IN4080 – 2022 FALL

NATURAL LANGUAGE PROCESSING

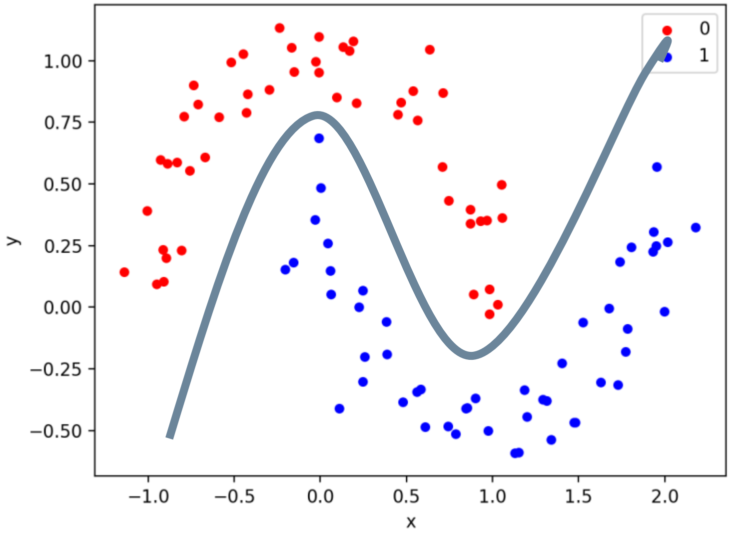
Jan Tore Lønning

Lecture 12, part 2, 10 Nov.

Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language models

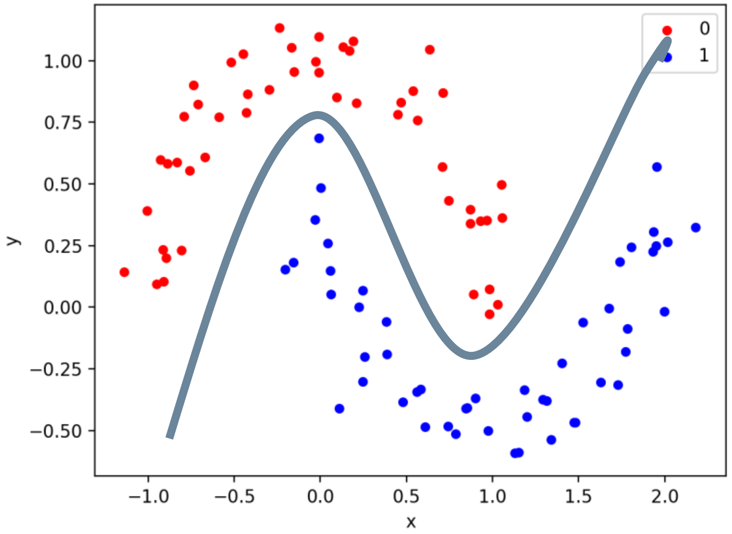
Non-linearity



w2=bad	neg	pos
w2=good	pos	neg
	w1≠ not	w1 = not

- Logistic regression is a linear classifier
- What to do with data that are far from linearly separable?

Alt. 1: Feature engineering



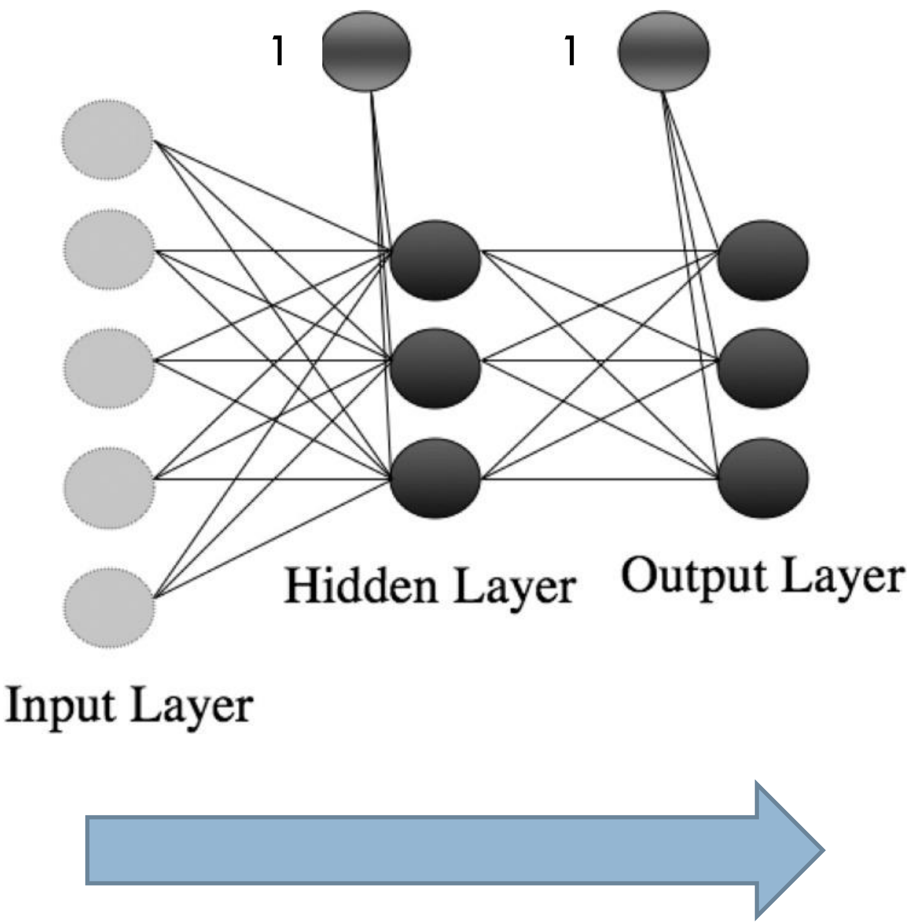
- In addition to x_1 and x_2 add e.g., the features
  - $x_1{}^2, x_2{}^2, x_1x_2, x_1{}^3, \ldots$

w2=bad	neg	pos
w2=good	pos	neg
	w1≠ not	w1 = not

- In addition to
  - $f_1 = w_1$ and $f_2 = w_2$
  - Add $f_3 = w_1w_2$

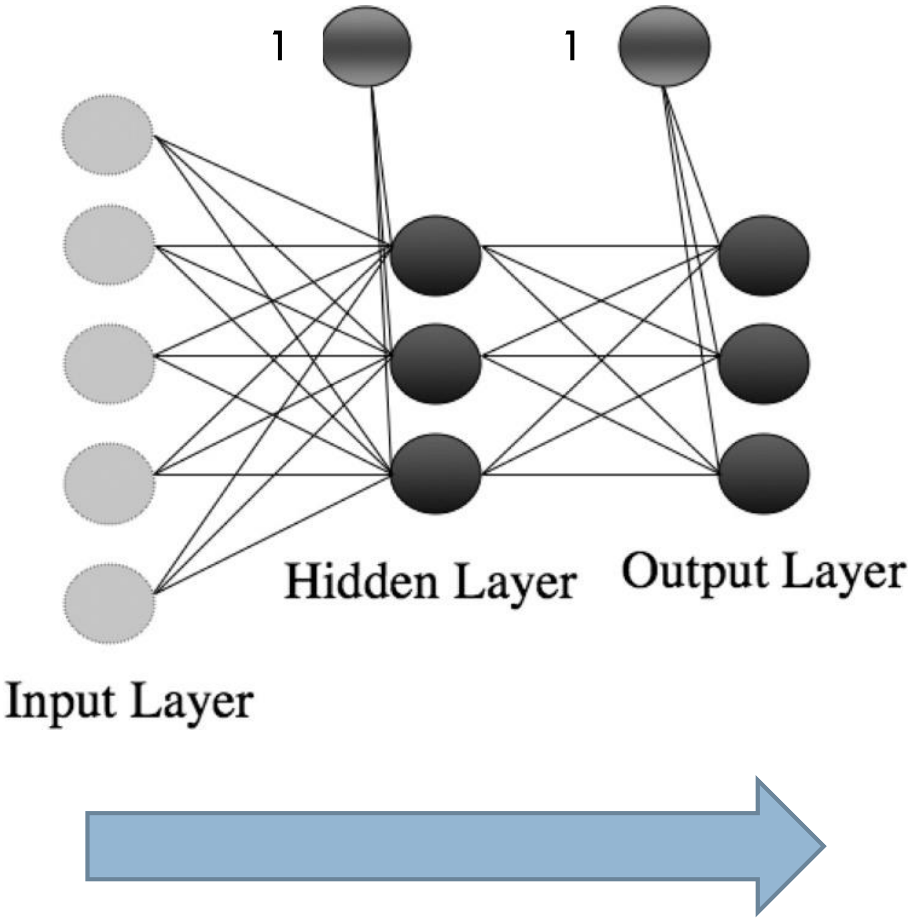
Artificial neural networks (= alt. 2)

- Inspired by the brain
  - neurons, synapses
- Does not pretend to be a model of the brain
- The simplest model is the
  - Feed forward network, also called
  - Multi-layer Perceptron



Feed forward network

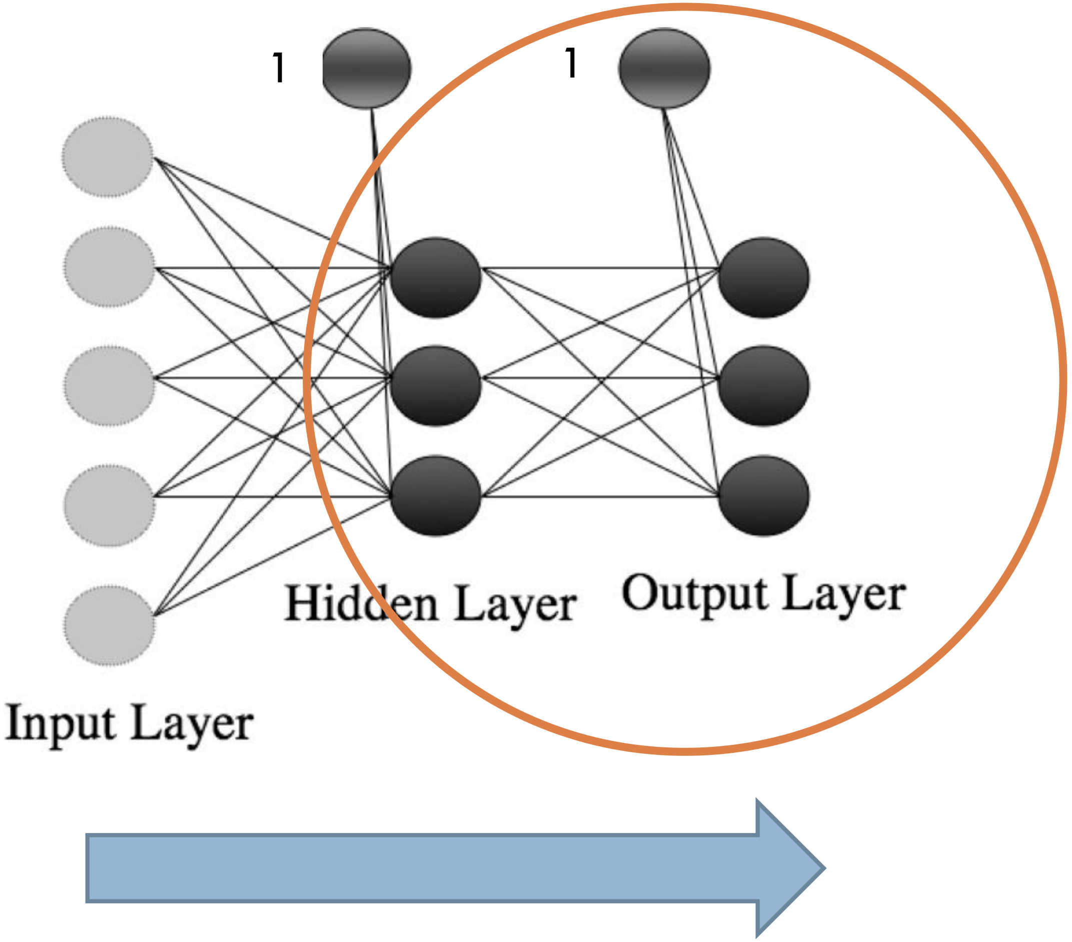
- An input layer
- An output layer: the predictions
- One or more hidden layers
- Connections from one layer to the next (from left to right)
- A weight at each connection



The output layer – as with no hidden layers

Alternatives

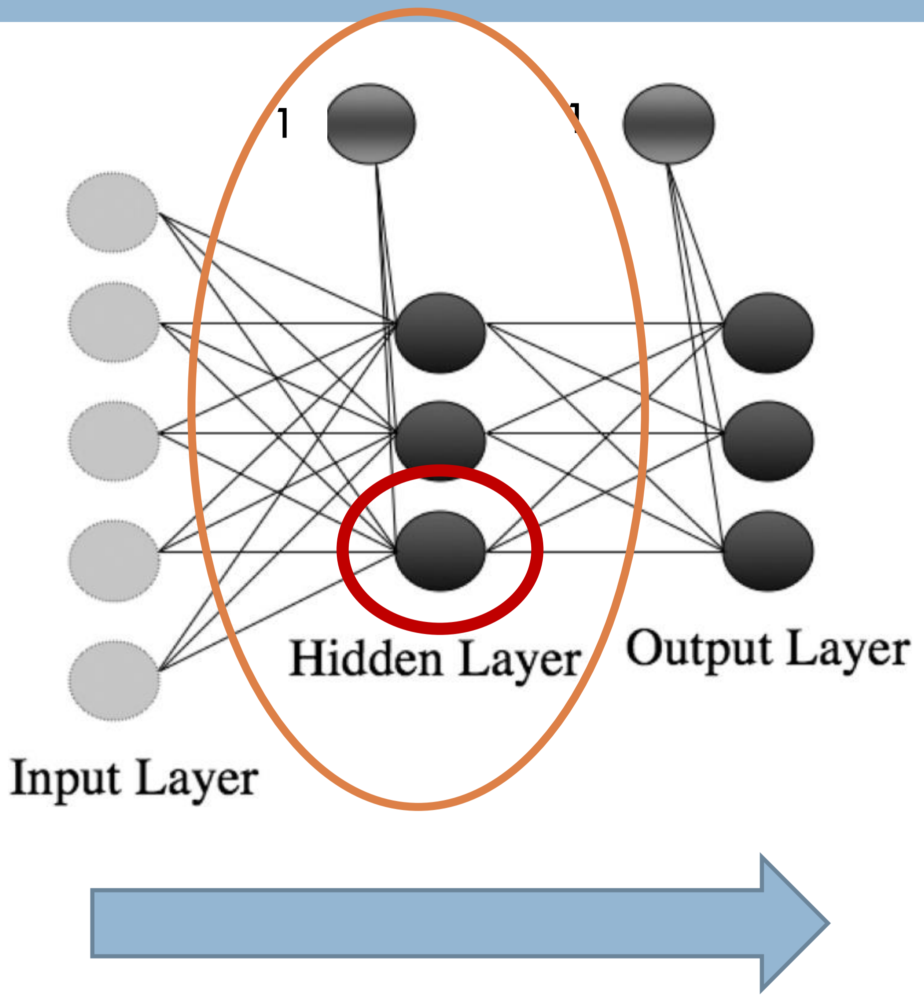
- Regression:
  - One node
  - No activation function
- Binary classifier:
  - One node
  - Logistic activation function
- Multinomial classifier
  - Several nodes
  - Softmax
- + more alternatives
- Choice of loss function depends on task



What is new

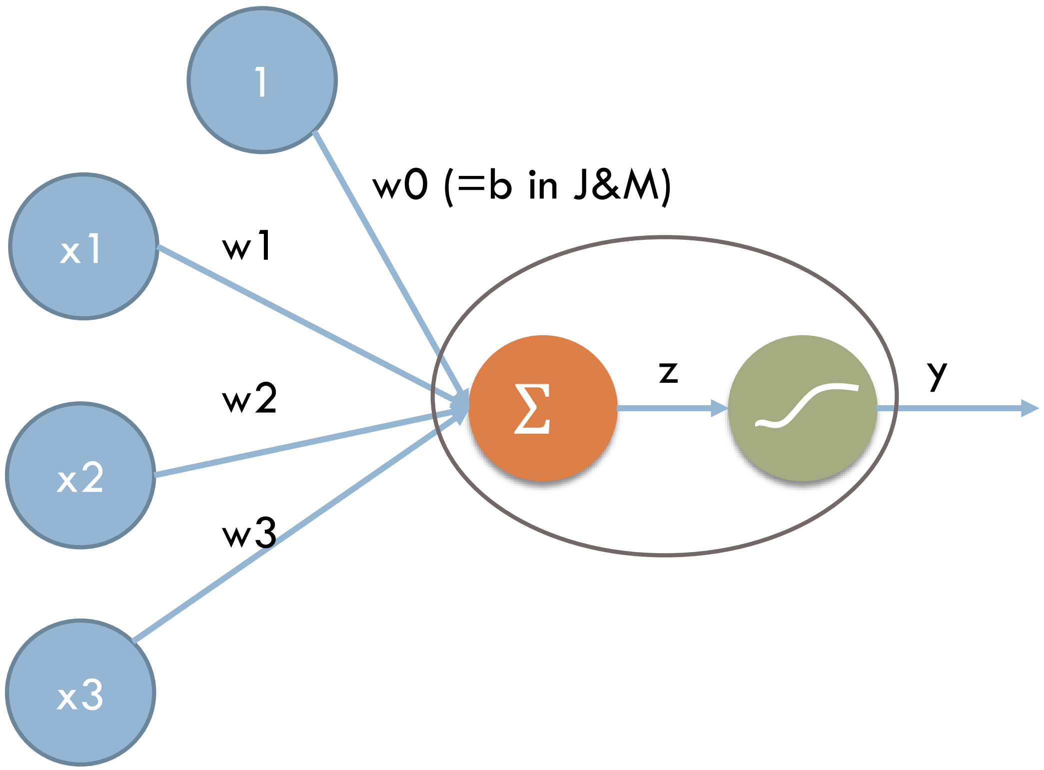
One or more hidden layers

What happens in the hidden layers



The hidden nodes

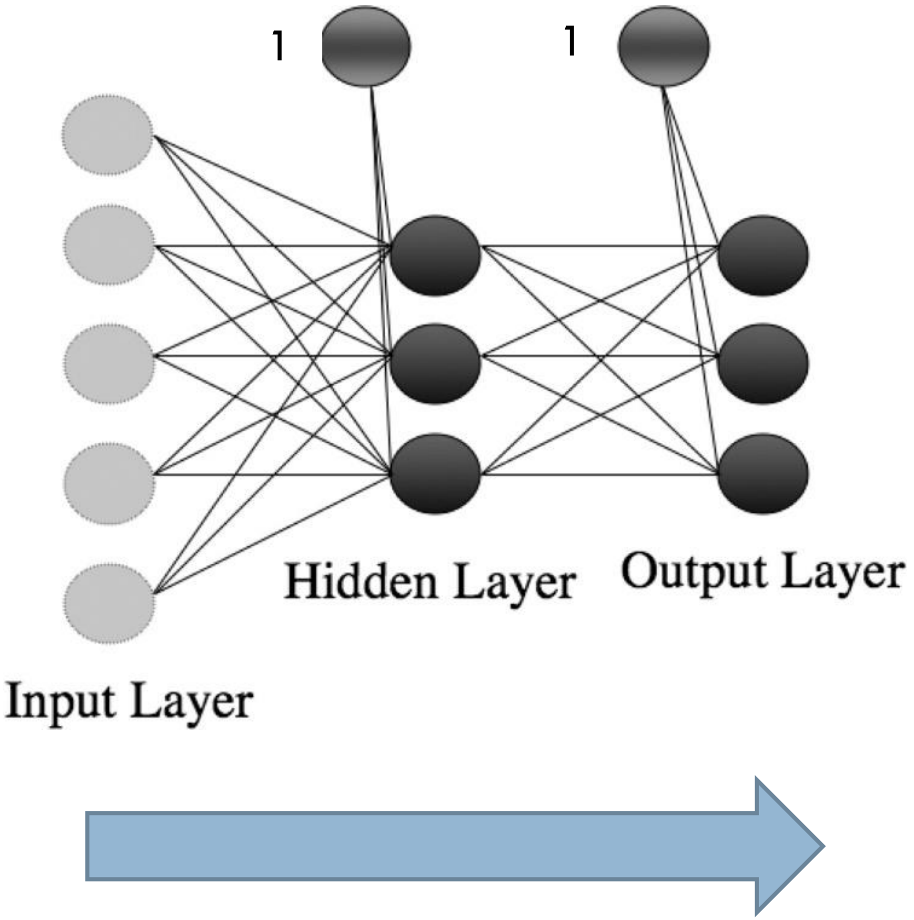
- Each hidden node is like a small logistic regression:
  - First sum of weighted inputs :
    - $z = \sum_{i=0}^{m} w_i x_i = \boldsymbol{w} \cdot \boldsymbol{x}$
  - Then the result is run through an activation function, e.g. σ
    - $y = \sigma(z) = \frac{1}{1+e^{-\vec{w}\cdot\vec{x}}}$



It is the non-linearity of the activation function which makes it possible for MLP to predict non-linear decision boundaries

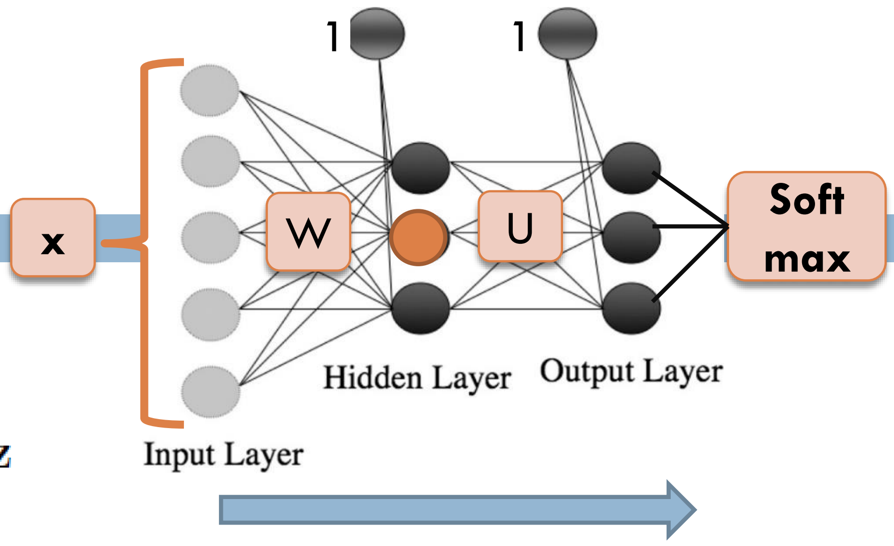
Forward

- Applying the network:
  - Start with the input vector
  - Run it step-by-step through the network



Forward

$$W\mathbf{x} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,n} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1} & w_{m,2} & \cdots & w_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_m \end{bmatrix} = \mathbf{z}$$



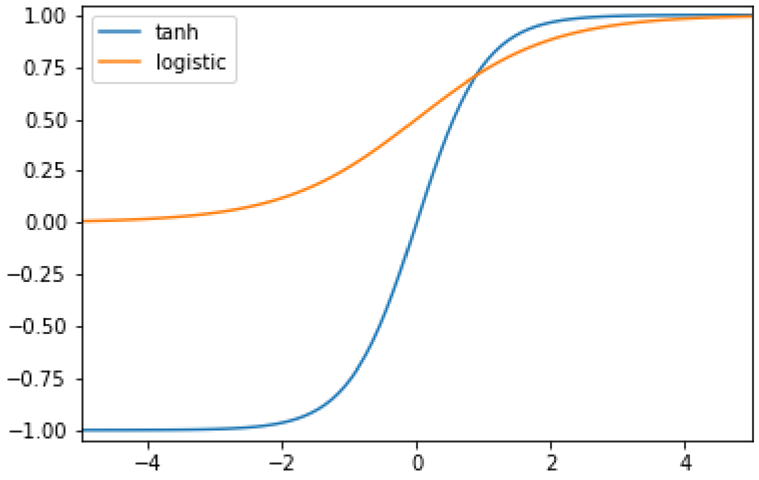
- Each layer can be considered a vector
- The connections between the layers: a matrix
- Running it through the connections: matrix multiplication

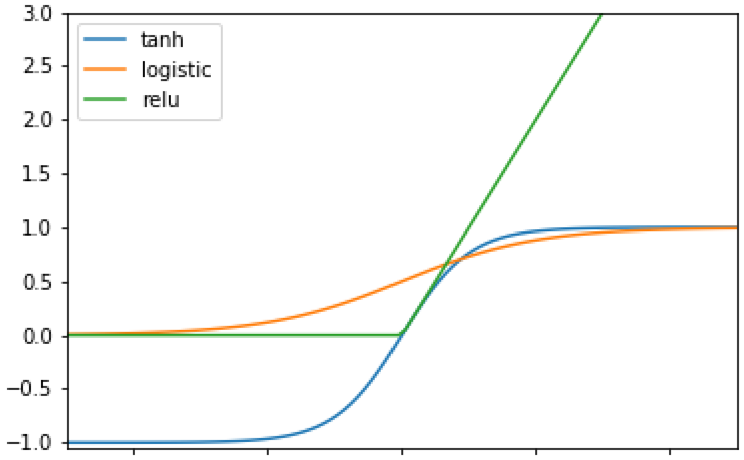
Example network:

- $\boldsymbol{h} = \sigma(W\boldsymbol{x} + \boldsymbol{b1})$
- $\boldsymbol{z2} = U\boldsymbol{h} + \boldsymbol{b2}$
- $\boldsymbol{y} = \text{softmax}(\boldsymbol{z2})$

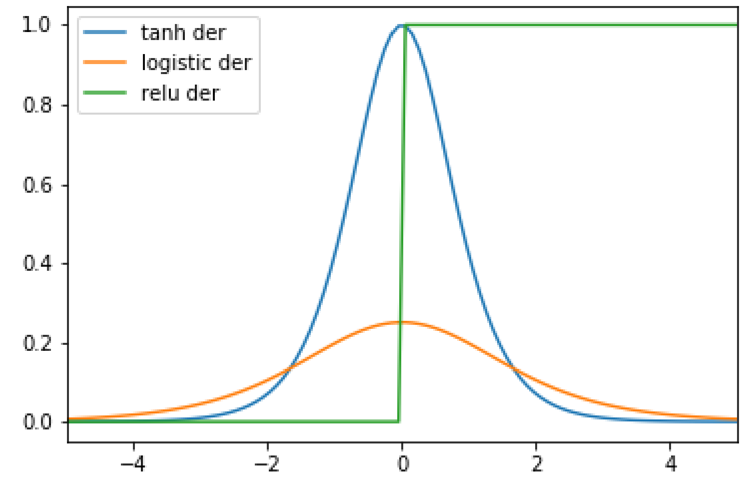
Beware: Jurafsky and Martin use $w_{i,j}$ where Marsland, IN3050, uses $w_{j,i}$
Marsland, and Goldberg (IN5550): $\boldsymbol{h} = \sigma(\boldsymbol{x}W + \boldsymbol{b})$, where $\boldsymbol{x}$ is a row vector

Alternative activation functions



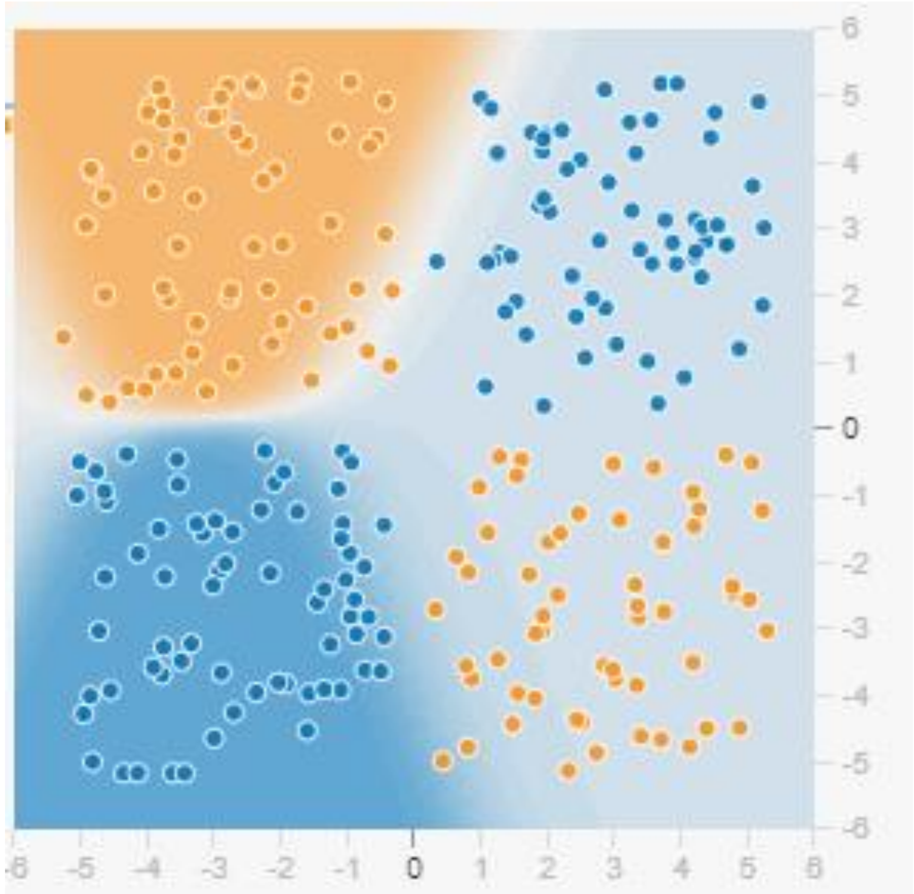


- There are alternative activation functions:
  - $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - $ReLU(x) = \max(x, 0)$
- ReLU is the preferred method in hidden layers in deep networks



Demo

- https://playground.tensorflow.org



Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language models

Computational graphs

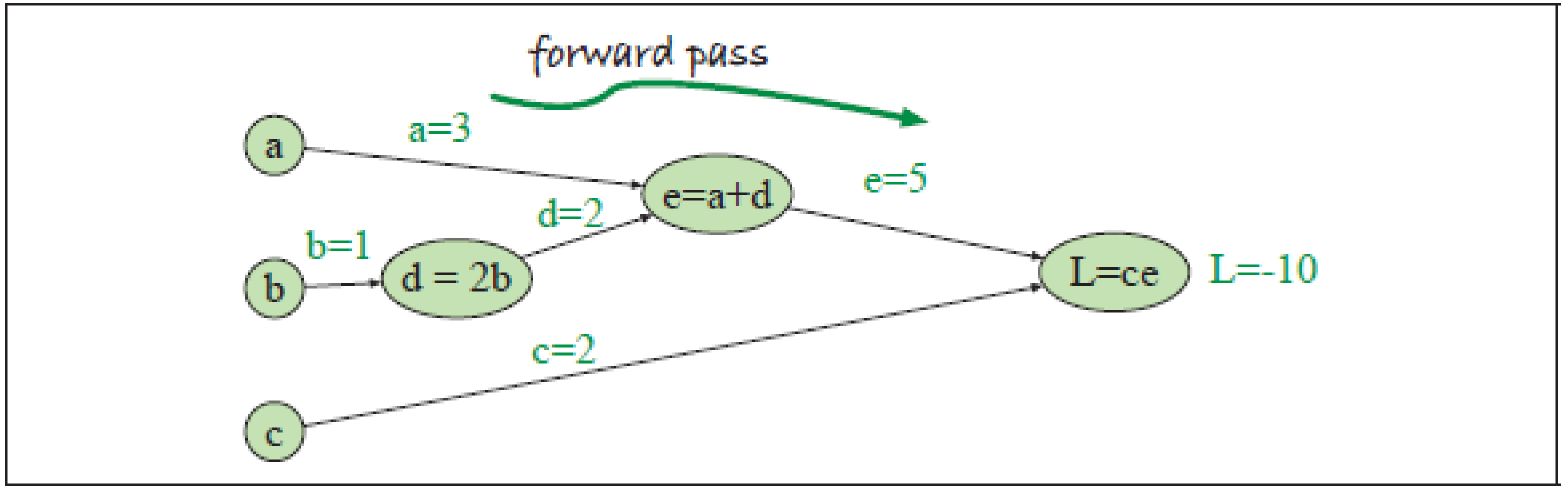


Figure 7.14 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

From J&M, 3.ed., 2021

- A convenient tool for describing composite functions
- And follow the partial derivatives backwards
- There are tools that let us specify the computations at an high-level as graphs
- In particular useful for "hiding" vectors, matrices, tensors
- After you have specified the graph, the tool computes the derivatives

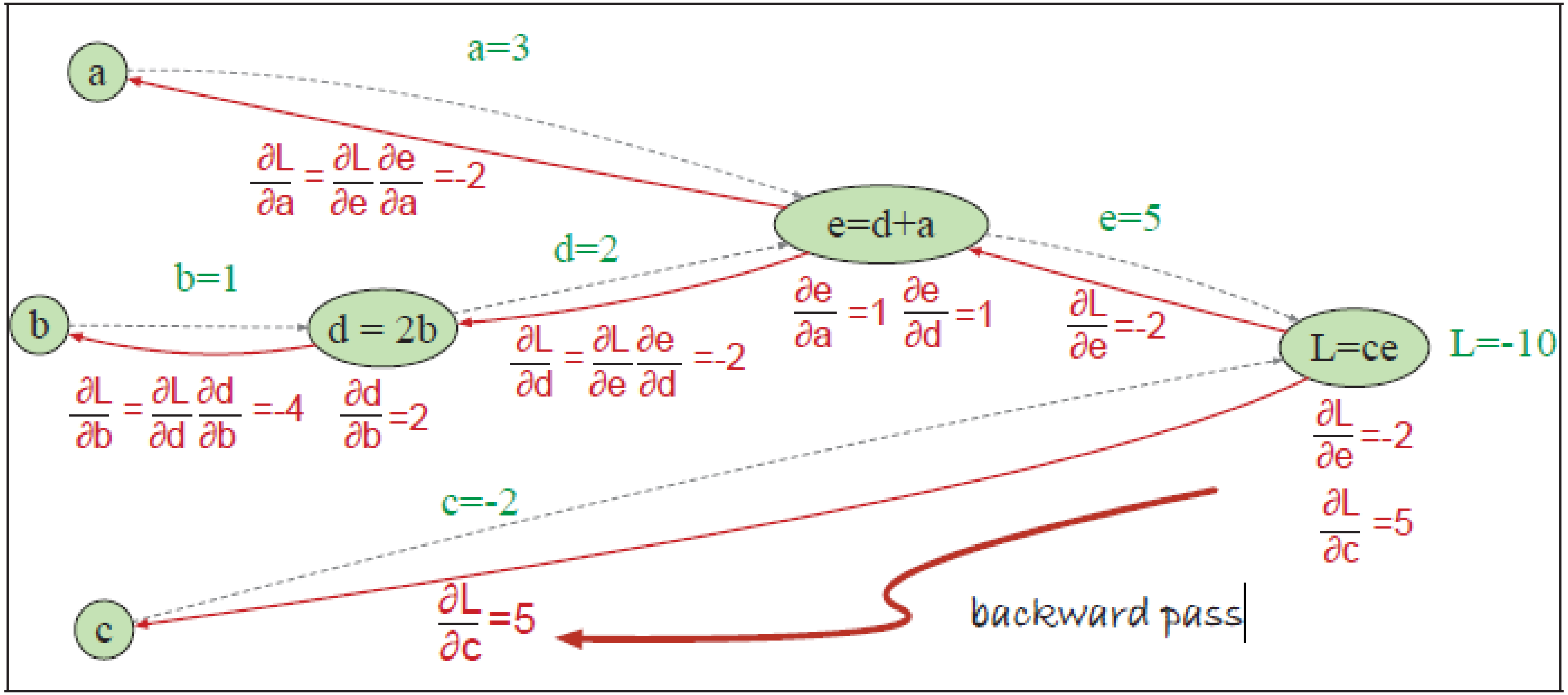


Figure 7.16 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

From J&M, 3.ed., 2021

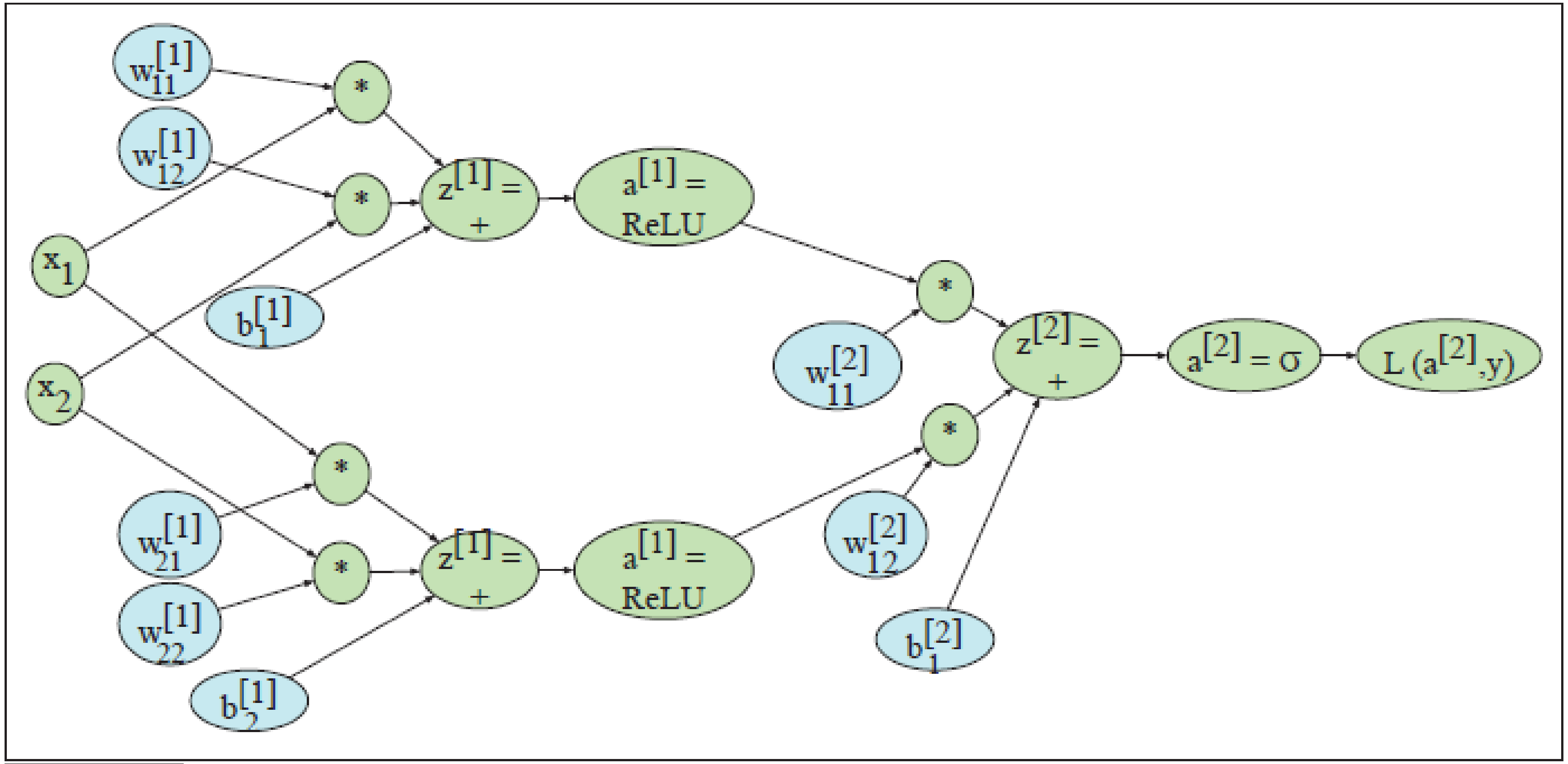


Figure 7.17 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.

From J&M, 3.ed., 2021

How would you draw this if x has dim 100,000 and there are 3 million parameters (weights)?

Using vector notation

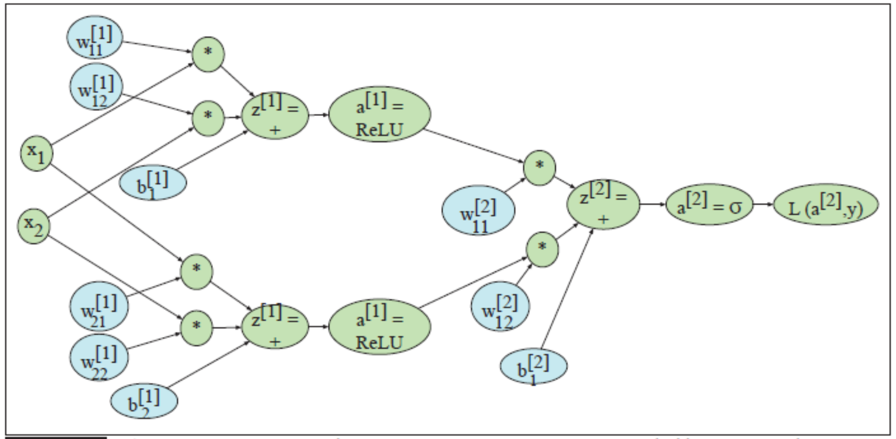
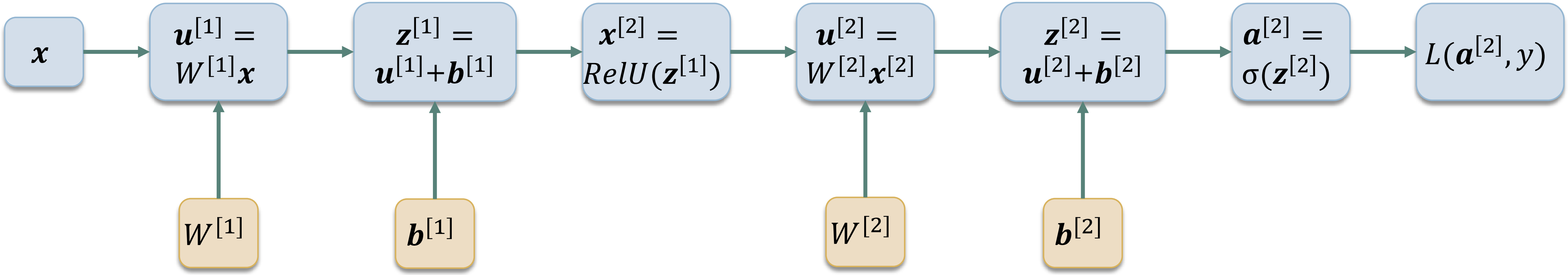


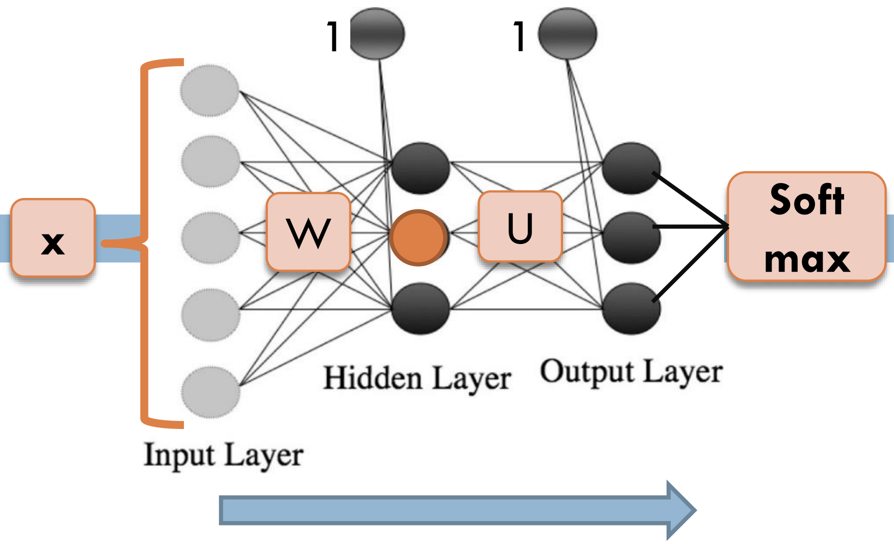
Figure 7.17 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.



Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language models

Learning



As we have seen for logistic regression

- Introduce a loss function: $L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
- Update each weight in each layer, e.g., $w_{i,j}$ according to its contribution to the loss
  - $w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial}{\partial w_{i,j}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
- Calculate the partial derivatives using the chain rule
  - "Follow the network backwards collecting partial derivatives along the path"

Example network:

- $\boldsymbol{h} = \sigma(W\boldsymbol{x} + b)$
- $\boldsymbol{z} = U\boldsymbol{h}$
- $\boldsymbol{y} = \text{softmax}(\boldsymbol{z})$

Log.Reg. Update one observation (remember?)

- $\hat{y} = f(x_0, x_1, \ldots, x_n) = \sigma(\sum_{i=0}^{n} w_i x_i) = \sigma(\vec{w} \cdot \vec{x}) = \frac{1}{1+e^{-\sum_{i=0}^{n} w_i x_i}}$
- $w_i \leftarrow (w_i - \eta \frac{\partial}{\partial w_i} L_{CE}(\hat{y}, y))$
- $w_i \leftarrow (w_i - \eta(\hat{y} - y)x_i)$

Vektor form:

- $\boldsymbol{w} \leftarrow (\boldsymbol{w} - \eta(\hat{y} - y)\boldsymbol{x})$
- $\eta > 0$ is a learning rate

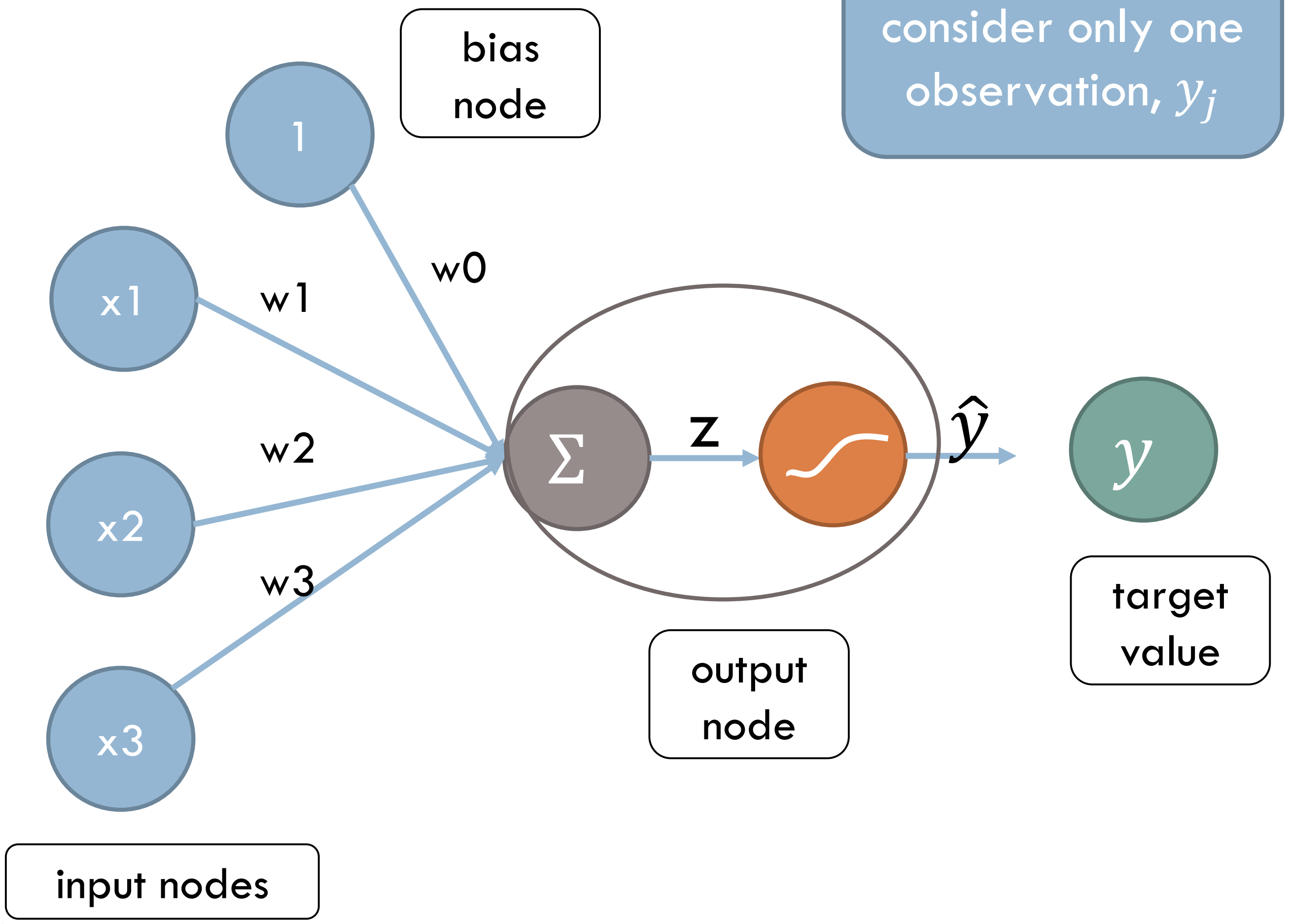
Warning

- You don't have to understand the next slide
- I have included it in case your are interested in how we find the gradient and the update
- It illustrates the use of the chain rule for (partial) derivatives.

Log.reg. the gradient

- $z = \sum_{i=0}^{m} w_i x_i = \boldsymbol{w} \cdot \boldsymbol{x}$
- $\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}$
- $L_{CE}(\vec{w}) = -\log \prod_{i=1}^{m} P(y^{(i)} | \vec{x}^{(i)}) =$
- $= -\sum_{j=1}^{n} \log \left[\hat{y}_j^{\,y_j} (1 - \hat{y}_j)^{(1-y_j)} \right]$
- $\frac{\partial}{\partial w_i} L_{CE} = \frac{\partial}{\partial \hat{y}} L_{CE} \times \frac{\partial \hat{y}}{\partial z} \times \frac{\partial z}{\partial w_i}$
- $\frac{\partial}{\partial \hat{y}} L_{CE} = -\frac{(y-\hat{y})}{\hat{y}(1-\hat{y})}$
- $\frac{\partial \hat{y}}{\partial z} = \hat{y}(1 - \hat{y})$
- $\frac{\partial z}{\partial w_i} = x_i$
- $\frac{\partial}{\partial w_i} L_{CE} = -\frac{(y-\hat{y})}{\hat{y}(1-\hat{y})} \hat{y}(1 - \hat{y}) x_i = -(y - \hat{y}) x_i$

To simplify, consider only one observation, y_j

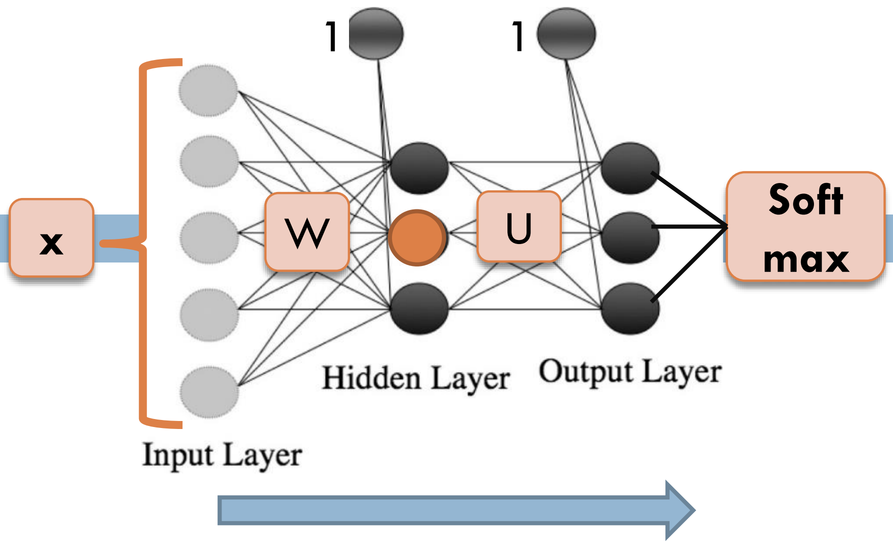


Learning

- We have considered the last layer update
- $u_{i,j} = u_{i,j} - \eta \frac{\partial}{\partial u_{i,j}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) =$

$$u_{i,j} - \eta \frac{\partial}{\partial z_i} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) \times \frac{\partial}{\partial u_{i,j}} z_i$$

The delta term at this node $\delta_{Out,i}$

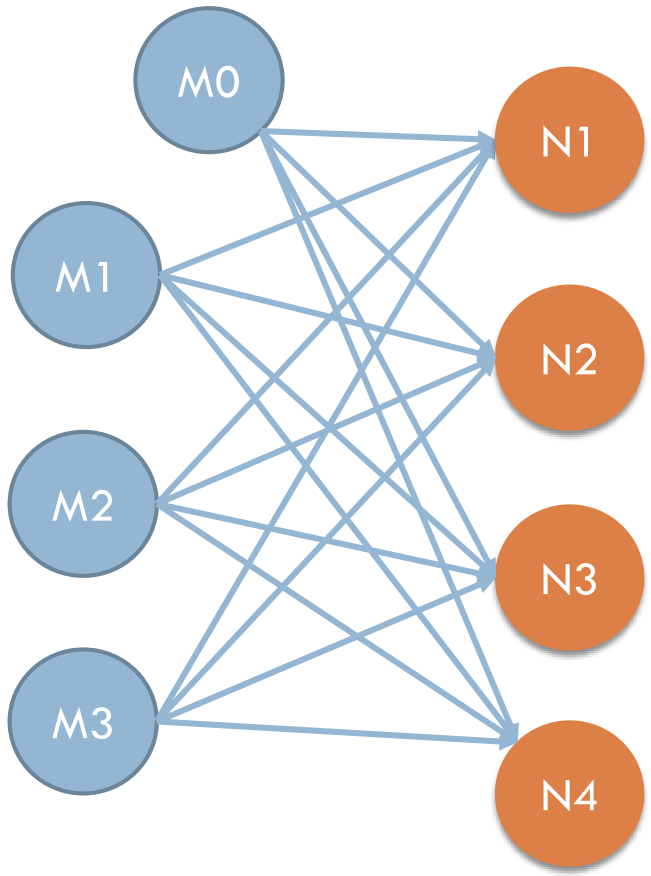


Example network:

- $\boldsymbol{h} = \sigma(W\boldsymbol{x} + b)$
- $\boldsymbol{z} = U\boldsymbol{h}$
- $\boldsymbol{y} = \text{softmax}(\boldsymbol{z})$

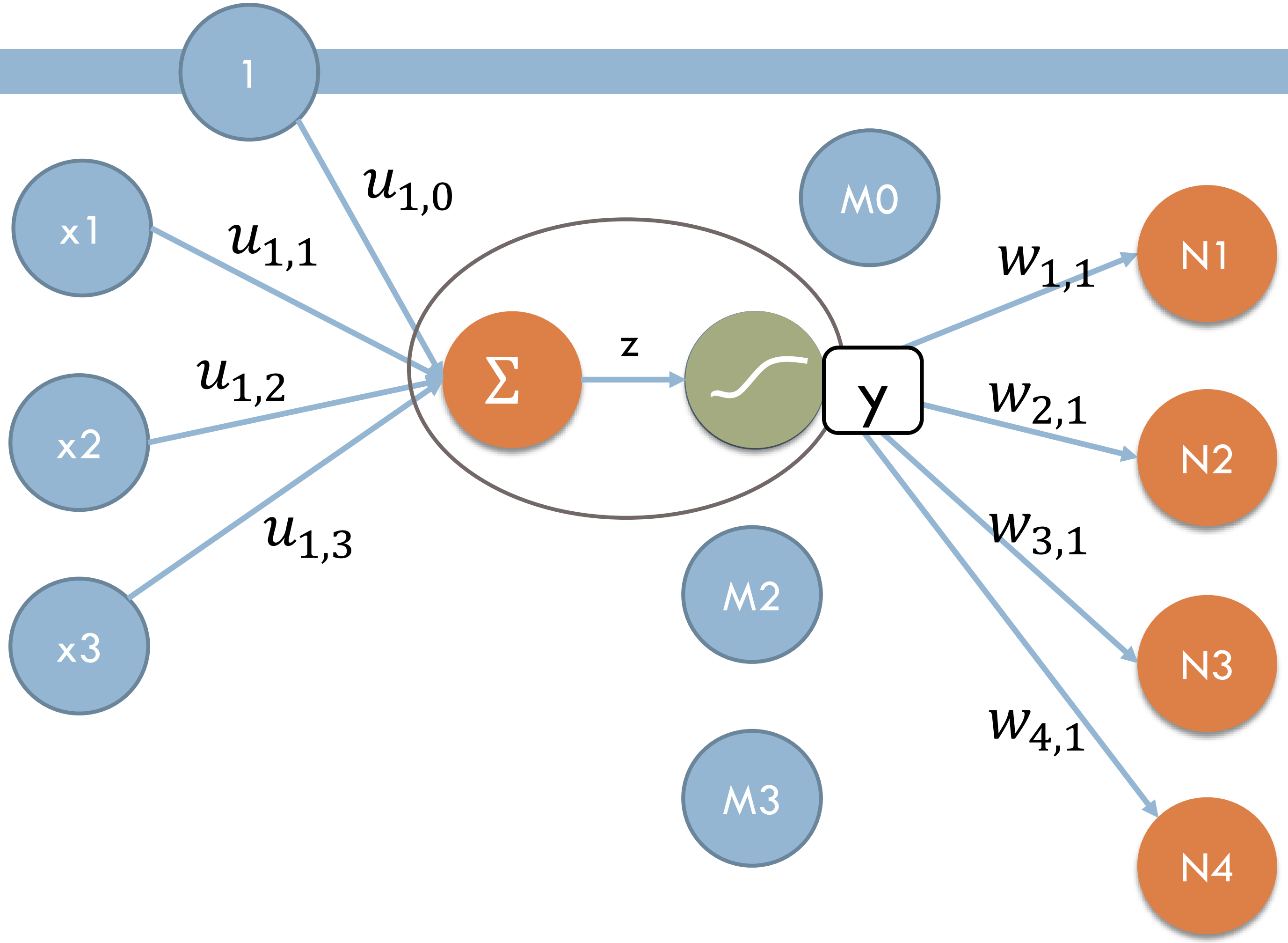
Learning in multi-layer networks

- Consider two consecutive layers:
  - Layer M, with $1 \leq i \leq m$ nodes, and a bias node M0
  - Layer N, with $1 \leq j \leq n$ nodes
  - Let $w_{j,i}$ be the weight at the edge going from M_i to N_j



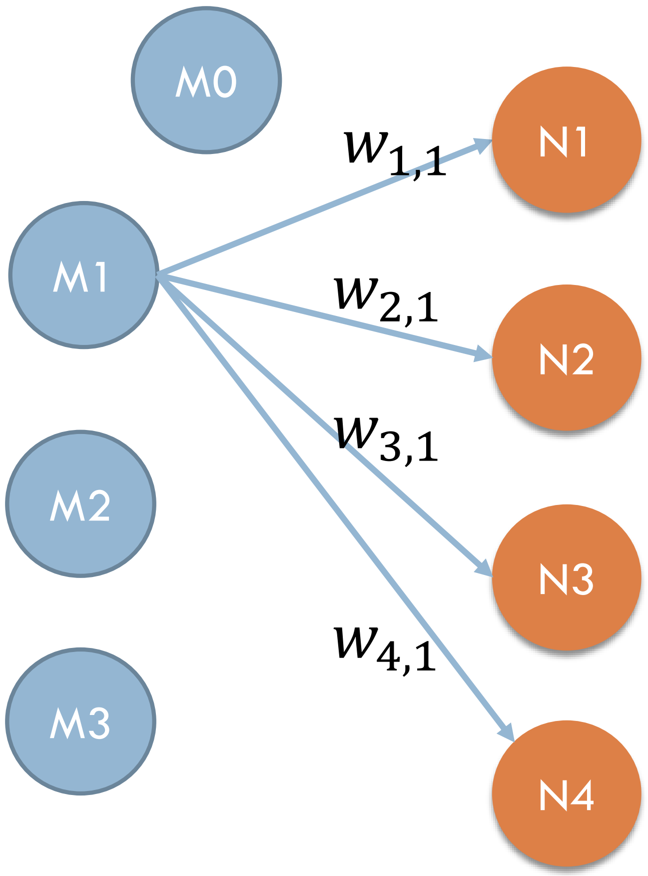
Learning in multi-layer networks

- We assume we have calculated the delta terms δ_j^N at each node N_j
- If M is a hidden layer: Calculate the error term at the nodes combining
  - A weighted sum of the error terms at layer N
  - The derivative of the activation function
  - $\delta_i^M = \left(\sum_{j=1}^{n} w_{j,i}\delta_j^N\right)\frac{d}{dz}\sigma(z)$



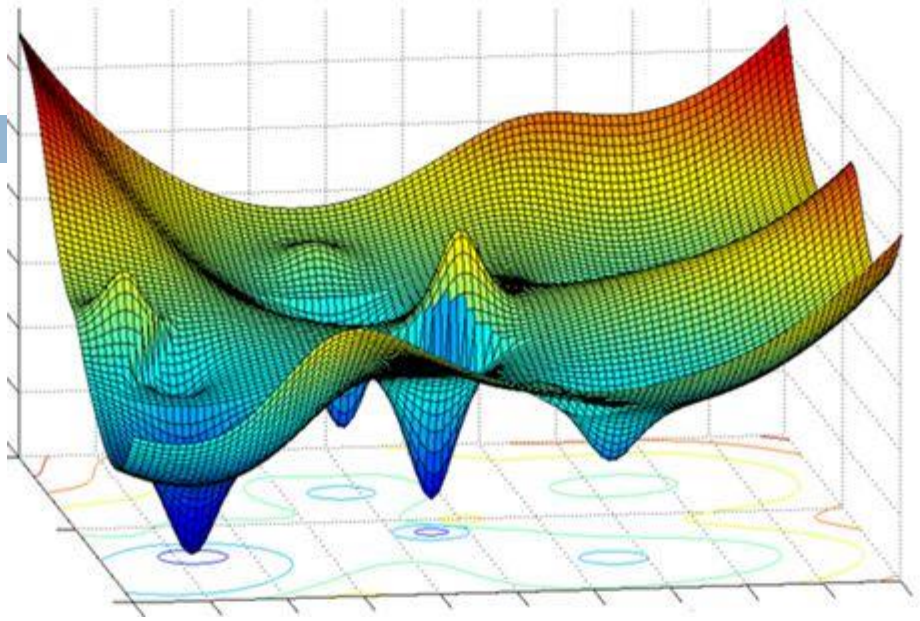
Learning in multi-layer networks

- By repeating the process, we get delta terms at all nodes in all the hidden layers.
- After we have calculated all the error terms at all the layers, we can update the weights between the layers as before:
  - $w_{j,i} = w_{j,i} - x_i \delta_j^N$
  - where x_i is the value going out of node M_i
- This is a sketch of the Backpropagation algorithm



Details on training

- First round
  - Start with random weights.
  - Train the network.
  - Test on dev data
- Repeat:
  - You get a different result
  - Why?
  - The problem is not convex
  - There exist local non-global minima



https://www.fromthegenesis.com/gradient-descent-part-2/

- Solution:
  - Run several rounds
  - Repeat
  - Report mean and st.dev.

Details on training

- There are many hyper-parameters that may be tuned
  - Example: embeddings
    - Context window size
    - Dimensions
    - "Drop-out"
- Drop-out
  - A way of regularization
  - Disregard some features during training
  - Different features for each round of training

Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language models

Dense vectors

How?

- Shorter vectors.
  - (length 50-1000)
  - ``low-dimensional'' space
- Dense (most elements are not 0)
- Intuitions:
  - Similar words should have similar vectors.
  - Words that occur in similar contexts should be similar.

Properties

- Generalize better than sparse vectors.
- Input for deep learning
  - Fewer weights (or other weights)
- Capture semantic similarities better.
- Better for sequence modelling:
  - Language models, etc.

Constructing embeddings: Idea

- Instead of counting, use a neural network to learn a LM
- Simplest form: a bigram model:
  - For a given word w_{i-1}, try to predict the next word w_i
  - i.e. try to estimate $P(w_i|\, w_{i-1})$
- Use a simple feed-forward network for this task

Model

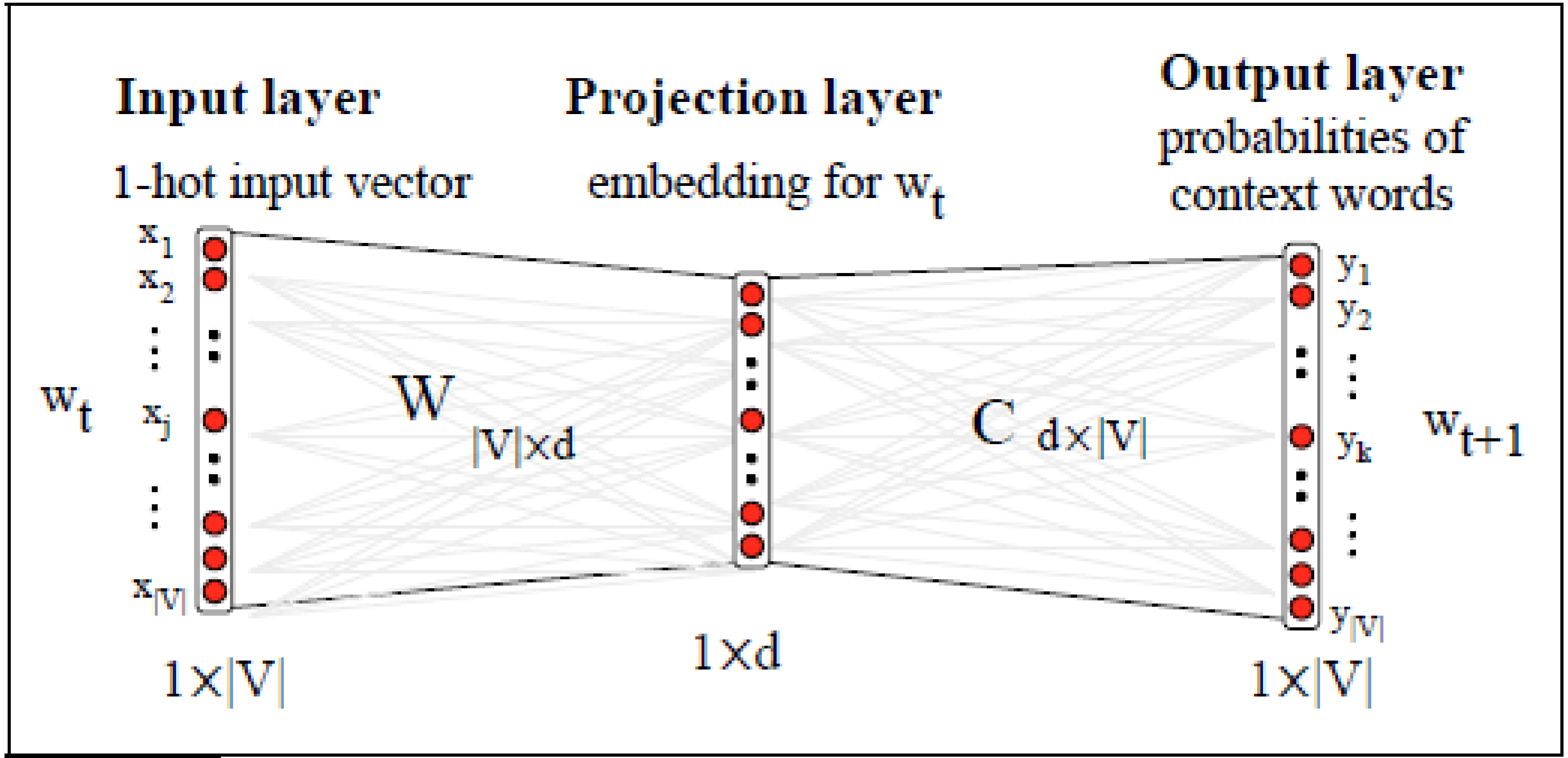
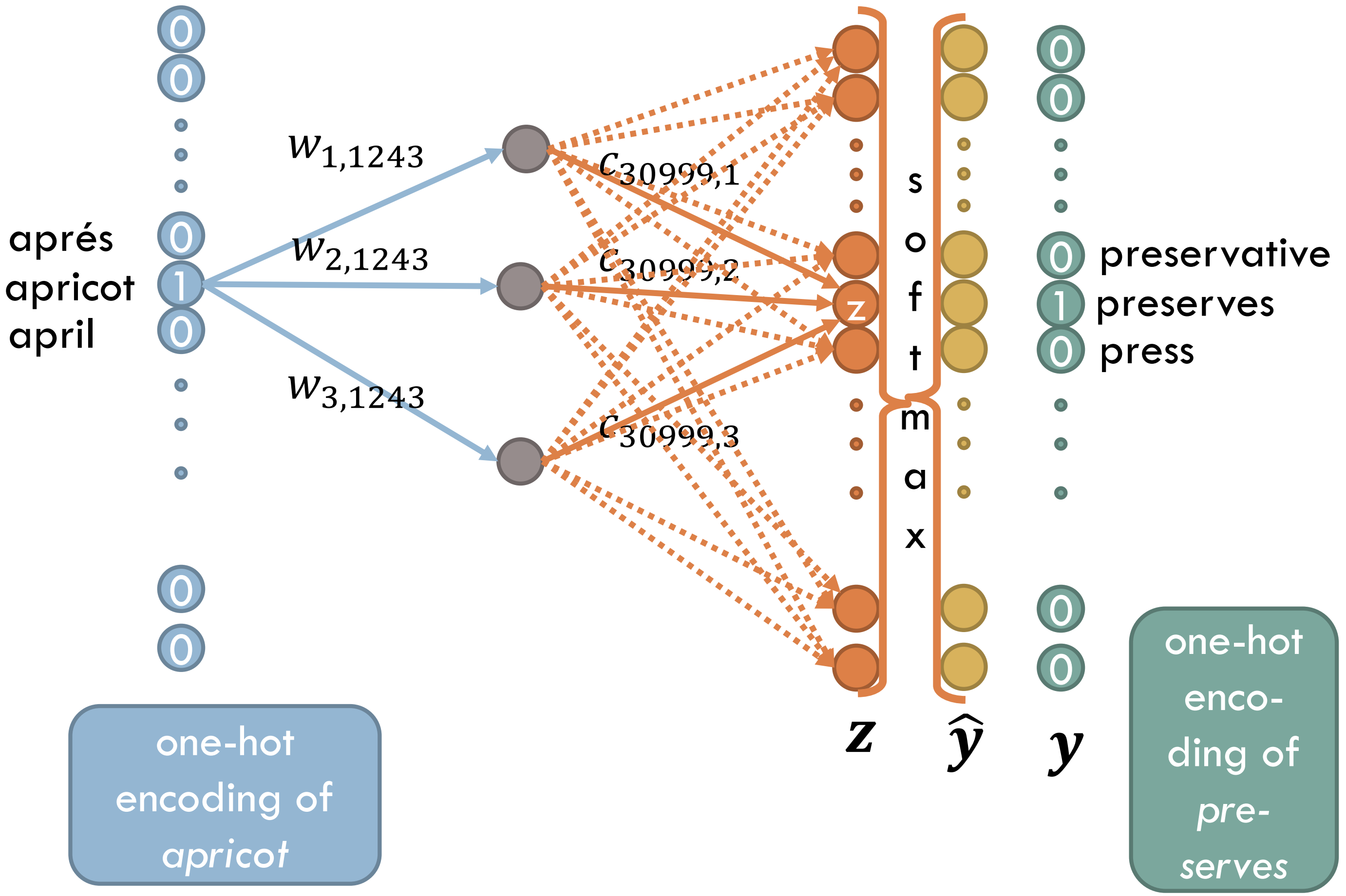


Figure 16.5 The skip-gram model viewed as a network (Mikolov et al. 2013, Mikolov et al. 2013a).

From J&M 3.ed. 2018 Ch. 16

Model

- Input and output word are represented by sparse one-hot vectors
- Dim *d* typically 50-300
- Idea for training:
  - Consider all possible next words for w' for this word
  - Use softmax to get a probability distribution of all next words



Embeddings from this

- Idea: Use the weight matrix $W_{|V|\times d}$ as embeddings, i.e.:
- Represent word j by $(w_{j,1}, w_{j,2}, \dots, w_{j,d})$ = the weights that sends this word to the hidden layer
- Why? since similar words will predict more or less the same words, they will get similar embeddings

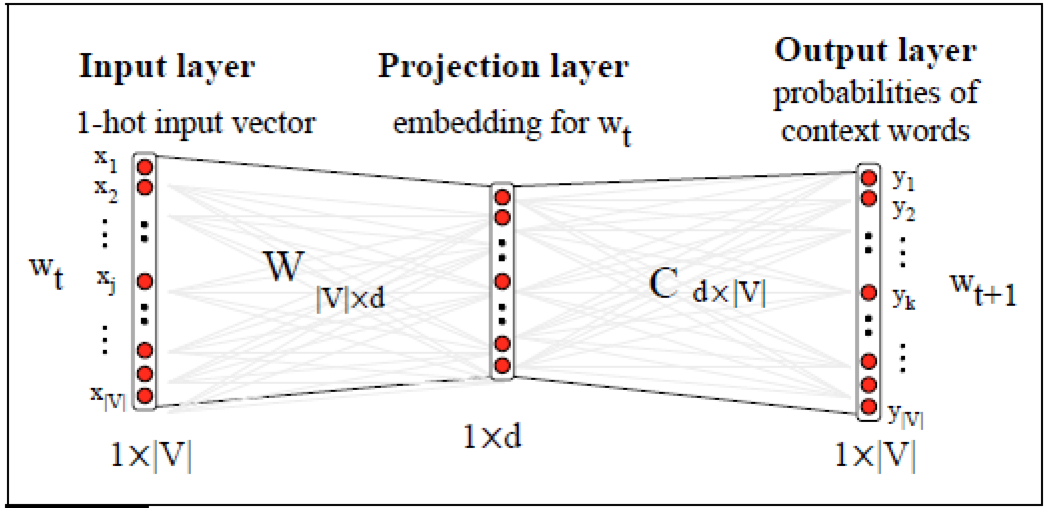
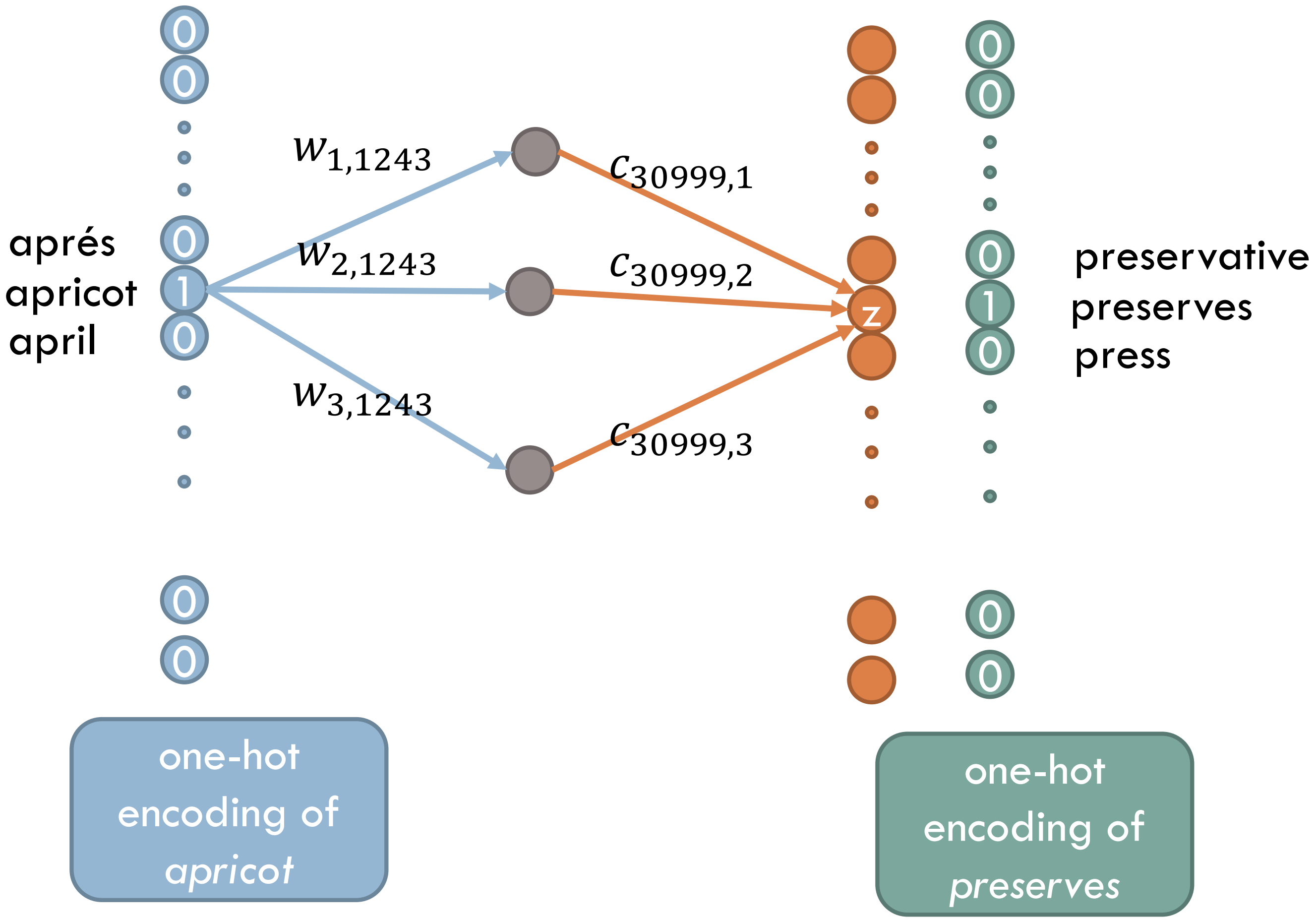


Figure 16.5 The skip-gram model viewed as a network (Mikolov et al. 2013, Mikolov et al. 2013a).

Model: zoom in

- *apricot* is word 1243
  - word-embedding:
  - $\boldsymbol{w} = (w_{1,1243}, \ldots w_{d,1243})$
- *preserves* is word 30999
  - context-embedding:
  - $\boldsymbol{c} = (c_{30999,1}, \ldots c_{30999,d})$
- $z = \boldsymbol{w} \cdot \boldsymbol{c} = \sum_{i=1}^{d} w_{i,1243} c_{i,30999}$



Observations

- Since two words that are similar are predicted by the same words, there will also be similarities between similar words in $C_{d\times|V|}$
- This will help the training of $W_{|V|\times d}$
- We could alternatively use $C_{d\times|V|}$ as the embeddings

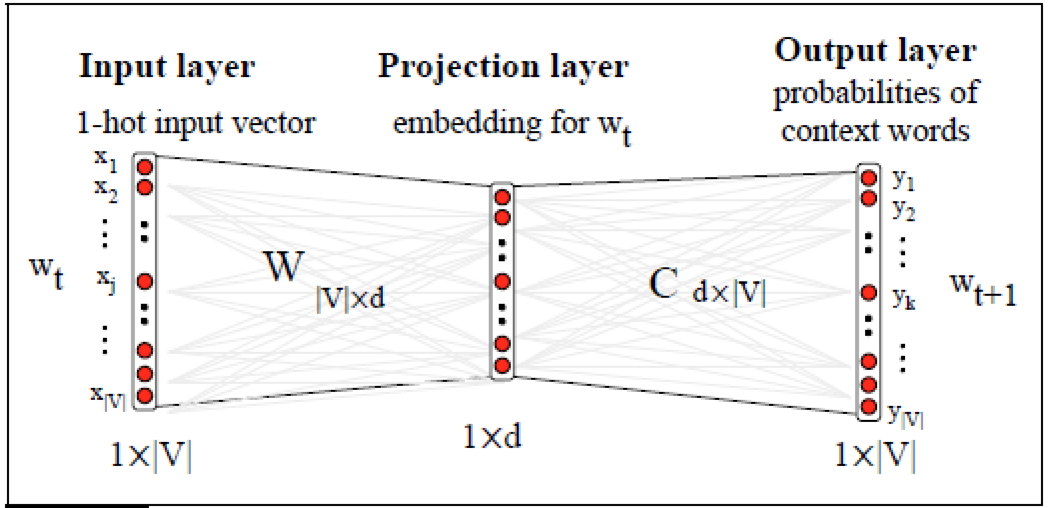
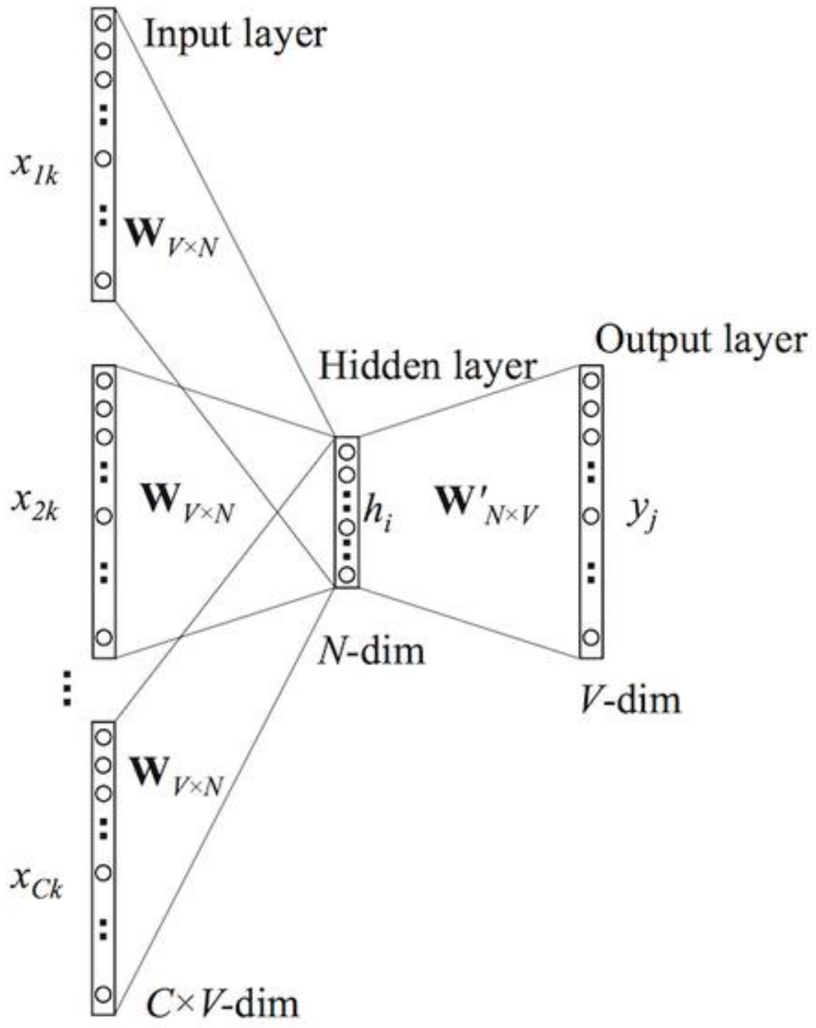


Figure 16.5 The skip-gram model viewed as a network (Mikolov et al. 2013, Mikolov et al. 2013a).

CBOW

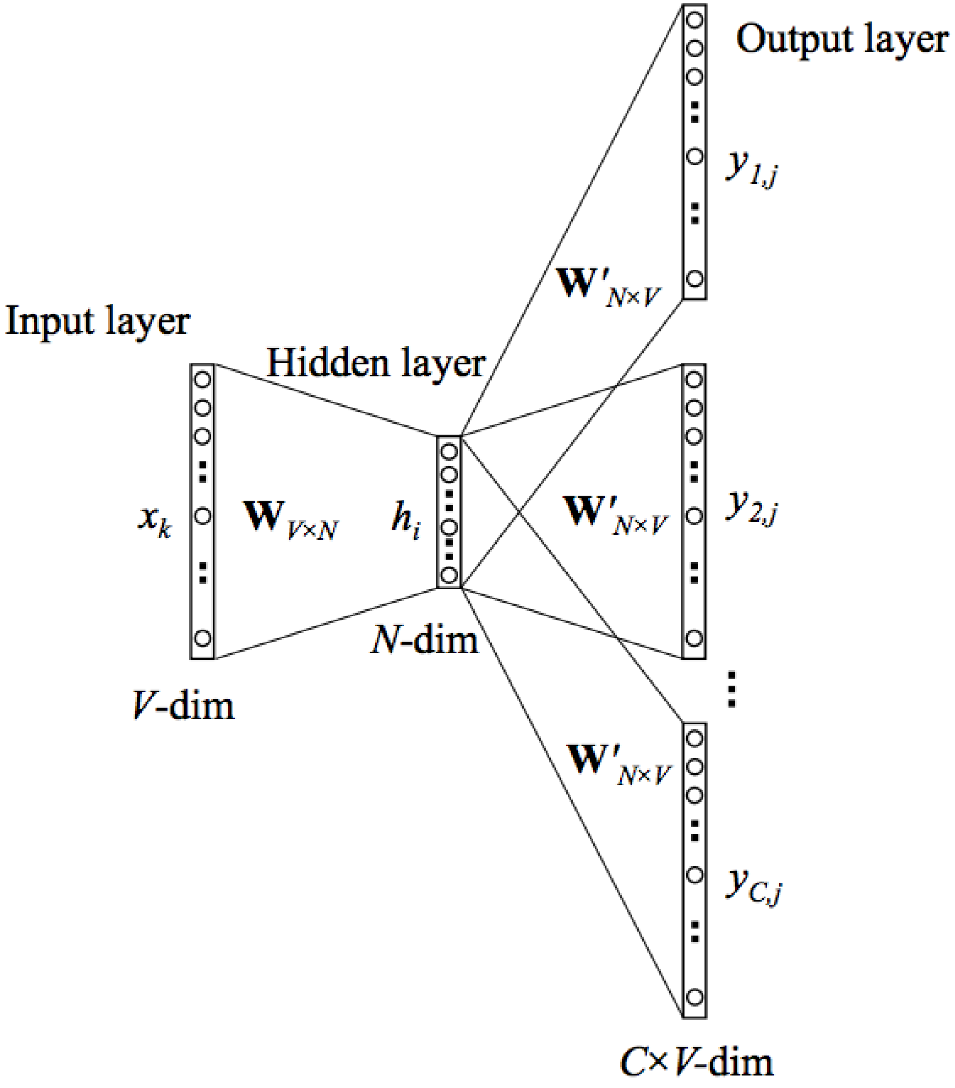
- We could generalize to predicting from a number of preceding words, e.g. 3, as indicated in the figure.
- Observe this is order-independent
- Continuous bag of words model (CBOW):
  - Predict w_t from a window $(w_{t-k}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+k})$



https://commons.wikimedia.org/wiki/File:Cbow.png

Skip-gram

- From w_t predict all the words in a window $(w_{t-k}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k})$
- Assume independence of the context words, i.e. from w_t predict each of the words w in $\{w_{t-k}, \dots, w_{t-1}, w_{t+1}, \dots, w_{t+k}\}$
- The size of the window will influence which embeddings you get



https://commons.wikimedia.org/wiki/File:Skip-gram.png

Skip-gram model

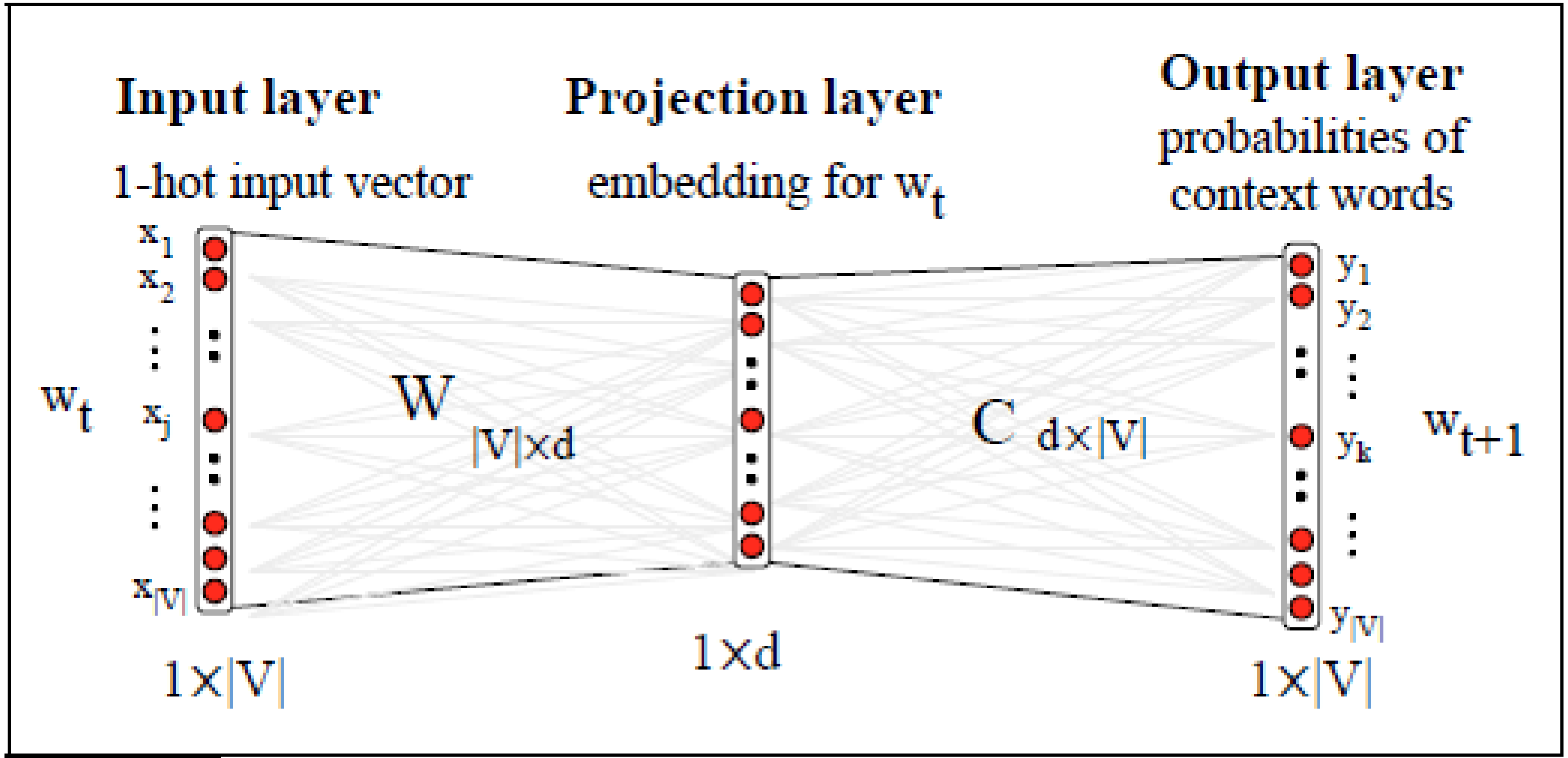
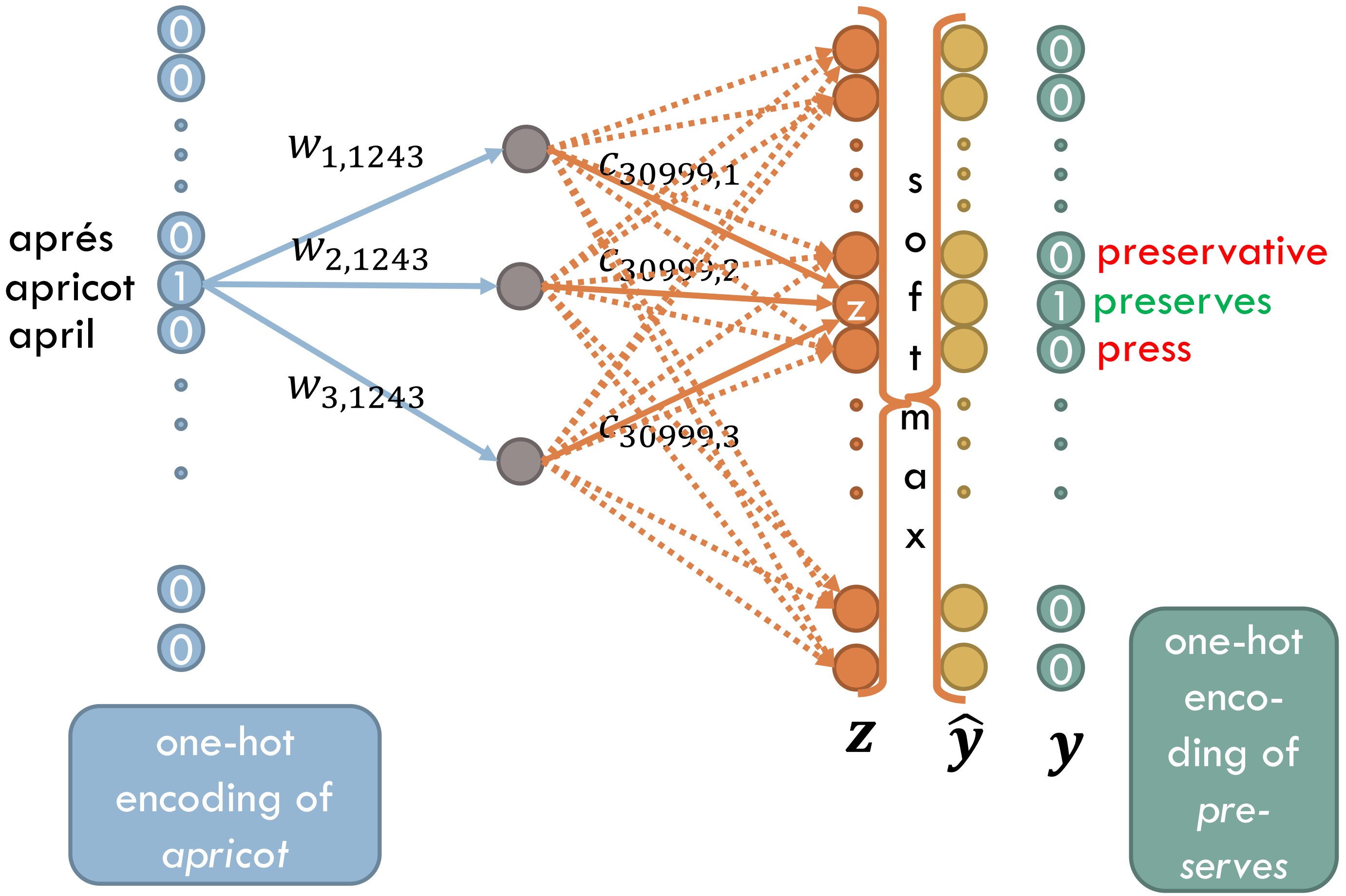


Figure 16.5 The skip-gram model viewed as a network (Mikolov et al. 2013, Mikolov et al. 2013a).

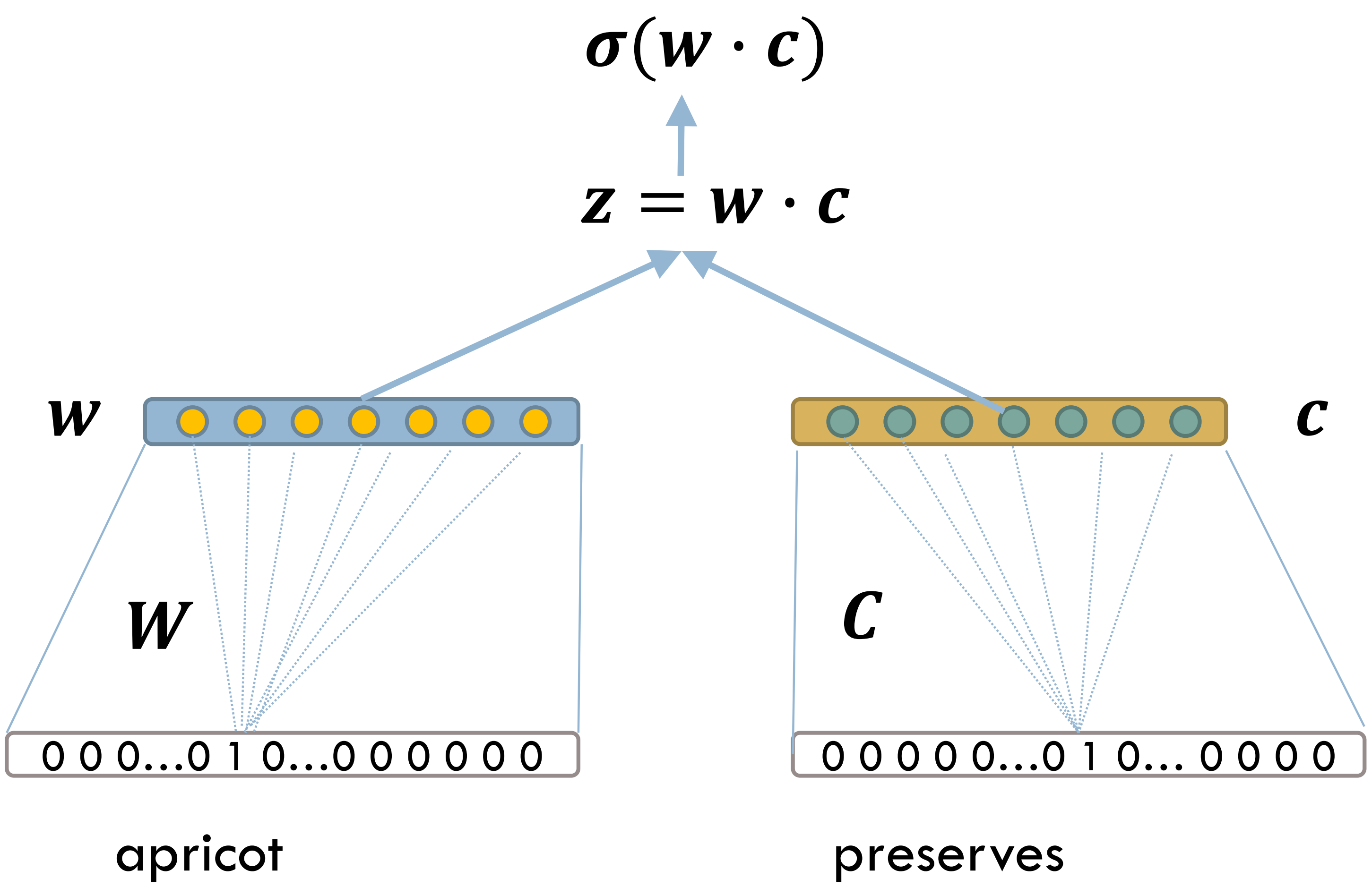
From J&M 3.ed. 2018 Ch. 16

Softmax is expensive

- The use of softmax is expensive
- For one observation, *apricot* *preserves*, one must change all the $c_{i,j}$-s to
  - increase the probability for *preserves*
  - decrease the probabilities for predicting other words
- $d \times |C|$, say $300 \times 50{,}000$



Prediction as classification



- To predict preserves from apricot, corresponds to a classification task where
  - class(apricot, preserves)=+
  - class(apricot, w)= – for all other w

Skip-gram with negative sampling

1. Treat the target word and a neighboring context word as a positive example.
2. Randomly sample other words in the lexicon to get negative samples
   - sample accordance to frequency
   - adjusted for high-frequent and low-frequent words: $P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{w'} count(w')^\alpha}$
3. Use logistic regression to train a classifier to distinguish between a positive example and the corresponding negative examples
4. Use the weights as the embeddings

Skip-Gram Training Data

- Training sentence:
- `... lemon, a tablespoon of apricot preserves or a ...`
- `c1 c2 t c3 c4`

- Training data: input/output pairs centering on *apricot*
- Asssume a +/- 2 word window

Skip-Gram Training Data

- ... lemon, a tablespoon of apricot preserves or a ...
- c1 c2 t c3 c4
- For each positive example, we'll create *k* negative examples.
  - Using *noise* words: Any random word that isn't *t*

positive examples +

t	c
apricot	tablespoon
apricot	of
apricot	preserves
apricot	or

negative examples -

t	c	t	c
apricot	aardvark	apricot	twelve
apricot	puddle	apricot	hello
apricot	where	apricot	dear
apricot	coaxial	apricot	forever

Learning

- Like Logistic Regression
- Start with randomly initialized weights for W and C
- For the training items (w, c), calculate $\hat{y} = \sigma(\boldsymbol{c} \cdot \boldsymbol{w}) = \frac{1}{1+e^{-\boldsymbol{c} \cdot \boldsymbol{w}}}$
- Compare to the gold labels using cross-entropy loss
  - The gold label is 1 if c is a context word and 0 if c is a negative example
  - This is like Logistic regression
- Use the derivative of the loss with respect to **c**: $\frac{\partial}{\partial \boldsymbol{c}} Lce$ to update **c**
- and the derivative of the loss with respect to **w** to update **w**

Update equations in SGD

- We skip the derivation, but these are the resulting update equations

$$
\begin{aligned}
\mathbf{c}_{pos}^{t+1} &= \mathbf{c}_{pos}^{t} - \eta[\sigma(\mathbf{c}_{pos}^{t} \cdot \mathbf{w}^{t}) - 1]\mathbf{w}^{t} \\
\mathbf{c}_{neg}^{t+1} &= \mathbf{c}_{neg}^{t} - \eta[\sigma(\mathbf{c}_{neg}^{t} \cdot \mathbf{w}^{t})]\mathbf{w}^{t} \\
\mathbf{w}^{t+1} &= \mathbf{w}^{t} - \eta\left[[\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}^{t}) - 1]\mathbf{c}_{pos} + \sum_{i=1}^{k}[\sigma(\mathbf{c}_{neg_i} \cdot \mathbf{w}^{t})]\mathbf{c}_{neg_i}\right]
\end{aligned}
$$

- $\hat{y} = \sigma(\boldsymbol{c} \cdot \boldsymbol{w})$
- Similar to the logistic regression, where we update weights
- Her we update both the w-s and the c-s.

Result

- We learn two separate embedding matrices W and C
- We can use W as representations for the words
  - (or combine with C in some ways)

- What have we learned:
  - If two words *w1* and *w2* occur in similar contexts
    - = with the same (or similar) context words, e.g. c,
  - then both *w1* and *w2* should have a large cosine with c,
    - hence get similar vectors.

Use of embeddings

- Embeddings are used as representations for words as input in all kinds of NLP tasks using deep learning:
  - Text classification
  - Language models
  - Named-entity recognition
  - Machine translation
  - etc.
- These embeddings are nowadays called static
- Since 2018, Transformers:
  - The embedding of each word depends on the context
  - Superior results in all tasks
- IN5550, Spring

Resources

- gensim
  - Easy-to-use tool for training own models
- Word2wec
  - https://code.google.com/archive/p/word2vec/
- https://fasttext.cc/
- https://nlp.stanford.edu/projects/glove/
- http://vectors.nlpl.eu/repository/
  - Pretrained embeddings, also for Norwegian

Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language models

Classification

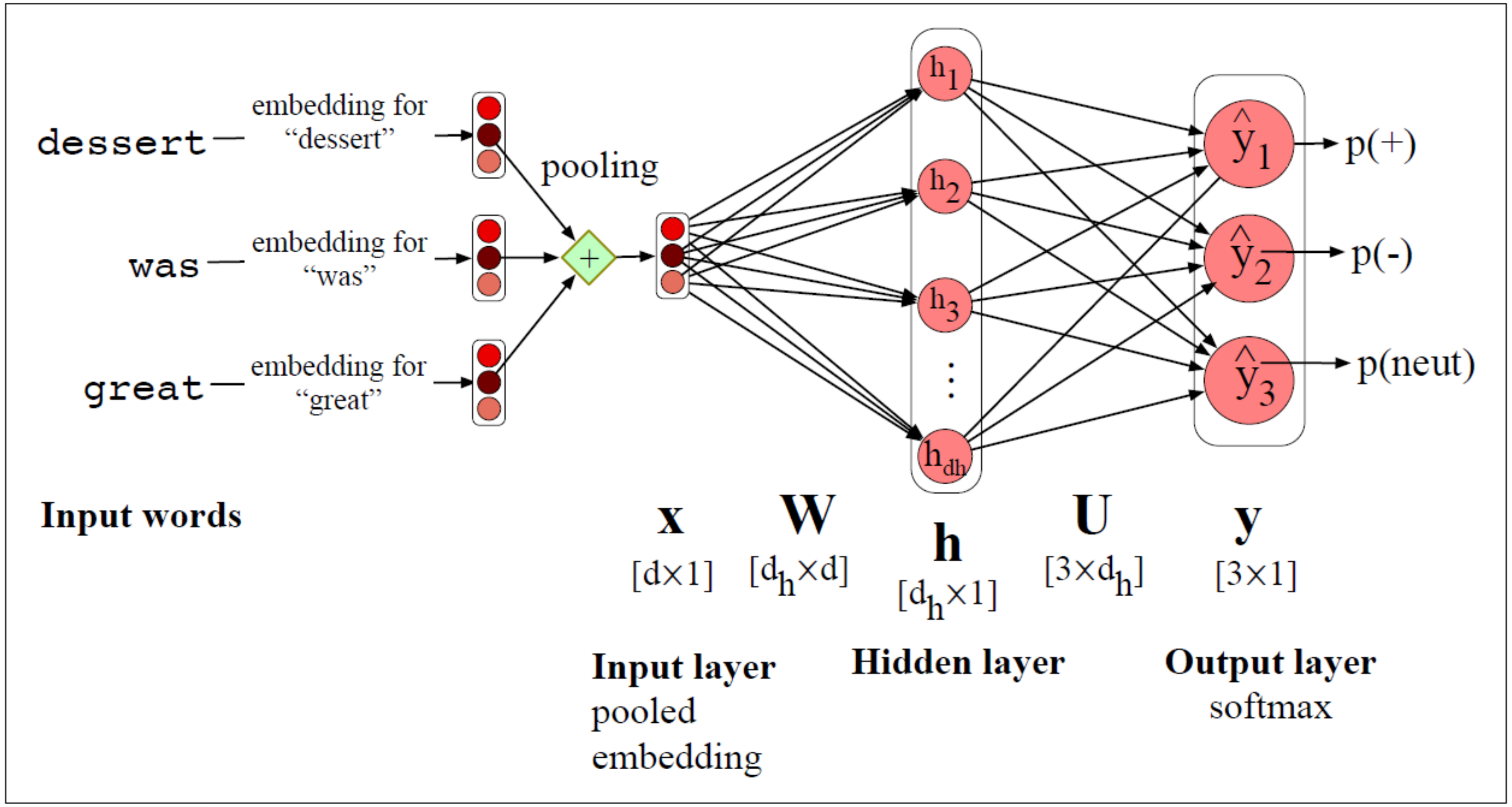


Figure 7.11 Feedforward sentiment analysis using a pooled embedding of the input words.

Today (and next week)

- Feedforward Neural Networks
- Computational graphs
- Training FNN
- Word embeddings and Word2vec
- Applying embeddings
- Neural Language Models

n-gram language models – remember?

- Goal: Ascribe probabilities to word sequences
- $P(w_1, w_2, w_3, \ldots, w_n) \approx$
- $\prod_i^n P(w_i | \, w_{i-k}, w_{i+1-k}, \ldots, w_{i-1}) = \prod_i^n P\left(w_i | \, w_{i-k}^{i-1}\right)$

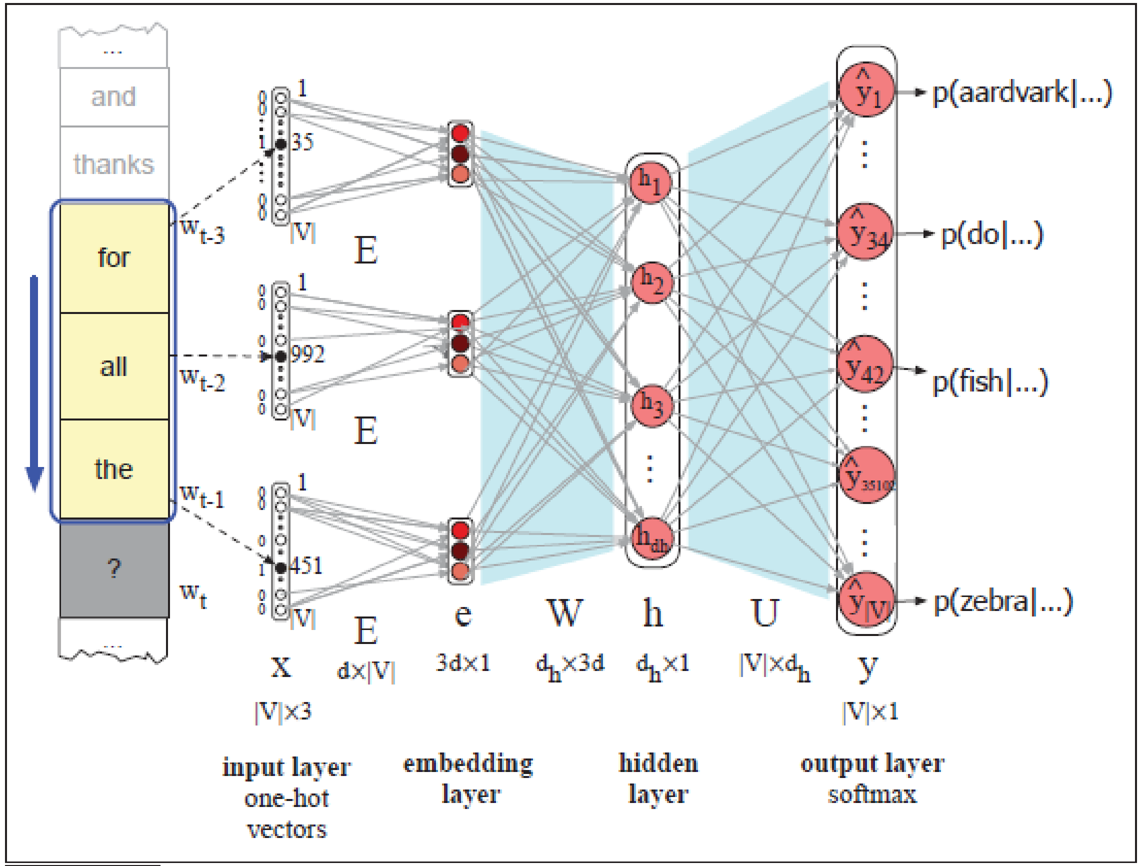
- The probabilities are estimated by counting occurrences over a corpus.

Challenges

- There might be words that is never observed during training.
- N-grams which are seen no – or only a few – times during training
- Add-*k* smoothing is not appropriate
- Possibilities:
  - Back-off
  - Interpolation
  - Kneser-Ney (best)
- Short-comings of all n-gram models
  - The smoothing is not optimal
  - The context are restricted to a limited number of preceding words

Neural Language Models

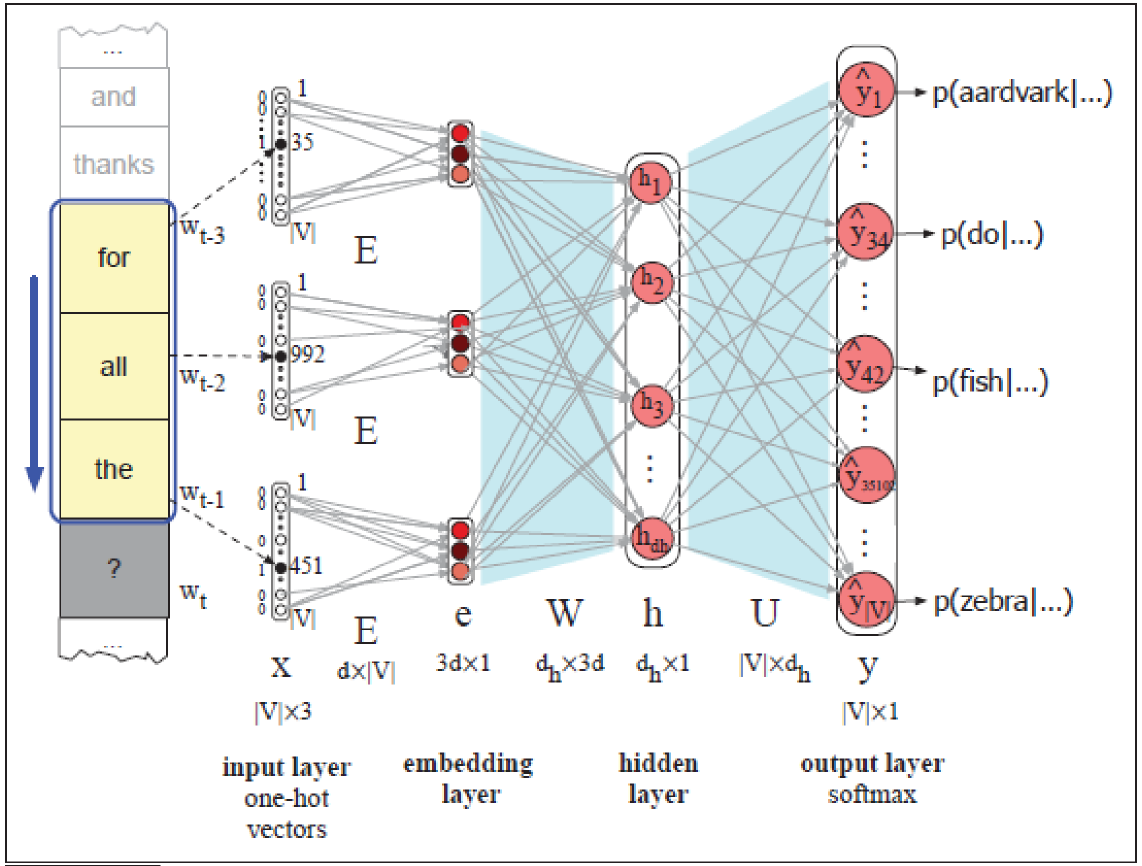
- Neural language model (*k*-gram)
  - $P\left(w_i | w_{i-k}^{i-1}\right)$
- Use embeddings for representing the w_i-s
- Use neural network for estimating $P\left(w_i | w_{i-k}^{i-1}\right)$



Neural Language Models

At each timestep t:

- Each of the words $w_j, j = t-1, t-2, t-3$
  - is represented by a one-hot-vector $\boldsymbol{x}_j$
  - which is multiplied with the same matrix E to a d-dimensional embedding $\boldsymbol{e}_j = E\boldsymbol{x}_j$
- They are concatenated to get the embedding layer **e**.
- **e** is multiplied by a weight matrix **W** and
- An activation function is applied element-wise to produce the hidden layer **h**, which is
- multiplied by another weight matrix **U**.
- Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i.



Training the language models

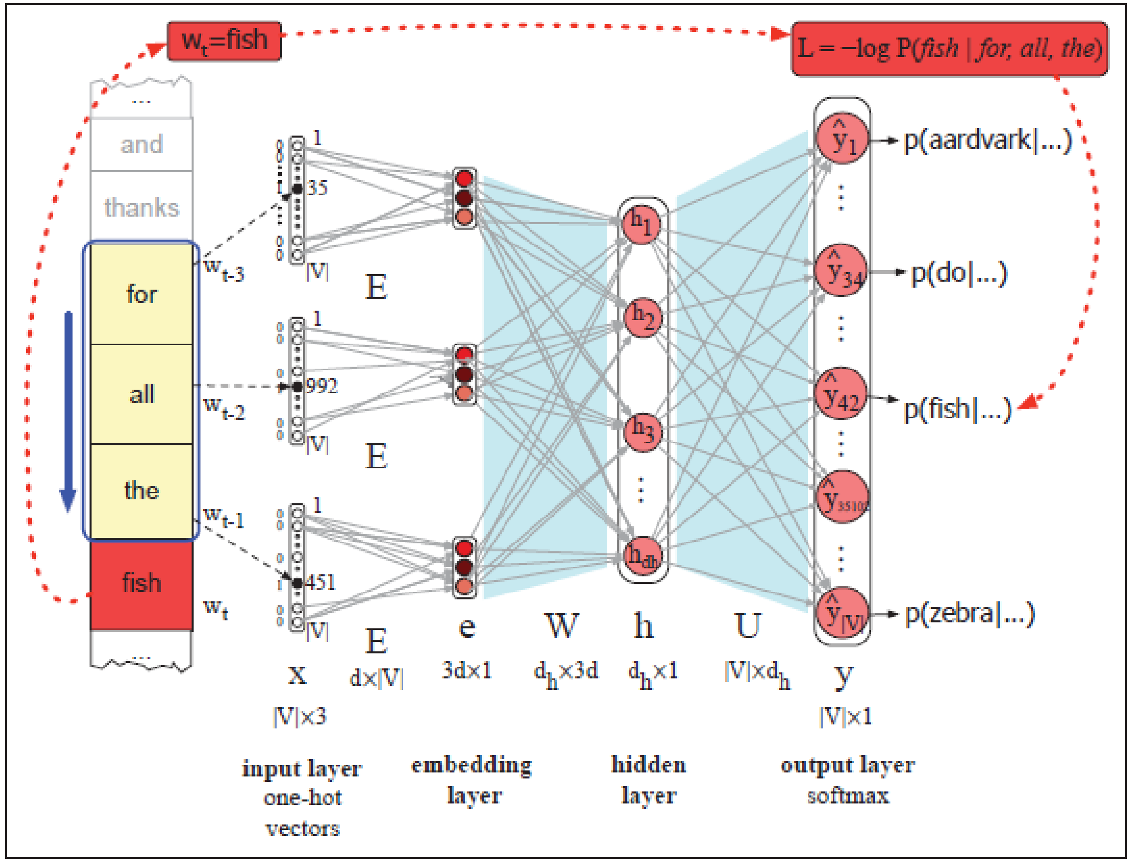


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix E is shared among the 3 context words.

Training the language models, alt. 1

- We may use pretrained embeddings
  - Trained with some method, SkipGram, CBOW, Glove, …
  - On some specific corpus
  - Can be downloaded from the web
- This means that the matrix E is fixed and that we update W and U during training

Training the embeddings

- Alternatively:
  - Start with one-hot representations of words and train the embeddings as the first layer in our models
    - (=the original model for training the embeddings)
  - Start with pre-trained embeddings, but update them during training
  - Use two set of embeddings for each word – one pretrained and one which is trained during the task.
- If the goal is a task different from language modeling, this may result in embeddings better suited for the specific tasks.

Computational graph

This picture is if we train the embeddings E
With pretrained embeddings, we look up $u_1^{[1]}$ in a table for each word