Error Backpropagation

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

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

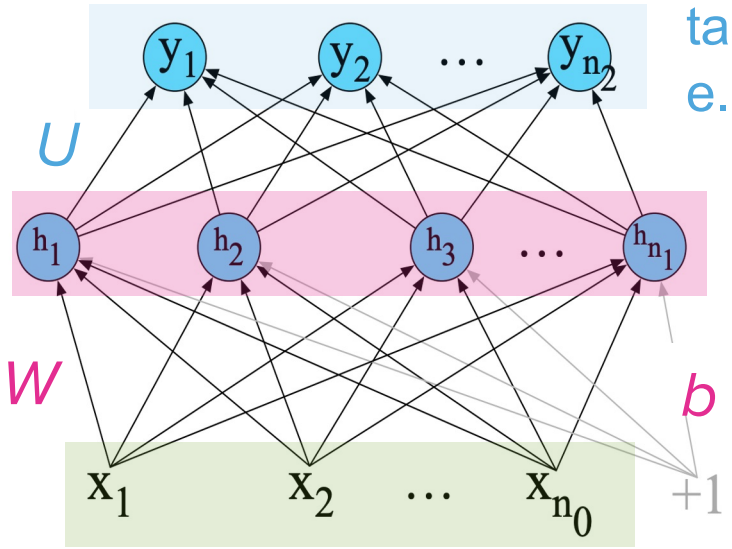
Last class

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

Announcements

- No pups today... but have a Marseille video

Recall: an FFNN computing a distribution over tags



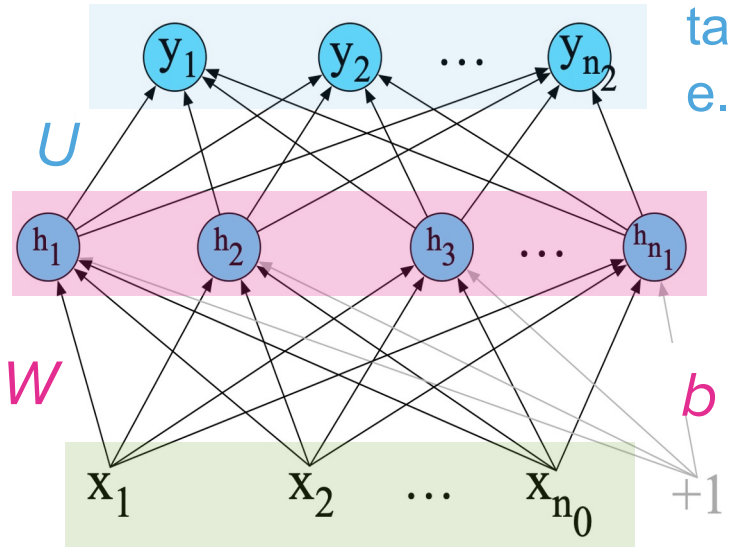
task specific outputs,
e.g., $y_1 = P(ORG|w)$

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

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

- In this picture, the tag for a word is not affecting the tag for any other word.
  (This will change in later lectures.)
- In this picture, there's only one hidden layer.
  But an FFNN can have multiple hidden layers, one feeding into the next.

Recall: an FFNN computing a distribution over tags



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

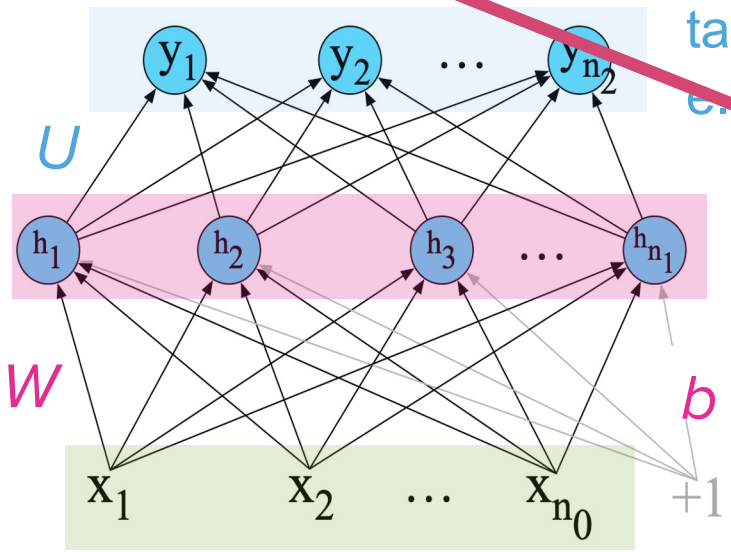
W $n_1 \times n_0$
$\mathbf{b}$ an $n_1 \times 1$ *bias* vector,
$\mathbf{x}$ $n_0 \times 1$
g an activation function that is applied elementwise to a vector.

$\mathbf{w}^{[i]}$:= 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

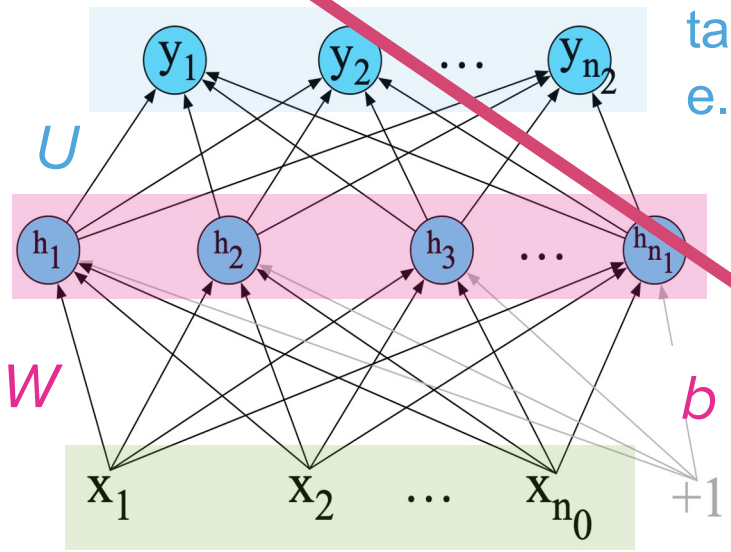
W $n_1 \times n_0$
$\mathbf{b}$ an $n_1 \times 1$ *bias* vector,
$\mathbf{x}$ $n_0 \times 1$
g an activation function that is applied elementwise to a vector.

$\mathbf{w}^{[i]}$:= 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**)

Today

- Learning the weights --- error backpropagation
- Cross-entropy loss
- FFNN's as language models

New (sub)topic: where do the weights come from?

Initial idea:

- Consider the weights to be *variables*.
- Assume existence of **labeled** training data, so we know some "true answers"
- Pick a ***loss*** function: a way to measure the error between our model's prediction and the true answers
- Choose the weight values that *minimize* the loss, using (advanced-ish) calculus!

Stochastic gradient descent: generic formulation (the eta (η) variable is the learning rate hyperparameter)

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) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow \cancel{0}$ small random values

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

But, for real models and useful loss functions, we can't just "solve for the optimal weight values".

...

There will be A LOT going on to compute *g*

repeat til done # see caption
  For each training tuple $(x^{(i)}, y^{(i)})$ (
    1. Optional (for reporting):
      Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$
      Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$
    2. $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$
    3. $\theta \leftarrow \theta - \eta\, g$
return θ

High-level algorithm

- Input: *network* with randomly initialized weights, training *examples*

repeat
    for each *e* in *examples*
        feedforward pass to compute output
        compute ***error (i.e. loss)*** at output layer
        update weights in *network* to reduce *error*
until stopping criterion is reached
return *network*

For now, assume that the **loss function** at each output node is $y - \hat{y}$

i.e. the difference between the desired output (i.e., gold standard label) and the predicted output

Note that this particular algorithm description considers and updates with respect to one example at a time.

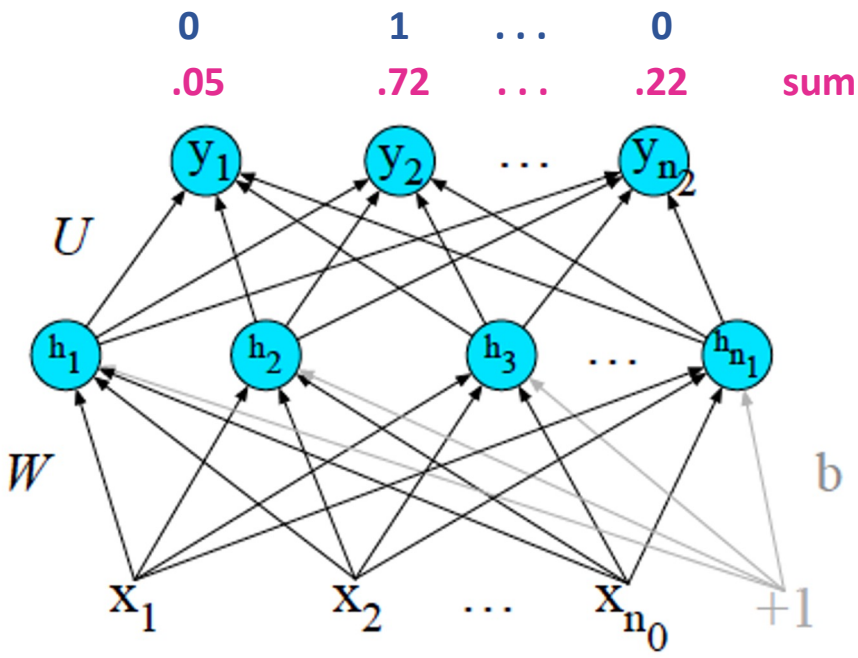
Batching of several examples at a time, where the loss of a batch is the average loss over its component items, is definitely useful.

Feedforward network

y = softmax(z)

z = Uh

$$h = \sigma(Wx + b)$$

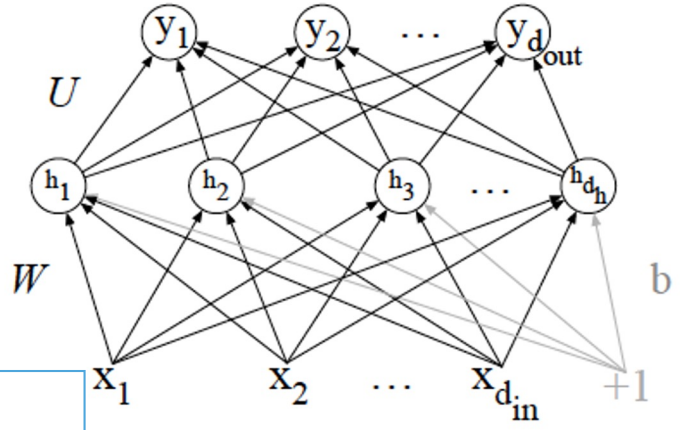


Randomly to small values near 0

Backpropagation Procedure

Initialize weights. Until performance is satisfactory*,

1. Present all training instances. For each one,
   (a) Calculate actual output. (forward pass)
   (b) Compute the weight changes. (backward pass)
      i. Calculate error at output nodes. Compute adjustment to weights from hidden layer to output layer accordingly.
      ii. Calculate error at hidden layer. Compute adjustment to weights from initial layer to hidden layer accordingly.
2. Add up weight changes and change the weights.

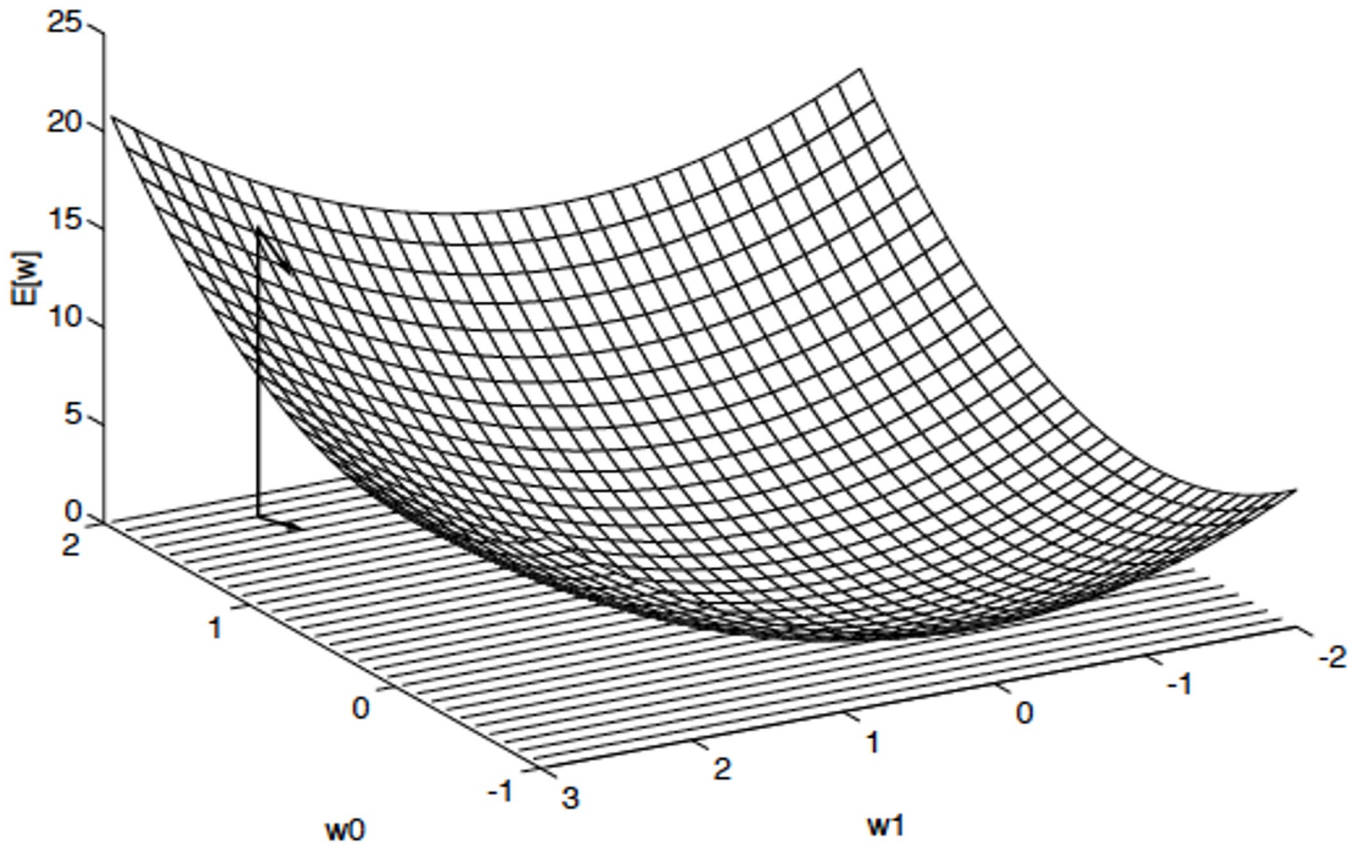


Updating the weights

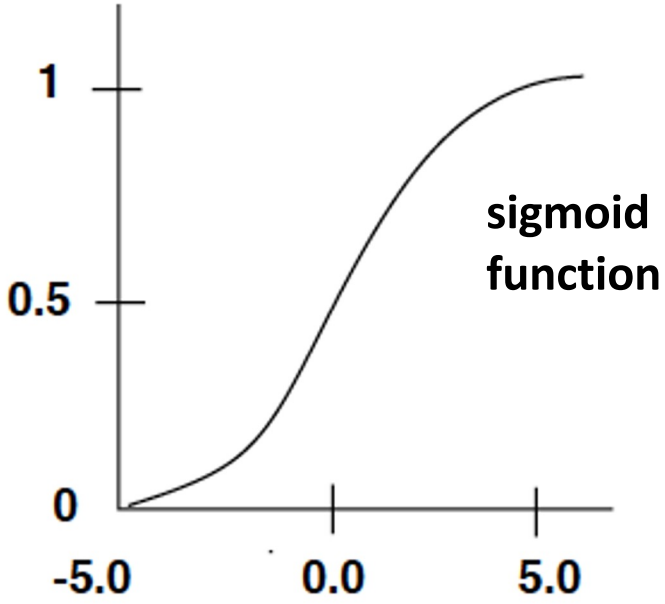
Stochastic gradient descent: weights are updated after each (randomly selected) training example

- Need an optimization algorithm for iteratively updating the weights so as to minimize the loss.
- The standard algorithm for this is **gradient descent**.
- The gradient of a function of many variables is a vector pointing in the direction of the greatest **increase** in function value.
- We want to **minimize** error, so…we will find the gradient of the loss function at the current point and move in the opposite direction.

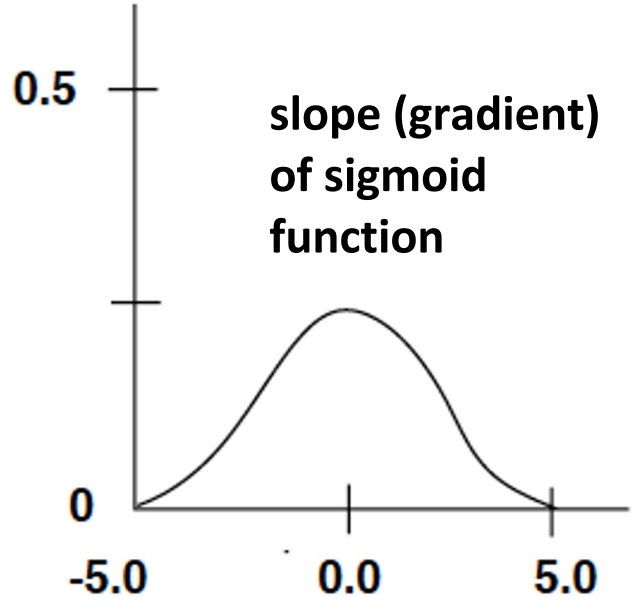
Gradient Descent Through Weight Space



Error backpropagation procedure requires a differentiable activation function.



Gradient is highest when small changes in the input produce largest change in the output.



Slope of Sigmoid Function

$$f(x) = \frac{1}{1+e^{-x}}$$

Computation of the output at a particular hidden node *j* where *x* is the input (i.e. a weighted sum)

$$\begin{aligned}
\text{Slope: } \frac{df(x)}{dx} &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) \\
&= (1+e^{-x})^{-2}\, e^{-x} \\
&= \frac{e^{-x}}{(1+e^{-x})(1+e^{-x})} \\
&= f(x)\frac{e^{-x}}{(1+e^{-x})} \\
&= f(x)(1-f(x))
\end{aligned}$$

View in terms of output at node:

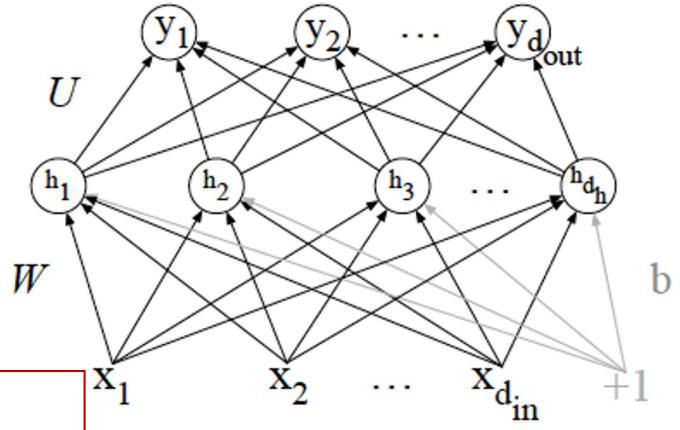
$$= o_j(1-o_j)$$

Back to

Backpropagation Procedure

Initialize weights. Until performance is satisfactory*,

1. Present all training instances. For each one,
   (a) Calculate actual output. (forward pass)
   (b) Compute the weight changes. (backward pass)
      i. Calculate error at output nodes. Compute adjustment to weights from hidden layer to output layer accordingly.
      ii. Calculate error at hidden layer. Compute adjustment to weights from initial layer to hidden layer accordingly.
2. Add up weight changes and change the weights.



Use gradient descent

Problem

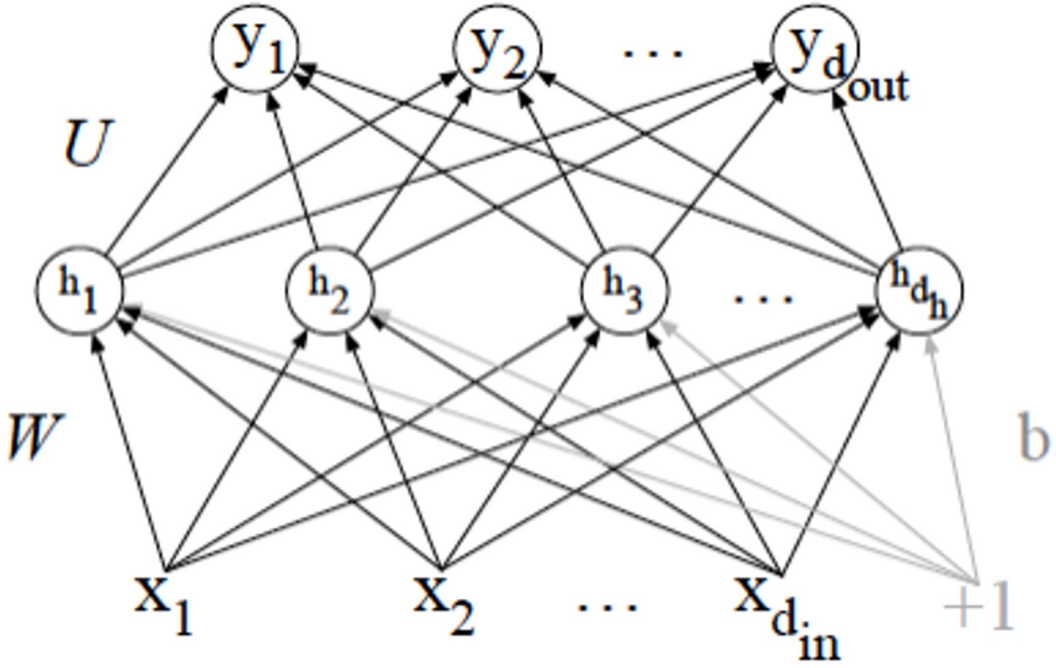
- But we only know the the loss at the output layer!!!

- For deep networks, computing the gradients for each weight is much more complex
  - we need to compute the derivative with respect to weight parameters that appear all the way back in the lowest layers of the network

Adjusting the Weights

- **Make a large change to a weight, $w_{i->j}$, if the change leads to a large reduction in the errors observed at *j*.**

- Weight change should be proportional to
  - The degree of error at *j*
  - The output at *i*
  - The gradient at *j*
- Learning rate, *r* : higher (faster) learning rate means that we should change weight more on each step.



The Backpropagation Procedure

Pick a learning rate parameter, r.

Until performance is satisfactory,

For each training instance,

- Compute the resulting output. feedforward pass
- Compute β (error) for nodes in the output layer. ?
- Compute β (error) for all other nodes. ?
- Compute weight changes for all weights using

output at i

$$\Delta w_{i \to j} = r \, o_i \times \text{gradient at j} \times \beta_j$$

Add up weight changes for all training instances, and change the weights.

Degree of error at node *j* in output layer

- Usually, we'll use *cross-entropy loss*
- For now, assume it is

  desired output – system output = $d_j - o_j$

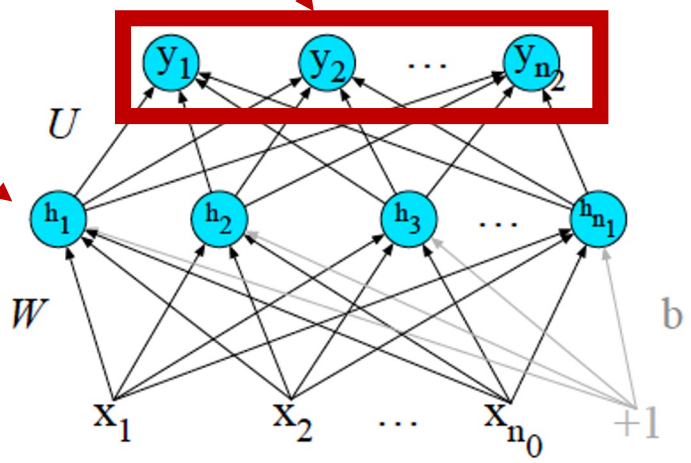
We'll cover this a little later

Computing the error at hidden layer nodes

- Consider hidden node *j*, connected to next layer *K*
  - Error at *j*:

    summation over all connecting nodes *k in K*

    the error at *k* * gradient at *k* * $w_{j->k}$



Backprop for sigmoid activation function and linear loss function

Pick a learning rate parameter, *r*.

Until performance is satisfactory,

For each training instance,

- Compute the resulting output.
- Compute $\beta = d_z - o_z$ for nodes in the output layer.
- Compute $\beta = \sum_k w_{j \to k}$ (gradient @ k) β_k for all other nodes.
- Compute weight changes for all weights using

output at i

$$\Delta w_{i \to j} = r\, o_i$$

gradient at j

$$\beta_j$$

Add up weight changes for all training instances, and change the weights.

More on backpropagation

- Will find a ***local minimum***, not necessarily **global** error minimum
- Update weights after single random example (**stochastic** gradient descent) **or** after batches of training examples
- If learning rate is too high, takes “steps” that are too large (and can overshoot the min of the loss function); too low, too slow.
- Training can take thousands of iterations --- **slow!**
- Using network **after training** is **very fast**

Cross-Entropy Loss

Cross-entropy loss

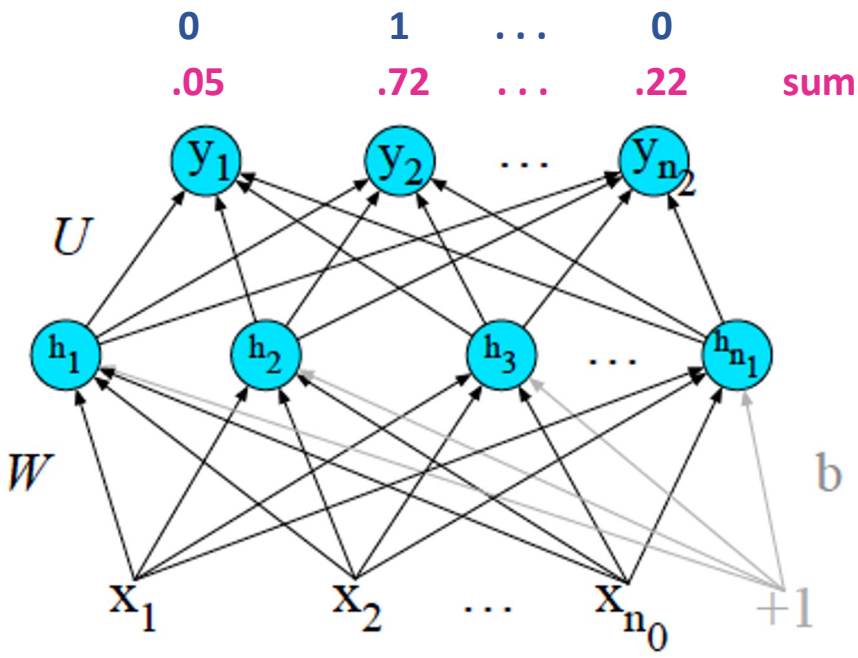
- For neural nets (and probabilistic classifiers in general), we use the ***cross-entropy loss***
  - method for computing the distance between two probability distributions

Loss: Comparing output probabilities to ground-truth non-probabilistic labels: consider both as vectors.

y = softmax(z)

z = Uh

$$h = \sigma(Wx + b)$$



Cross-entropy loss

- Let *y*, be a vector over the *C* classes representing the true output probability distribution. The cross-entropy loss is defined as

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

- For a **hard classification** task (i.e. exactly one class is correct), we can simplify as (where *i* is the correct class)

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i \xrightarrow{\text{softmax}} L_{CE}(\hat{y}, y) = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Stochastic gradient descent: generic formulation (the eta (η) variable is the learning rate hyperparameter)

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) $y^{(1)}$, $y^{(2)}$,..., $y^{(m)}$

$\theta \leftarrow \cancel{0}$ small random values

repeat til done # see caption

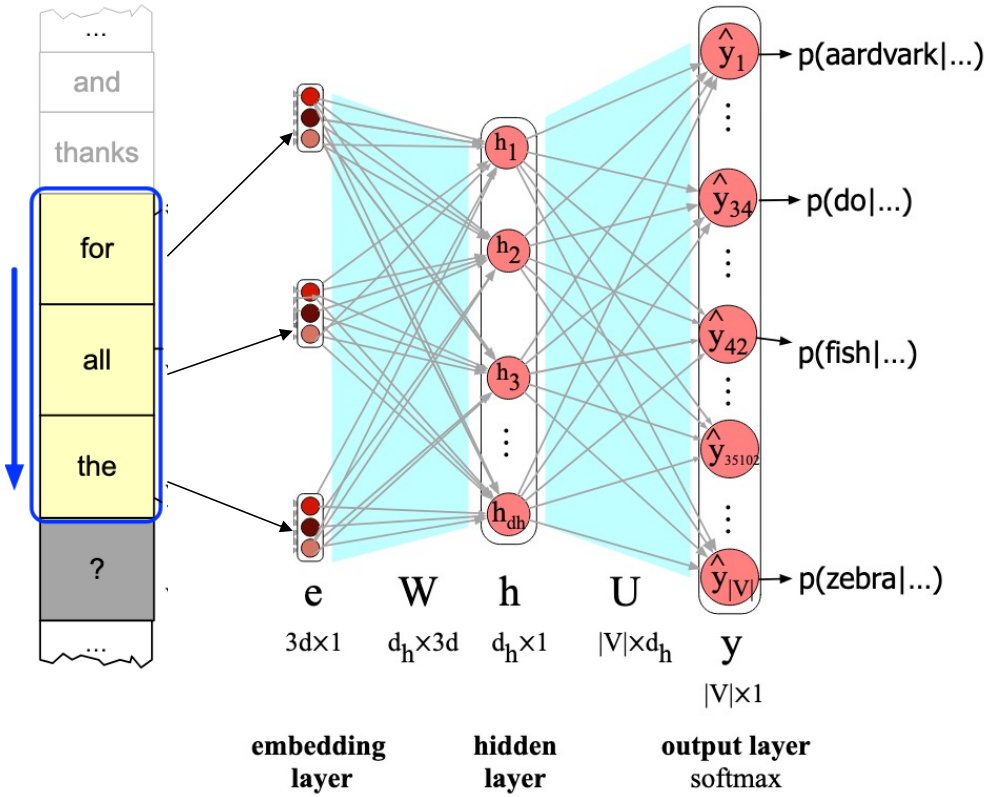
For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)

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

FFNNs for language modeling

- Can use pre-trained word embeddings



Incorporating embedding vector selection

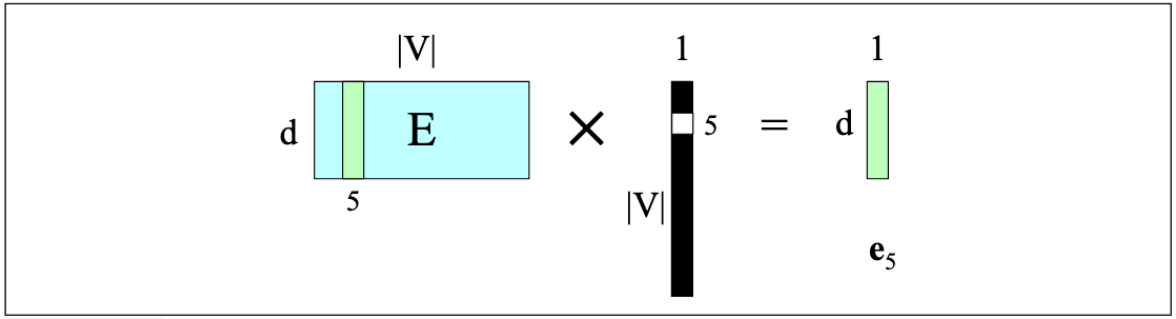
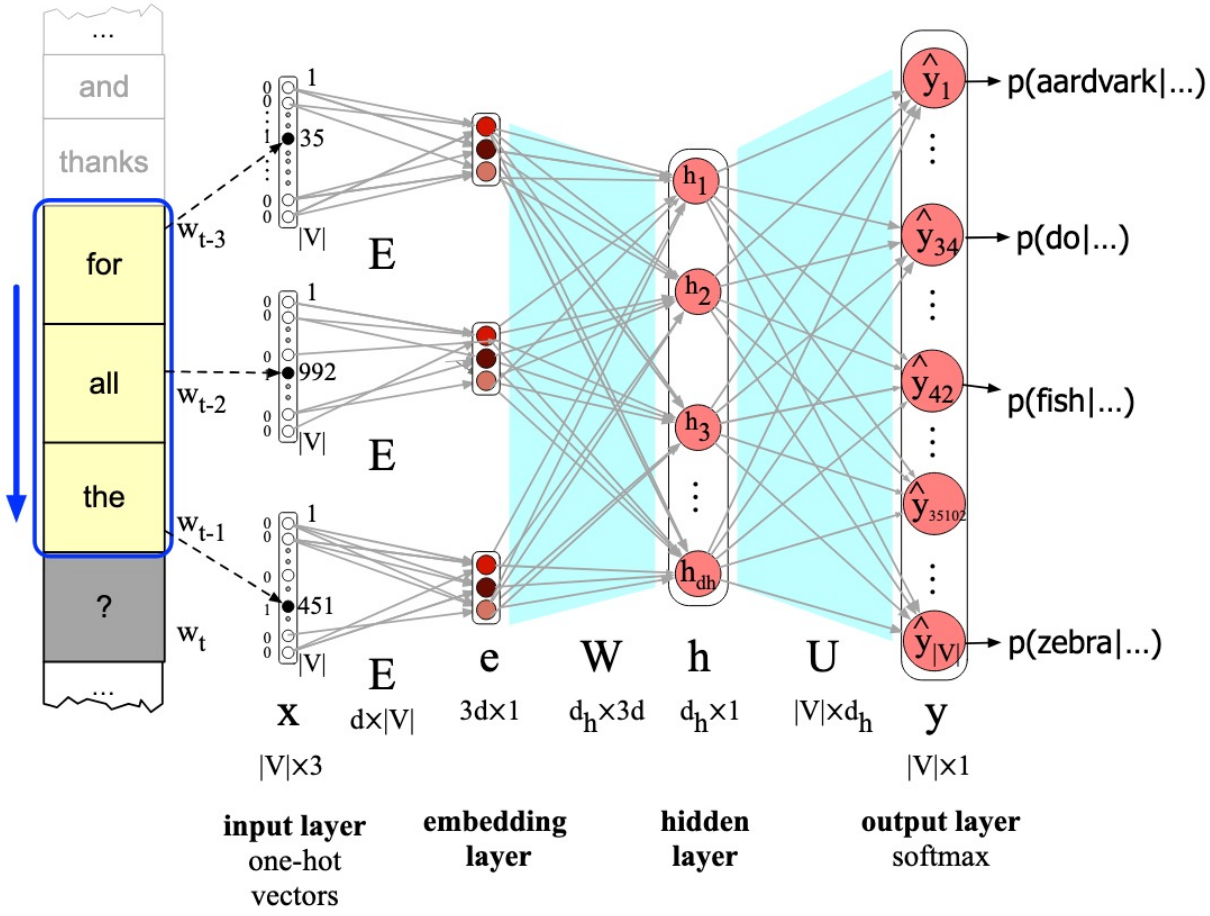


Figure 7.16 Selecting the embedding vector for word V_5 by multiplying the embedding matrix **E** with a one-hot vector with a 1 in index 5.

FFNNs for language modeling

- Forward inference at each timestep

$$\mathbf{e} = [\mathbf{E}\mathbf{x}_{t-3}; \mathbf{E}\mathbf{x}_{t-2}; \mathbf{E}\mathbf{x}_{t-1}]$$
$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{e} + \mathbf{b})$$
$$\mathbf{z} = \mathbf{U}\mathbf{h}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z})$$



Learning all the way back to embeddings

- Key: embedding matrix is shared among the context words

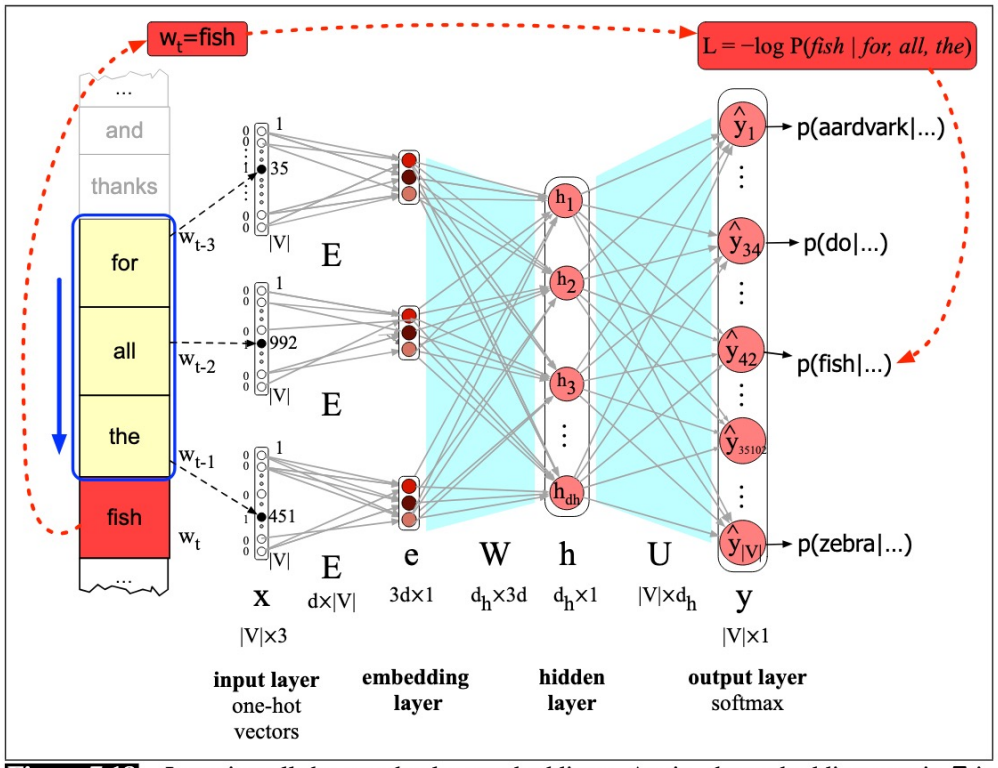


Figure 7.18 Learning all the way back to embeddings. Again, the embedding matrix **E** is shared among the 3 context words.