

COMP 3225

Natural Language Processing

NLP Tips and Tricks

Stuart E. Middleton

sem03@soton.ac.uk

University of Southampton

Copyright University of Southampton 2022.

Content for internal use at University of Southampton only.

This material was motivated by lectures from Stanford Online CS224N NLP with Deep Learning

<https://www.youtube.com/playlist?list=PLoROMvodv4rOhcuXMZkNm7j3fVwBBY42z>

Sections

- Deep Learning NLP Tips and Tricks
-

This lecture is supporting material not core examinable content.

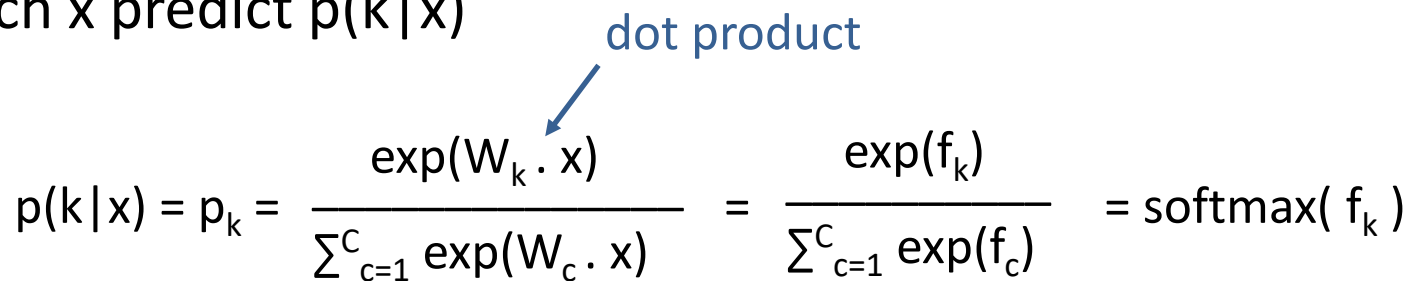
This lecture will help you understand previous examinable lecture content better. It will also help you develop your own practical NLP applications using one of the variety of toolkits available.

Overview - Deep Learning NLP Tips and Tricks

- Softmax classifier
- Cross entropy loss
- Matrix notation
- Activation functions
- Model parameters and gradients
- Backpropagation
- Matrix shapes in NLP models
- Design pattern for using word embeddings
- Tips for performance

Softmax classifier

- Softmax gives probability of different classes from a set of scores (very similar to logistic regression classifier)
- x = vector of words (length n padded if sent not long enough)
- y = vector of labels (there are C classes or labels)
 - Classes = POS tag, named entity BIO tag
- W = softmax weight matrix (dimension $C \times n$)
- For each x predict $p(k|x)$


$$p(k|x) = p_k = \frac{\exp(W_k \cdot x)}{\sum_{c=1}^C \exp(W_c \cdot x)} = \frac{\exp(f_k)}{\sum_{c=1}^C \exp(f_c)} = \text{softmax}(f_k)$$

$$f_k = \sum_{i=1}^n W_{ki} x_i$$

$$f_c = \sum_{i=1}^n W_{ci} x_i$$

- f_k = softmax score for class k
- $\text{softmax}()$ = softmax function

Cross entropy loss

- Train model to give highest probability to correct class, so we want to minimize the cross entropy loss
- p = true class prob distribution q = predicted prob distribution
- C = set of classes T = num of training examples
- Dataset of examples = $\{x_i, y_i\}_{i=1}^T$

$$\text{cross entropy} = H(p, q) = - \sum_{k=1}^K p(k) \log q(k)$$

- p is usually a one-hot vector $[0, \dots, 0, 1, 0, \dots, 0]$ with only a single entry for a true class, so $p(c) \neq 0$ only for the true class

$$H(p, q) = - \log q(k = \text{true_class})$$

- Cross entropy loss = per example average of cross entropy

$$J(\theta) = - \sum_{i=1}^T \sum_{k=1}^K y_k \log(\text{softmax}(f_k)) \quad \text{where } y_k = \text{true class prob for } k$$

- Cross entropy vector gradient per class $\Delta_k J(\theta) = \sum_{i=1}^T (p_k - y_k) x_i$

Matrix notation

- Instead of

$$f_k = f_k(x) = W_k \cdot x = \sum_{i=1}^n W_{ki} x_i$$

- We can write this using matrix notation (simpler)

$$f = Wx$$

Activation functions

- Activation function f will calculate hidden layer h values given an input x

$$h(x) = f(w^T x + b)$$

f = activation function (e.g. sigmoid = logistic function)

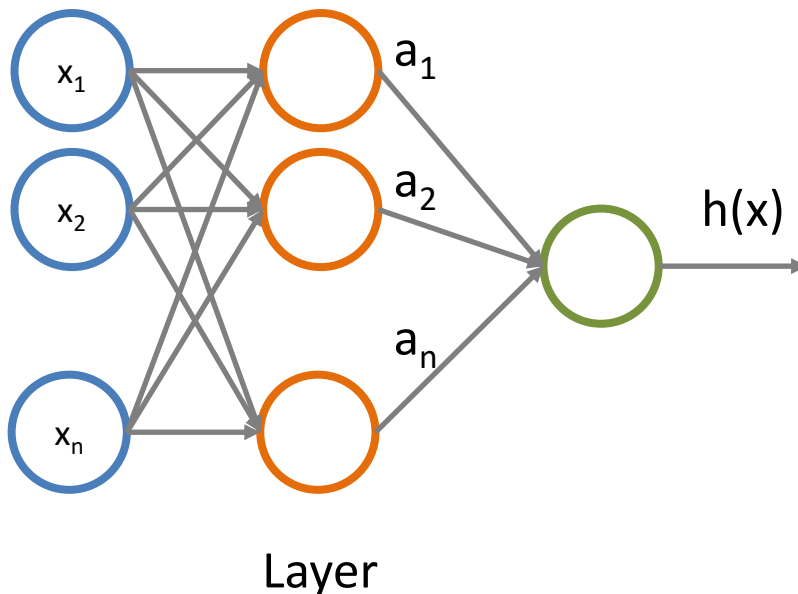
w = weights

x = inputs

b = bias term

(e.g. $b = 1$ to avoid 0's)

$$\text{sigmoid} = f(z) = \frac{1}{1 + e^{-z}}$$



output value vector

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{1n}x_n + b_1)$$

matrix notation

$$z = Wx + b = \text{layer weight matrix}$$

$$a = f(z)$$

activation func f applied element wise

$$f([z_1, z_2, z_n]) = [f(z_1), f(z_2), f(z_n)]$$

Activation functions

- Activation function f will calculate hidden layer h values given an input x

$$h(x) = f(w^T x + b)$$

f = activation function (e.g. sigmoid = logistic function)

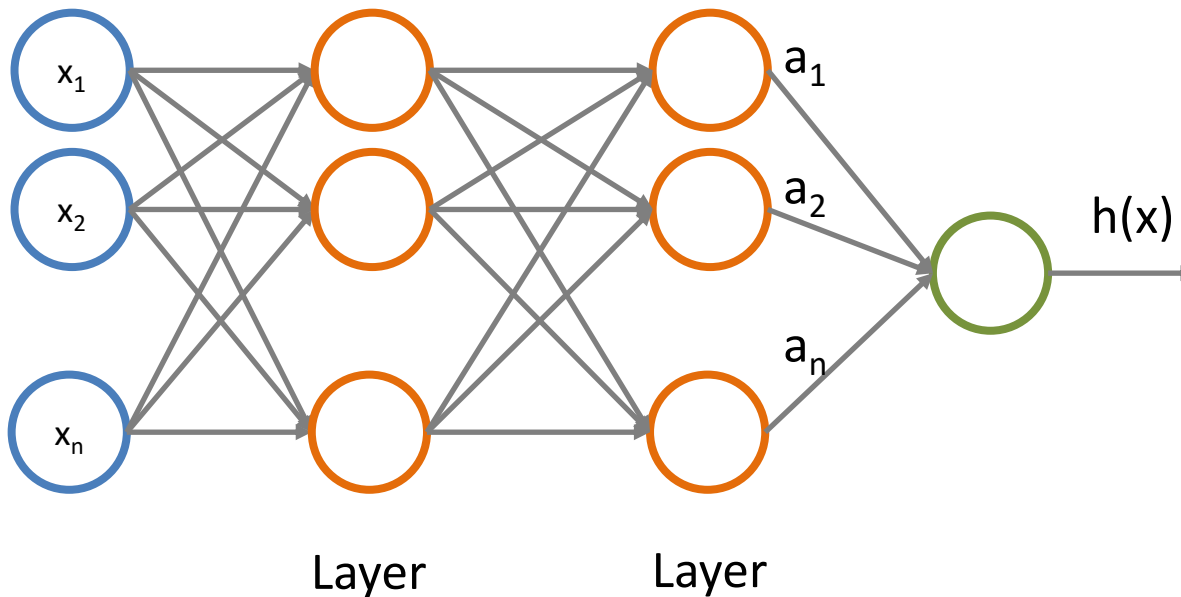
w = weights

x = inputs

b = bias term

(e.g. $b = 1$ to avoid 0's)

$$\text{sigmoid} = f(z) = \frac{1}{1 + e^{-z}}$$



Model parameters and gradients

- Model parameters are a $C \times n$ matrix of weights

$$\theta = \begin{bmatrix} W_{11} & \dots & W_{1C} \\ \vdots & & \\ \vdots & & \\ W_n & \dots & W_{nC} \end{bmatrix} = \text{parameter matrix for model}$$

- Gradient of loss with respect to parameters

$$\nabla J(\theta) = \begin{bmatrix} \nabla W_1 & \dots & \nabla W_{1C} \\ \vdots & & \\ \vdots & & \\ \nabla W_n & \dots & \nabla W_{nC} \end{bmatrix} = \text{gradient of cross entropy loss for parameters}$$

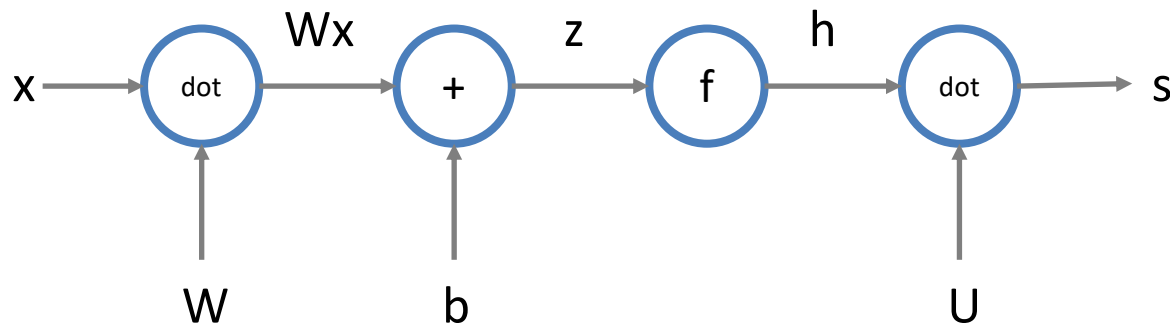
Backpropagation

- Backpropagation is a way to compute gradients efficiently using matrix computation
- Gradient ∇ = derivative = set of partial derivatives (chain rule)
- Chain rule $\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$
- $z = Wx$ $\frac{dh}{dx} = \frac{dh}{dz} \times \frac{dz}{dx}$
- $h = f(z)$
- Shape convention >> Derivative shape = Variable shape
- PyTorch / Tensorflow have special code for handling gradients efficiently in sparse matrices and sort/optimize node calc

Backpropagation

- Forward propagation
 - Compute hidden layer values using activation function
- Back propagation
 - Calc partial derivatives at each step, and pass gradients back through graph
 - Compute local gradients (calculus) + apply chain rule
 - [downstream gradient] = [upstream gradient] x [local gradient]
 - multiple inputs >> multiple local gradients
- Computation graphs representing network propagation

$$h(x) = f(z) \quad z = Wx + b \quad s = Uh$$

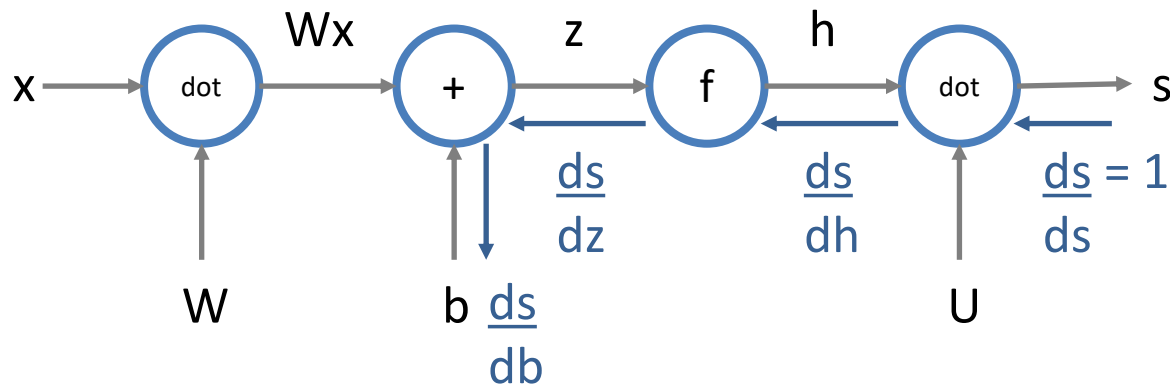


forward propagation

Backpropagation

- Forward propagation
 - Compute hidden layer values using activation function
- Back propagation
 - Calc partial derivatives at each step, and pass gradients back through graph
 - Compute local gradients (calculus) + apply chain rule
 - [downstream gradient] = [upstream gradient] x [local gradient]
 - multiple inputs >> multiple local gradients
- Computation graphs representing network propagation

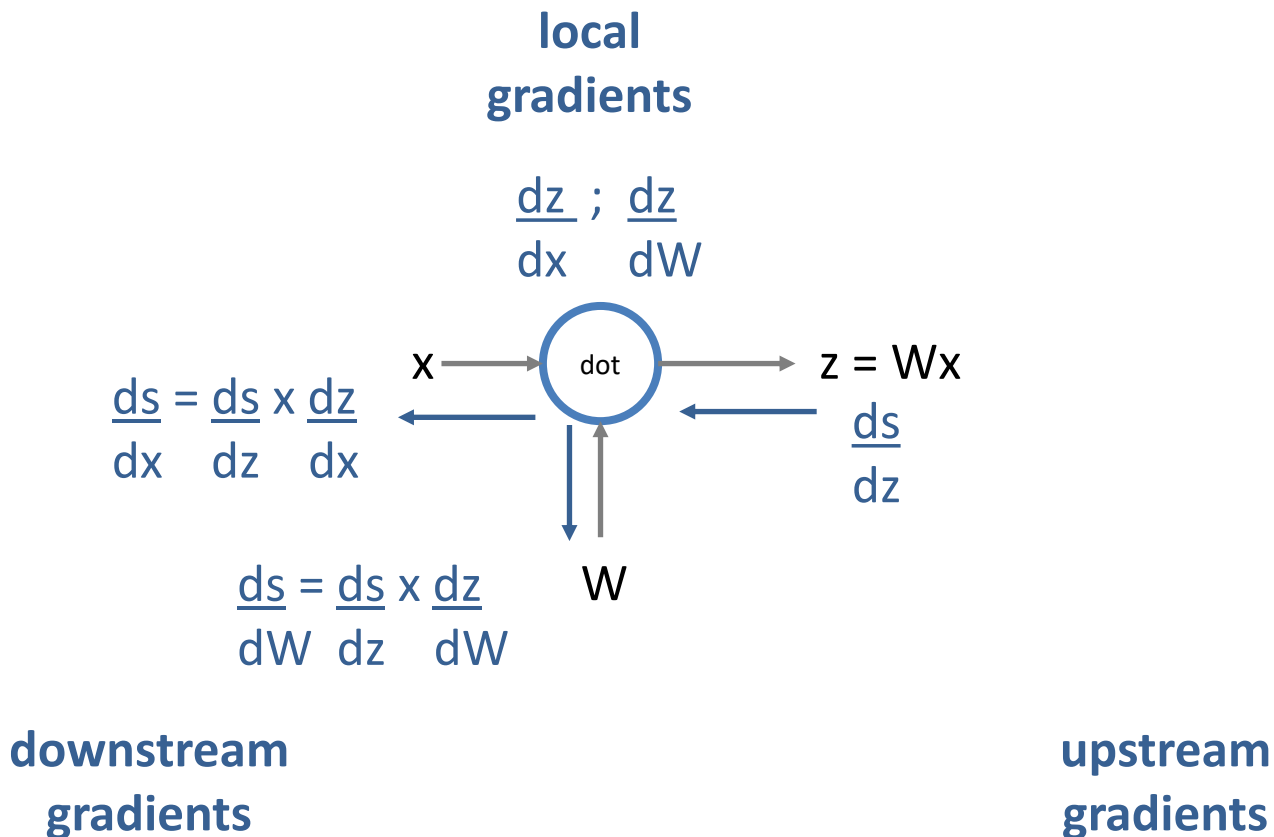
$$h(x) = f(z) \quad z = Wx + b \quad s = Uh$$



back propagation

Backpropagation

- Use chain rule as move downstream
 - [downstream gradient] = [upstream gradient] x [local gradient]
 - multiple inputs >> multiple local gradients

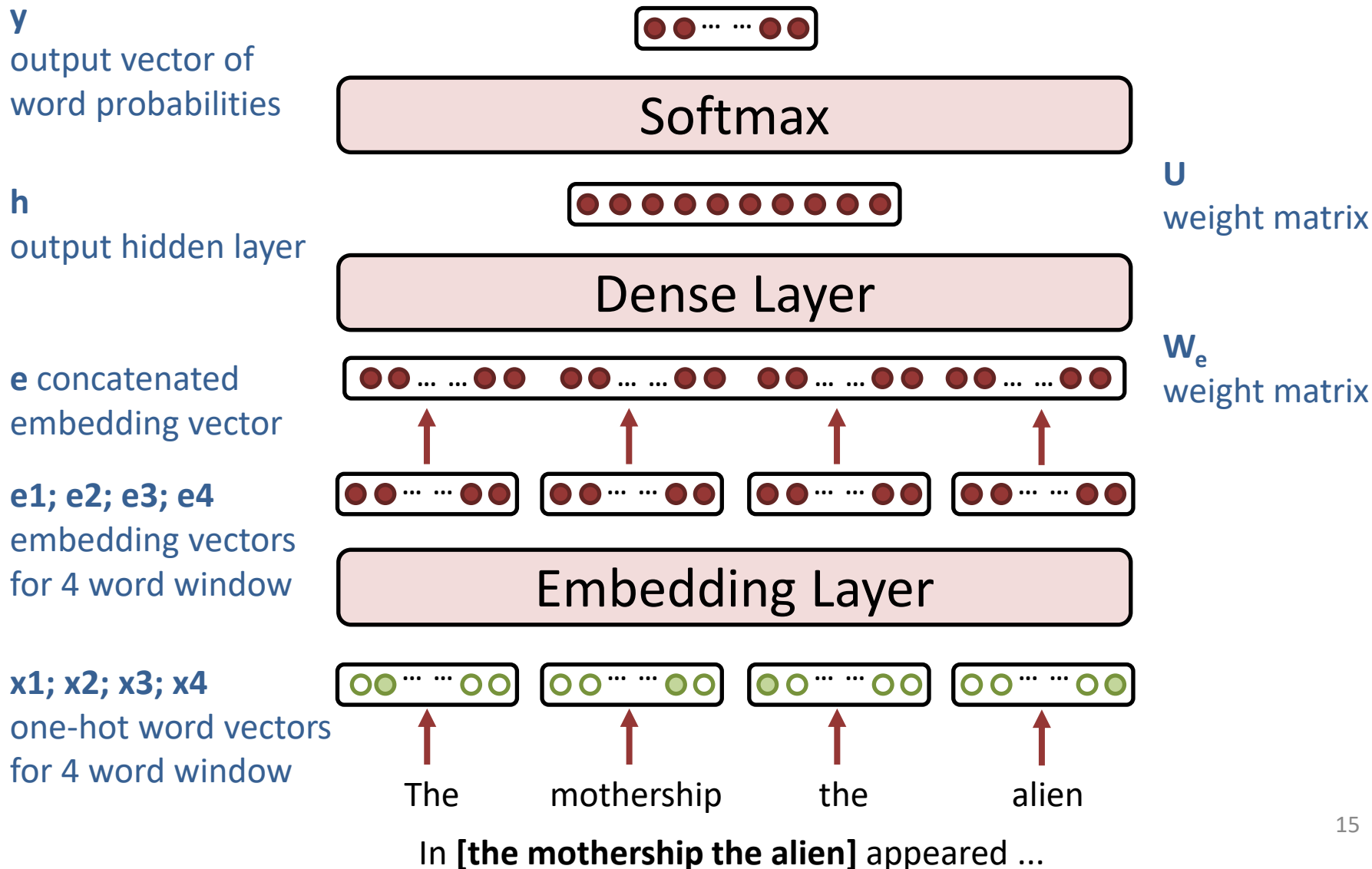


Backpropagation

- PyTorch / Tensorflow will automate the backpropagation
 - Developers still calculate local derivatives at each node/layer
 - Custom loss function executed each training step
 - But ... there are functions provided to calculate this for you
 - e.g. `tf.keras.losses.SparseCategoricalCrossentropy()`
 - If you are researching new methods, you need to calculate it the hard way!
 - Learn more <https://karpathy.medium.com/yes-you-should-understand-backprop-e2f06eab496b>

Matrix shapes in NLP models

- Fixed-window neural language model (predict next word)



Matrix shapes in NLP models

- Fixed-window neural language model (predict next word)

- Training input = n words in an input window

- Vocab size = V

- d_e ; d_h = layer dim size

- input $x = n \times \text{vector}(1 \times V)$

- $E = \text{matrix}(V \times d_e)$

- output $e = \text{vector}(1 \times d_e * n)$

- $W_e = \text{matrix}(d_e * n \times d_h)$

- $b_1 = \text{vector}(1 \times d_h)$ = bias term

- output $h = \text{vector}(1 \times d_h)$

$\gg h_t = f(W_e e^t + b_1)$

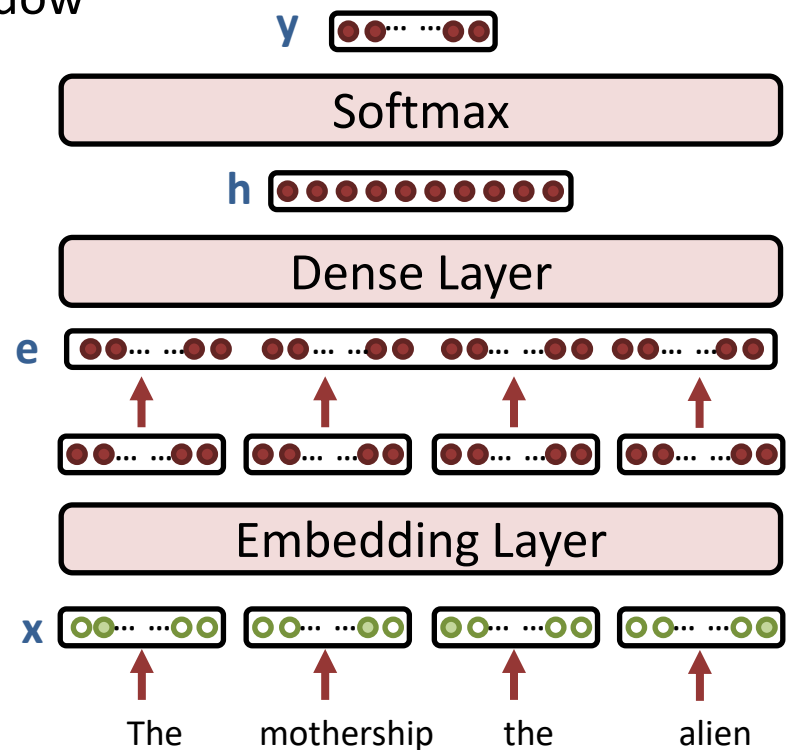
- $U = \text{matrix}(d_h \times V)$

- output $y = \text{vector}(1 \times V)$

$\gg y_t = \text{softmax}(U h_t)$

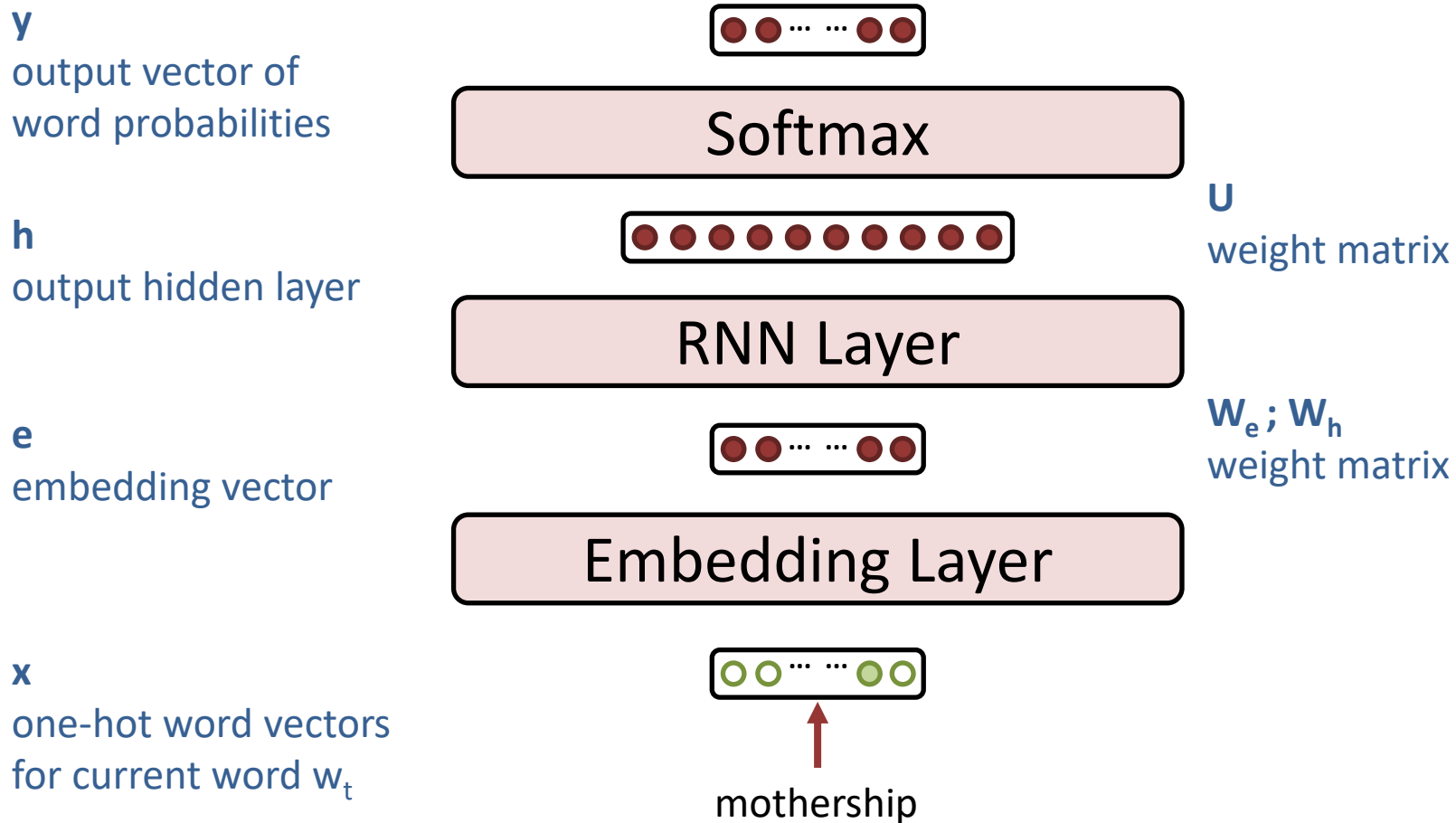
- $J(\theta)$ = cross entropy loss between predicted word (word probability distribution) and true word (one hot vector)

- Batched input (good idea). 3D matrices (batch_size \times _ \times _)



Matrix shapes in NLP models

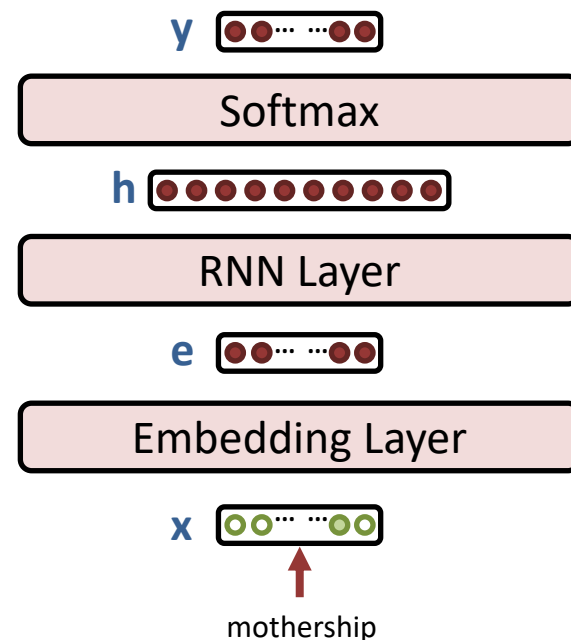
- RNN neural language model (predict next word)



In the **mothership** the alien appeared ...

Matrix shapes in NLP models

- RNN neural language model (predict next word)
 - Training input = single word (from a sentence)
 - Vocab size = V
 - d_e ; d_h = layer dim size
 - input x = vector($1 \times V$)
 - $E = \text{matrix}(V \times d_e)$
 - output e = vector($1 \times d_e$)
 - $W_e = \text{matrix}(d_e \times d_h)$
 - $W_h = \text{matrix}(d_h \times d_h)$
 - $b_1 = \text{vector}(1 \times d_h)$
 - output h = vector($1 \times d_h$)
>> $h_t = f(W_h h_{t-1} + W_e e^t + b_1)$
 - $U = \text{matrix}(d_h \times V)$
 - output y = vector($1 \times V$)
>> $y_t = \text{softmax}(U h_t)$
- $J(\theta)$ = cross entropy loss between predicted word (word probability distribution) and true word (one hot vector)



Design pattern for using word embeddings

- No pre-trained word embeddings?
 - Start with random weights (not zeros) and train on your app dataset
- Pre-trained word embeddings from TensorHub
 - Load pre-trained model (GloVE, word2vec etc) to access its pre-trained word similarity patterns
 - Throw away layers you do not need (keeping at least the embedding layer)
 - Add new layers for your model
 - Fine-tune the new model for your application
- Pre-trained word embedding as layer weights
 - Create new model
 - Replace embedding weights with pre-trained weights
 - Fine-tune the new model for your application
- Allow word embedding weights to change during fine-tuning?
 - large dataset >> updating word embeddings can improve app performance (e.g. size = 100M+, able to cover all useful vocabulary)
 - small dataset >> poor vocab coverage, keep word embeddings fixed usually (will move similarities and could lose value of useful pre-trained patterns)

Tips for performance

- L1 and L2 Regularization
 - Real deep learning models overfit - regularization helps prevent this
 - e.g. L2 regularization $J(\theta) = \text{loss method} + \lambda \sum_k \theta_k^2$
- Use vectors/matrices NOT python variables and iterations
 - Tensorflow / PyTorch much more efficient (10x quicker training)
 - Time / profile code to find problems
- Activation functions
 - tanh better than sigmoid, but both slow (maths is slow)
 - Use ReLU as a default option (simpler/faster than sigmoid but still good)
- Weight initialization
 - Starting with 0 weights generates symmetry that prevents learning
 - Initialize with small random weights (e.g. Xavier initialization)
- Optimizer
 - Stochastic Gradient Decent (SGD) needs manual adjustment of learning rate. Suggest halve the learning rate every k epochs (e.g. 3)
 - Adam is a good default optimizer (don't use SGD unless you have a reason)