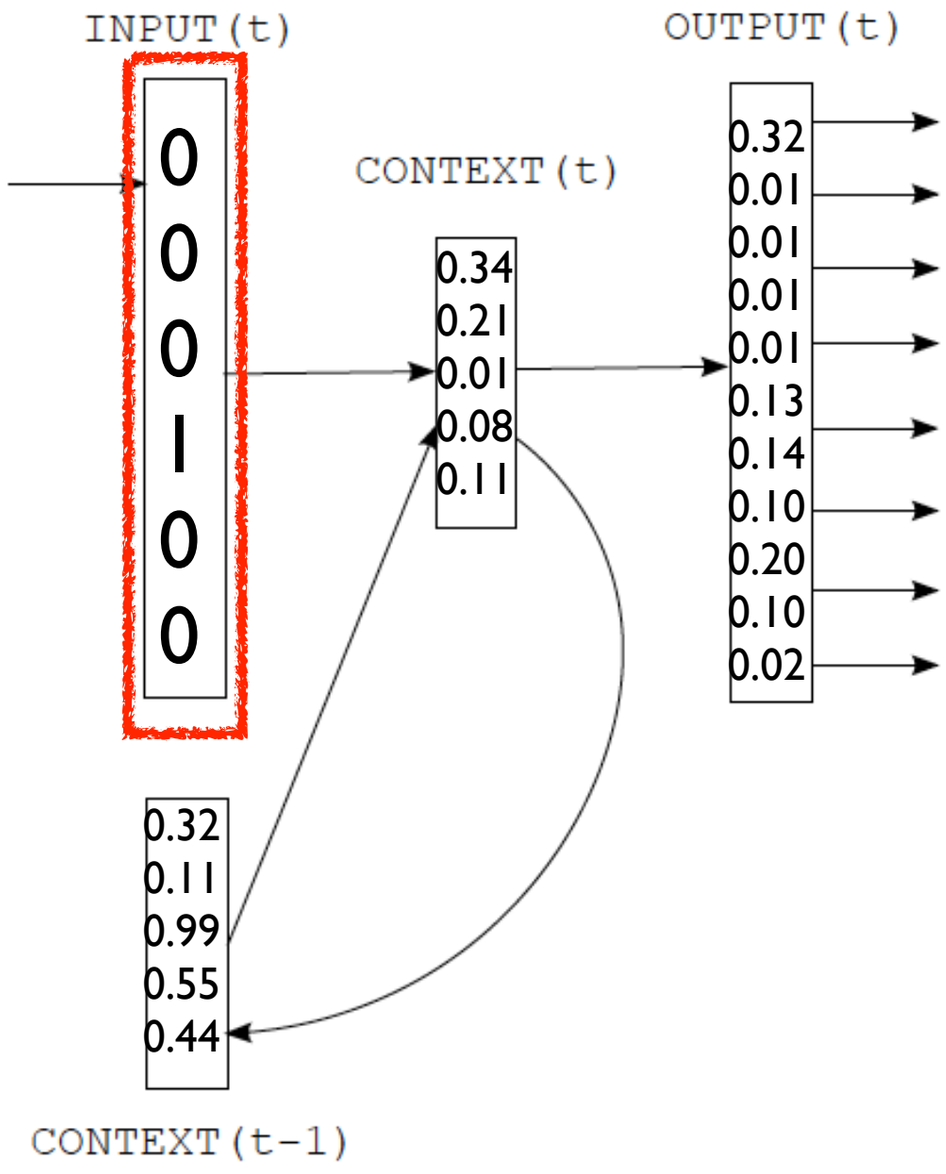
w_{t+1}

Task: maximise the probability of next word w_{t+1} given context of current word w_t and previous words before that!

RNNS for Language Models

john

w_t



likes

w_{t+1}

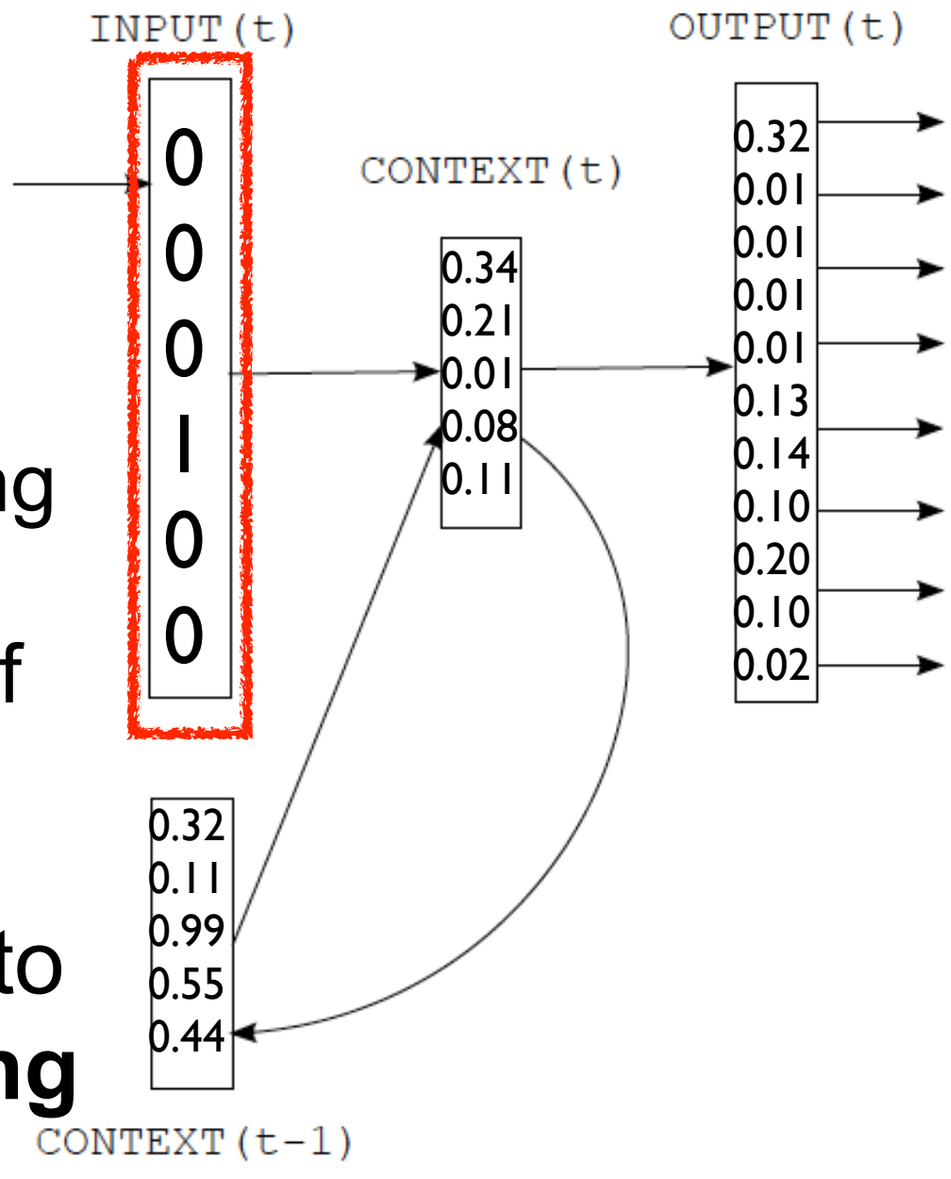
RNNS for Language Models

john

w_t

1-of-N encoding of word/ ‘1-hot’ vector of size **|V|**

Or indices map to **word embedding** of dimension **|e|**



likes

w_{t+1}

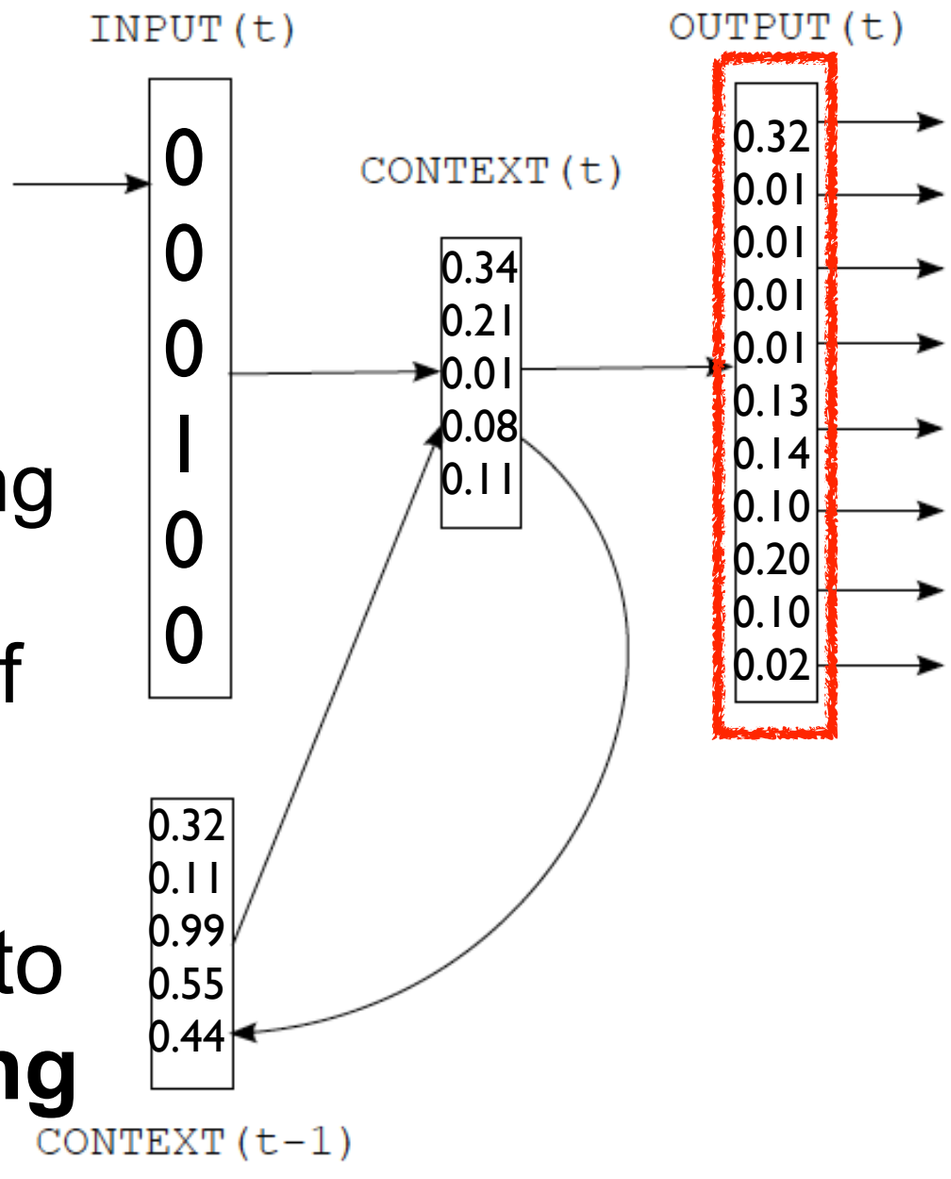
RNNS for Language Models

john

w_t

1-of-N encoding of word/ ‘1-hot’ vector of size **|V|**

Or indices map to **word embedding** of dimension **|e|**



likes

w_{t+1}

Probability distribution over the1-hot positions, again vector of size **|V|**

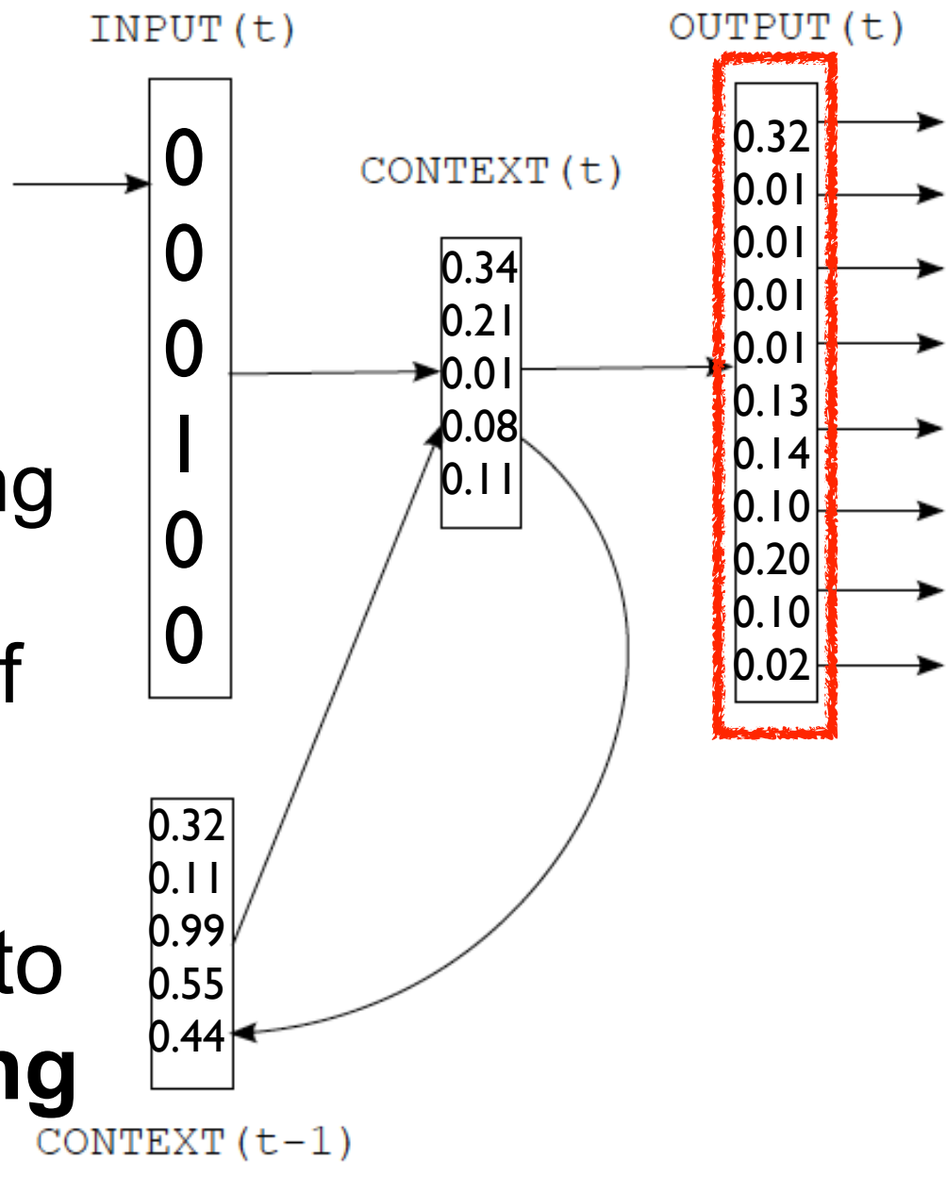
RNNS for Language Models

john

w_t

1-of-N encoding of word/ ‘1-hot’ vector of size **|V|**

Or indices map to **word embedding** of dimension **|e|**



likes

w_{t+1}

Probability distribution over the1-hot positions, again vector of size **|V|**

RNNS for Language Models

- RNNLMs learn to minimise the difference between their predictions of the next word and the actual next word until **stopping criterion** is met. **Validation set** often used during training to assess improvement.

- A suitable **error function** is ***softmax cross-entropy (negative log loss)*** *where* the lower the better- computed as an average over a **batch** of size *k* rather than after each word.

- Final performance is **evaluated** with *perplexity (PPL)* (proportional to cross-entropy) of test data

RNNS for Language Models

- **Backpropagation** used though graph potentially bigger than FNNs as it has to backpropagate through time steps (up to a fixed number, e.g. 10) in addition to the hidden layer weights for each step.

- Learning algorithm similar to feedforward nets: updates weights by **batch gradient descent** after each batch *k.* Surf down the error valley until error function cannot be reduced further.

- **Learning rate** part of the update function- again, choice important. It can be gradually reduced or be set to be constant.

RNNS for Language Models

- **Choices for designers/hyper-parameter selection:**
  - ***Number of hidden nodes*** (a few rules of thumb, generally should be less than input, recommended is square root of # training instances)
  - ***Activation functions*** Hidden layer activation ***F*** and output layer function ***G*** (sigmoid and soft max most common)
  - ***Error function*** (cross-entropy is a measure of divergence from target probability)
  - ***Stopping criterion*** the “good enough” criterion for completing training (fixed number of epochs, or several iterations)
  - ***Learning rate*** (if not fixed, then choose ***Momentum***)
  - ***Batch size*** (number of examples after which parameters are updated)

RNNS for Language Models

- **Potential problems for RNNs:**
  - ***Training time*** can take a long time. the important factors in the complexity are number of hidden nodes and |*V*|
  - ***Number of neurons*** *in practice no fixed rules of thumb to decide the size, often computational efficiency is considered for fitting into GPUs*
  - ***Small context in practice*** in reality, while context can span entire corpus, it normally only lasts 10 words max.
  - ***Vanishing gradient problem*** *- can be mitigated with ReLU nodes*

RNNS for Language Models

- **Successes:**
  - Mikolov et al. (2010) train RNN on and test on WSJ, and it outperforms existing models in terms of PPL:

	PPL		WER	
Model	RNN	RNN+KN	RNN	RNN+KN
KN5 - baseline	-	221	-	13.5
RNN 60/20	229	186	13.2	12.6
RNN 90/10	202	173	12.8	12.2
RNN 250/5	173	155	12.3	11.7
RNN 250/2	176	156	12.0	11.9
RNN 400/10	171	152	12.5	12.1
3xRNN static	151	143	11.6	11.3
3xRNN dynamic	128	121	11.3	11.1

RNNS for Language Models

- **Successes:**
  - Mikolov et al. (2010) train RNN on WSJ speech corpus, and as a language model in an ASR it outperforms existing models in terms of WER:

Model	DEV WER	EVAL WER
Baseline - KN5	12.2	17.2
Discriminative LM	11.5	16.9
Joint LM	-	16.7
Static 3xRNN + KN5	11.0	15.5
Dynamic 3xRNN + KN5	10.7	16.3

RNNS for Language Models

- **Successes:**
  - Mikolov et al. (2011) train RNN on some standard corpora, and it outperforms existing models in WER:

Model \ WSJ ASR task	Eval WER
KN5 Baseline	**17.2**
Discriminative LM	**16.9**
Recurrent NN combination	**14.4**

RNNS for NLU

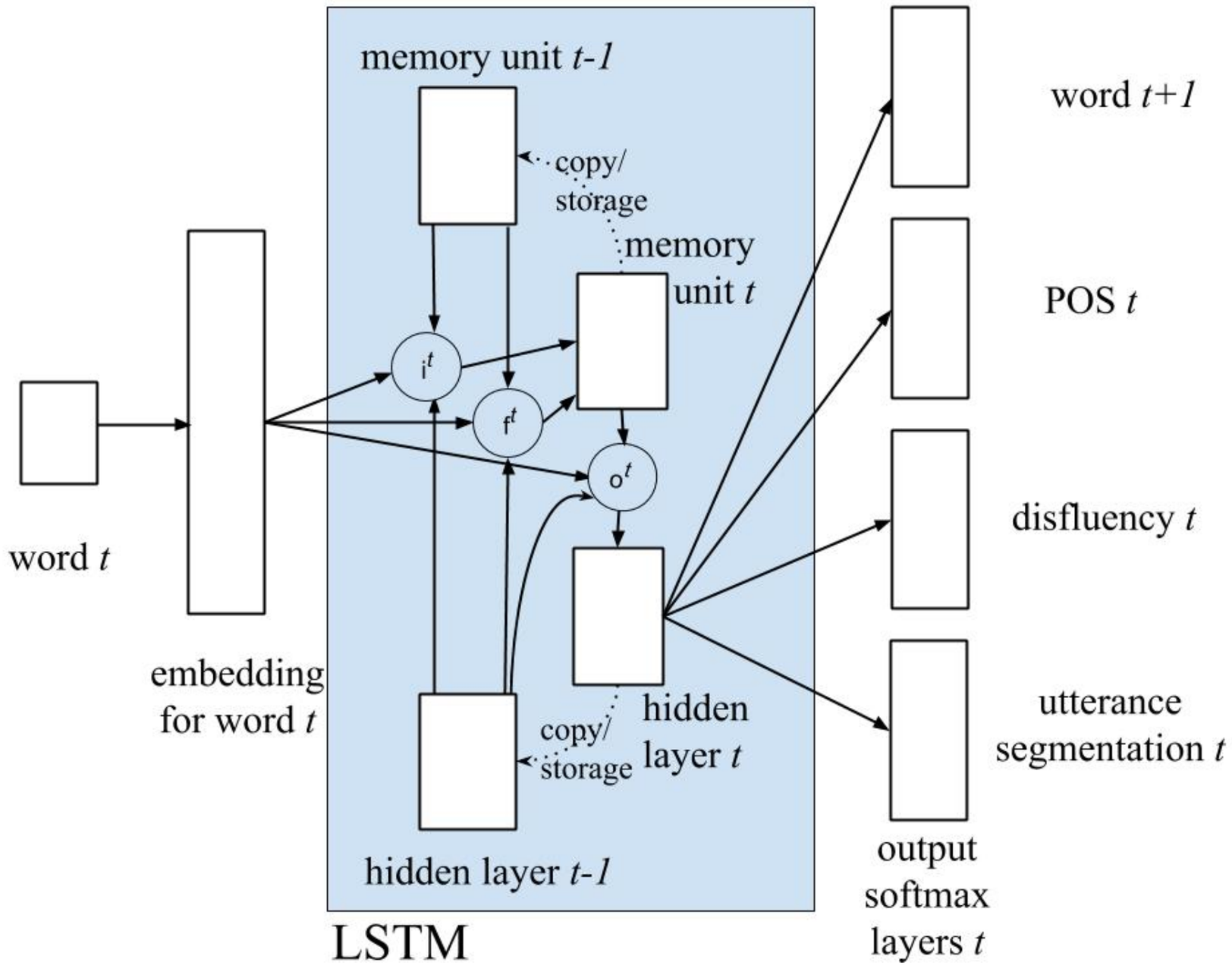
Task: maximise the probability of tag $\mathbf{t_t}$ given context of current word $\mathbf{w_t}$ and previous words before that

Sentence	*show*	*flights*	*from*	*Boston*	*to*	*New*	*York*	*today*
Slots/Concepts	O	O	O	B-dept	O	B-arr	I-arr	B-date
Named Entity	O	O	O	B-city	O	B-city	I-city	O
Intent	*Find_Flight*							
Domain	*Airline Travel*							

Table 1. ATIS utterance example IOB representation

- (Mensil et al. 2013) show improvement on state of the art sequence model (CRF)
- http://deeplearning.net/tutorial/rnnslu.html (Theano tutorial with good explanations)
- **See pytorch tutorial on NER- RNNs and LSTMs**

LSTMs: more memory



- (Rohanian and Hough 2020)

Just the beginning…

- This was just an introduction to the resurgence in neural nets in NLP in the early 2010's, giving some foundations.
- Since then there have been big developments of **powerful language models** which can improve their predictions using huge amounts of data (e.g. **word2vec**, **BERT**, **GPT-2/3** etc.)- these are used as pre-trained starting points for S.O.T.A. models on many NLP tasks!
- Important to try to get to grips with the fundamentals:
  - NNs can learn complex functions without specific feature engineering/manually specified feature interaction.
  - Why neural nets can be seen as a generalisation of logistic regression functions, though using them as units to become more powerful.
  - A well-defined loss function is important.
  - Different ways a neural net can be designed for a NL task - e.g. encoding memory through an RNN.

Reading

- Jurafsky and Martin (3rd Ed):
  - Chapter 5. “Logistic Regression”
  - Chapter 7. “Neural Networks and Neural Language Models”

- (Optional) de Mulder, W., Bethard, S., & Moens, M. F. (2015). A survey on the application of recurrent neural networks to statistical language modeling. *Computer Speech & Language*, *30*(1), 61-98

Reading (optional)

Mesnil, Grégoire, Xiaodong He, Li Deng and Yoshua Bengio. Investigation of Recurrent-Neural-Network Architectures and Learning Methods for Spoken Language Understanding. Interspeech, 2013.

Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. In *INTERSPEECH* (pp. 1045-1048).

Mikolov, T., Kombrink, S., Burget, L., Cernocky, J. H., & Khudanpur, S. (2011, May). Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on* (pp. 5528-5531). IEEE

Rumelhart, D. E., McClelland, J. L., & PDP Research Group. (1995). *Parallel distributed processing* (Vol. 1, p. 184). MIT press.