Review Word Vector Representations

Review Outline

- Word Vector Representations
- Neural Networks
- Backpropagation / Gradient Calculation
- Dependency Parsing
- RNNs

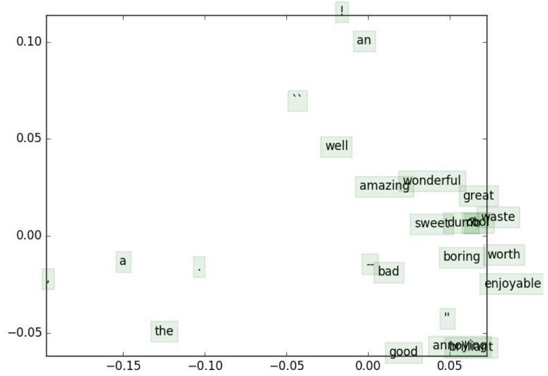
Word Vector Representations

Word Vectors

Definition: A vector that captures the meaning of a word.

Sometimes can also be called as word embeddings or word representations.

We will be reviewing: Word2Vec and GloVe.



Word2Vec

Task**:** Learn word vectors to encode the probability of a word given its context.

Consider the following example with context window size = 2:

"I understand the word2vec model now"

Context words

Center word

Context words

Word2Vec

***Task*:** Learn word vectors to encode the probability of a word given its context.

For each word, we want to learn 2 vectors:

v : **input vector** **u** : **output**

vector Two algorithms:

- **Skipgram:** predicts the probability of context words from a center word.
- **Continuous Bag-of-Words (CBOW):** predicts a center word from the surrounding context in terms of word

Word2Vec - Skipgram

- Predicts the probability of context words from a center word.
- Let's look at the previous example again:

"I understand the word2vec model now"

Context words

Center word

Context words

Word2Vec - Skipgram

- Predicts the probability of context words from a center word.
- Let’s look at the previous example again:

"I word2vec ? ? ? ?”

Context words

Center word

Context words

Word2Vec - Skipgram

" ? ? ?

word2vec ? ”

- Generate a one-hot vector, $\mathbf{w}_c$ of the center word, “word2vec”. It is a

  |VocabSize|-dim vector with a 1 at the word index and 0 elsewhere.

- Look up the input vector, $\mathbf{v}_c$ in $\mathbf{V}$ using $\mathbf{w}_c$. $\mathbf{V}$ is the input vector matrix.

- Generate a score vector, $\mathbf{z} = \mathbf{U}\mathbf{v}_c$ where $\mathbf{U}$ is the output

Word2Vec - Skipgram

"I ? ? word2vec ? ?"

- Turn the score vector into probabilities, $\hat{\mathbf{y}}$ = **softmax(z)**.
- $[\hat{\mathbf{y}}_{c-m}, \dots, \hat{\mathbf{y}}_{c-1}, \hat{\mathbf{y}}_{c+1}, \dots, \hat{\mathbf{y}}_{c+m}]$: probabilities of observing each context word (**m** is the context window size)
- Minimize cost given by: (**F** can be neg-sample or softmax-CE)

$$J_{\text{skip-gram}}(\text{word}_{c-m\dots c+m}) = \sum_{-m \le j \le m, j \ne 0} F(\boldsymbol{w}_{c+j}, \boldsymbol{v}_c)$$

Word2Vec - Continuous Bag-Of-Words (CBOW)

- Predicts a center word from the surrounding context
- Let’s look at the previous example again:

"I understand the word2vec model now”

Context words

Center word

Context words

Word2Vec - Continuous Bag-Of-Words (CBOW)

- Predicts a center word from the surrounding context
- Let’s look at the previous example again:

"I understand the ? model now”

Context words

Center word

Context words

Word2Vec - Continuous Bag-Of-Words (CBOW)

"I understand the ? model now"

- Generate one-hot vectors, $\mathbf{w}_{c-m}, \ldots, \mathbf{w}_{c-1}, \mathbf{w}_{c+1}, \ldots, \mathbf{w}_{c+m}$ for the context words.
- Look up the input vectors, $\mathbf{v}_{c-m}, \ldots, \mathbf{v}_{c-1}, \mathbf{v}_{c+1}, \ldots, \mathbf{v}_{c+m}$ in $\mathbf{V}$ using the one-hot vectors. $\mathbf{V}$ is the input vector matrix.
- Average these vectors to get $\mathbf{v}_{avg} = (\mathbf{v}_{c-m} + \ldots + \mathbf{v}_{c-1} + \mathbf{v}_{c+1} + \ldots + \mathbf{v}_{c+m})/2\mathbf{m}$

Word2Vec - Continuous Bag-Of-Words (CBOW)

"I understand the ? model now"

- Generate a score vector, $\mathbf{z} = \mathbf{U}\mathbf{v}_{avg}$ where **U** is the output vector matrix.
- Turn the score vector into probabilities, $\hat{\mathbf{y}} = \mathbf{softmax}(\mathbf{z})$.
- $\hat{\mathbf{y}}$: probability of the center word.
- Minimize cost given by: (**F** can be neg-sample or softmax-CE)

$$J_{CBOW}(word_{c-m...c+m}) = F(w_c, v_{avg})$$

GloVe

- Like Word2Vec, GloVe is a set of vectors that capture the semantic information (i.e. meaning) about words.
- Unlike Word2Vec, Glove makes use of global co-occurrence statistics.
- Fast Training
- Scalable to huge corpora
- Good Performance even with small corpus and small vectors

“GloVe consists of a weighted least squares model that trains on global word-word co-occurrence counts.”

GloVe

Co-occurrence Matrix (window-based):

Corpus:

- I like Deep Learning.
- I like NLP.
- I enjoy flying.

counts	I	like	enjoy	deep	learning	NLP	flying	.
I	0	2	1	0	0	0	0	0
like	2	0	0	1	0	1	0	0
enjoy	1	0	0	0	0	0	1	0
deep	0	1	0	0	1	0	0	0
learning	0	0	0	1	0	0	0	1
NLP	0	1	0	0	0	0	0	1
flying	0	0	1	0	0	0	0	1
.	0	0	0	0	1	1	1	0

GloVe

- Let **X** be the word-word co-occurrence counts matrix.
  - $\mathbf{X_i}$ is the number of times any word **k** appears in the context of word **i**.
  - $\mathbf{X_{ij}}$ is the number of times word **j** occurs in the context of word **i**.
- Like the case in Word2Vec, each word has 2 vectors, **input (v)** and **output (u)**.
- The cost function:

$$\hat{J} = \sum_{i=1}^{W} \sum_{j=1}^{W} X_i (\vec{u}_j^T \vec{v}_i - \log X_{ij})^2$$

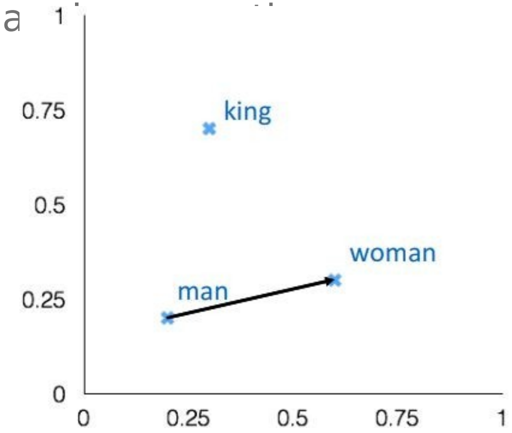
GloVe

- In the end, we have **V** and **U** from all the input and output vectors, **v** and **u.**

- Both capture similar co-occurrence information, and so the word vector for a word can be simply obtained by summing **u** and **v** up!

Evaluate Word Vectors

- Intrinsic Method
    - Word Vector Analogies: Evaluate word vectors by how well their cosine distance after addition captures intuitive semantic and syntactic a



- Extrinsic Method
    - Entity recognition

Neural Networks

Loss Functions

- Prediction of category or label (classification)

  - *Softmax + Cross-Entropy Loss*: optimize correct class

    $$\mathrm{softmax}(\theta)_i = \hat{y}_i = \frac{e^{\theta_i}}{\sum_{j=1}^{C} e^{\theta_j}} \qquad \mathrm{CE}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i)$$

  - *Max-Margin Loss:* optimize margin between correct class score and incorrect class scores.

    $$J = \max(0, 1 - s + s_c)$$

- Prediction of real values or continuous outputs (regression)

  - L2 Loss: $L_2(y, \theta) = ||\theta - y||_2^2$

  - Others: L1, etc.

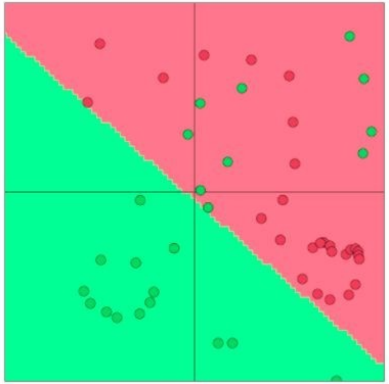
Network Structure

- Recall: forward pass of a neural network. Hidden layers computed as follows:

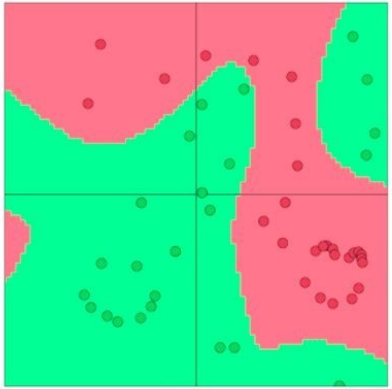
$$\mathbf{h_1} = f(\mathbf{W_1 x} + \mathbf{b_1})$$
$$\mathbf{h_2} = f(\mathbf{W_2 h_1} + \mathbf{b_2})$$
$$\cdots$$
$$\mathbf{h_i} = f(\mathbf{W_i h_{i-1}} + \mathbf{b_i})$$

- Number of hidden layers/size of each hidden layer affects representational power. More parameters => more expressive model.
- Initialization is important
  - Small random numbers (e.g. Xavier/Glorot) for weight matrices.

Linear Model

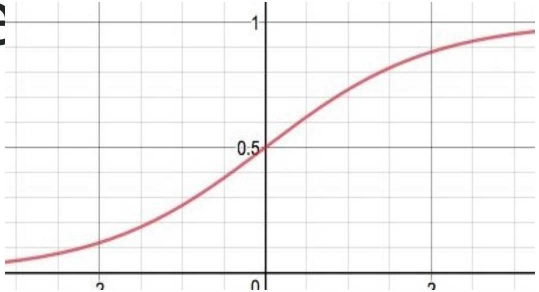


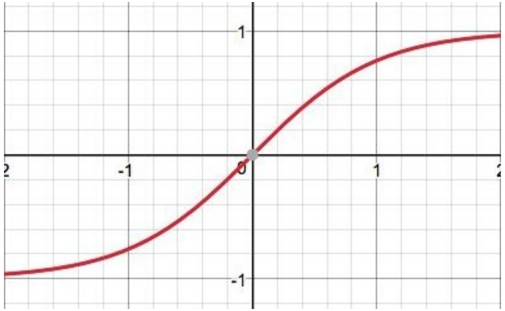
Multilayer Neural Network



Non-Linearitie

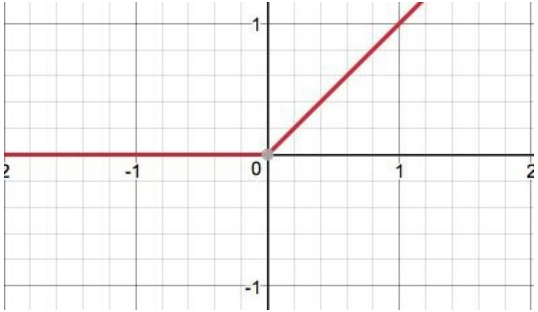
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

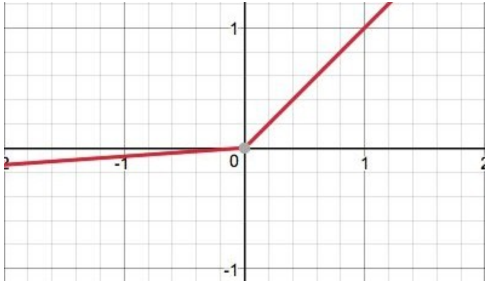




$$\tanh(x)$$
$$= 2\sigma(2x) - 1$$

$$\mathrm{ReLU}(x) = \max(0, x)$$





$$\mathrm{LeakyReLU}(x)$$
$$= \max(\alpha x, x)$$

- Responsible for network's expressiveness -- otherwise just a linear model
- Beware of saturation and "dead" neurons
- Other variants: PreLu, Maxout, Hard Tanh

Gradient Check

- Used to verify correctness of the **analytic gradient**
- Compute **numerical gradient** using the *central difference formula*:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+h) - f(x-h)}{2h}$$

- Vary one dimension of parameters at a time, observe change in output function (loss)
- Potentially very expensive to compute over large numbers of parameters; can sanity check by checking only a few dimensions a time

Optimization

- Optimize loss function with Gradient Descent to compute parameter updates:

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} J(\theta)$$

- Taking gradient over entire training set is expensive, so use mini-batches (Stochastic Gradient Descent)

- In addition to SGD, there are more complicated updates: *Adam* (see PA2),
AdaGrad, *RMSProp*, *Nesterov Momentum,* etc.

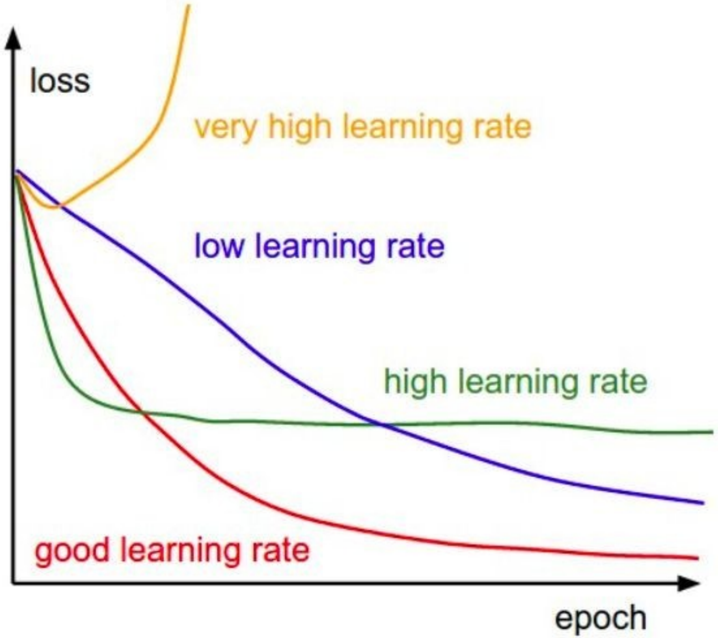
- Sanity check: If network is working properly, should be able to get close to 0 loss on small subset of training data.

- May be helpful to randomize order of examples

Monitoring Learning Curves

- Plot training loss as a function of iteration/time.

- Adjusting learning rate
    - Training loss increases => learning rate too high
    - Training loss plateaus at high value => learning rate too high
    - Linear decrease in training loss => learning rate too low
    - May be helpful to *anneal* learning rate over time



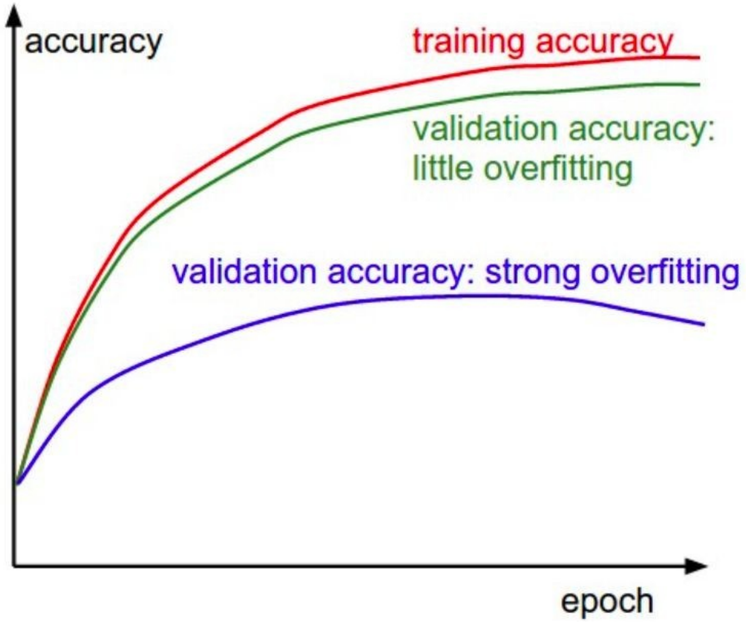
Monitoring Learning Curves

- Also should compare training and validation loss/accuracies

- Large gap => ***Overfitting***: Model does not generalize well to unseen data

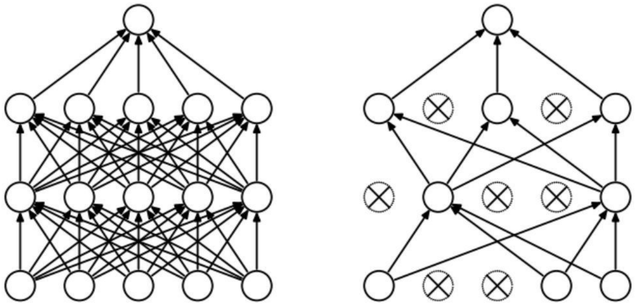
- Bad training performance=> ***Underfitting***: Model is not powerful enough to learn the training data, resulting in bad performance on both training and validation datasets.

- Note: do not compare to test set, which is reserved for final



Handling Overfitting

- Add Dropout
  - Constrain each neuron to learn more meaningful information.
  - Can also be interpreted as an “ensemble” of smaller networks.
  - Need to scale activations to maintain expected value (see PA2)
- L2 Regularization
  - Add $+\lambda||\theta||_2^2$ tunable lambda for non-bias parameters
  - Encourages weights to be more spread out, place less emphasis on any one input dimension
- Reduce Network depth/size
- Reduce input feature dimensionality
- Early Stopping
- Others: Max-Norm, L1 penalty,



(a) Standard Neural Net

(b) After applying dropout.

Handling Underfitting

- Increase model complexity/size
- Decreasing regularization effects
- Reducing L2 penalty weight
- Reducing Dropout probability
- Usually opposite of overfitting solutions

Other Helpful Techniques

- Ensembling
  - Combine separately trained models for more robust predictions
- Data Preprocessing
  - Mean-centering data
- Batch Normalization
  - Encourage outputs after hidden layer to have zero mean, unit variance
- Curriculum Learning
  - During training, present examples in a certain order to speed up optimization
- Data Augmentation
  - Can augment training set with additional examples by applying transformations to input

Backpropagation / Gradient Calculation

Matrix Calculus Primer

Scalar-by-Vector

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \frac{\partial y}{\partial x_2} \cdots \frac{\partial y}{\partial x_n} \end{bmatrix}$$

(We can transpose it to convert it to column shape)

Vector-by-Vector

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

Scalar-by-Matrix

$$\frac{\partial y}{\partial A} = \begin{bmatrix} \frac{\partial y}{\partial A_{11}} & \frac{\partial y}{\partial A_{12}} & \cdots & \frac{\partial y}{\partial A_{1n}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y}{\partial A_{m1}} & \frac{\partial y}{\partial A_{m2}} & \cdots & \frac{\partial y}{\partial A_{mn}} \end{bmatrix}$$

Backpropagation Shape Rule and Dimension Balancing

The gradient at each intermediate step has **shape of denominator**

$$X \in \mathbb{R}^{m \times n} \iff \delta_X = \frac{\delta Scalar}{\delta X} \in \mathbb{R}^{m \times n}$$

- **Dimension balancing** is the “cheap” but efficient way to calculate gradients in most practical settings
- Read **gradient computation notes** to understand how to derive matrix expressions for gradients from first principles
- Dimension balancing approach should be used with a good understanding of what is happening behind it

$$z = Wx$$

$$\frac{\partial z}{\partial x} = W$$

$$z = xW$$

$$\frac{\partial z}{\partial x} = W^T$$

$$z = Wx \quad \frac{\partial J}{\partial z} = \delta$$

$$\frac{\partial J}{\partial W} = \delta x^T$$

$$z = xW \quad \frac{\partial J}{\partial z} = \delta$$

$$\frac{\partial J}{\partial W} = x^T \delta$$

Activation Function

$$\boldsymbol{h} = f(\boldsymbol{z}), \text{what is } \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}? \qquad \boldsymbol{h}, \boldsymbol{z} \in \mathbb{R}^n$$

$$h_i = f(z_i)$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \begin{pmatrix} f'(z_1) & & 0 \\ & \ddots & \\ 0 & & f'(z_n) \end{pmatrix} = \mathrm{diag}(\boldsymbol{f}'(\boldsymbol{z})) \qquad \boldsymbol{f}'(\boldsymbol{z}) = [f'(z_1), f'(z_2), \ldots, f'(z_n)]$$

$$\frac{\partial \mathcal{J}}{\partial \boldsymbol{h}} = \delta_h \quad \Rightarrow \quad \frac{\partial \mathcal{J}}{\partial \boldsymbol{z}} = \frac{\partial \mathcal{J}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} = \delta_h \ \mathrm{diag}(\boldsymbol{f}'(\boldsymbol{z})) = \delta_h \circ \boldsymbol{f}'(\boldsymbol{z})$$

Backpropagation

$$h_1 = \sigma(xW_1 + b_1)$$

$$\hat{y} = softmax(h_1 W_2 + b_2)$$

$$J = CE(\hat{y}, y)$$

$$x \in \mathbb{R}^{D_x}$$

$$W_1 \in \mathbb{R}^{D_x \times D_z}$$

$$b_1 \in \mathbb{R}^{D_z}$$

$$h_1 \in \mathbb{R}^{D_z}$$

$$W_2 \in \mathbb{R}^{D_z \times D_y}$$

$$b_2 \in \mathbb{R}^{D_y}$$

1. Identify intermediate functions (forward prop)
2. Compute local gradients
3. Combine with downstream error signal to get full gradient

Backpropagation

Loss Function:

$$h_1 = \sigma(xW_1 + b_1)$$

$$\hat{y} = softmax(h_1W_2 + b_2)$$

$$J = CE(\hat{y}, y)$$

Intermediate Variables:
(forward propagation)

$$z_1 = xW_1 + b_1$$

$$h_1 = \sigma(z_1)$$

$$\theta = h_1W_2 + b_2$$

$$\hat{y} = softmax(\theta)$$

$$J = CE(\hat{y}, y)$$

Intermediate Variables:
(forward propagation)

$$z_1 = xW_1 + b_1$$
$$h_1 = \sigma(z_1)$$
$$\theta = h_1 W_2 + b_2$$
$$\hat{y} = softmax(\theta)$$
$$J = CE(\hat{y}, y)$$

$$x \in \mathbb{R}^{D_x}$$
$$W_1 \in \mathbb{R}^{D_x \times D_z}$$
$$b_1 \in \mathbb{R}^{D_z}$$
$$h_1 \in \mathbb{R}^{D_z}$$
$$W_2 \in \mathbb{R}^{D_z \times D_y}$$
$$b_2 \in \mathbb{R}^{D_y}$$

Let's do it for x first:

$$\frac{\partial J}{\partial x} = \frac{\partial J}{\partial z_1}\frac{\partial z_1}{\partial x} = \delta_2 W_1^T \in \mathbb{R}^{D_x}$$
$$\delta_2 = \delta_1 \circ h_1 \circ (1 - h_1) \in \mathbb{R}^{D_z}$$
$$\frac{\partial J}{\partial z_1} = \frac{\partial J}{\partial h_1}\frac{\partial h_1}{\partial z_1} = \delta_1 \circ \sigma'(z_1)$$
$$\delta_1 = (\hat{y} - y)W_2^T \in \mathbb{R}^{D_z}$$
$$\frac{\partial J}{\partial h_1} = (\hat{y} - y)W_2^T$$
$$\frac{\partial J}{\partial \theta} = \hat{y} - y$$

Intermediate Variables:
(forward propagation)

$$z_1 = xW_1 + b_1$$
$$h_1 = \sigma(z_1)$$
$$\theta = h_1 W_2 + b_2$$
$$\hat{y} = softmax(\theta)$$
$$J = CE(\hat{y}, y)$$

$$x \in \mathbb{R}^{D_x}$$
$$W_1 \in \mathbb{R}^{D_x \times D_z}$$
$$b_1 \in \mathbb{R}^{D_z}$$
$$h_1 \in \mathbb{R}^{D_z}$$
$$W_2 \in \mathbb{R}^{D_z \times D_y}$$
$$b_2 \in \mathbb{R}^{D_y}$$

Let's continue with: W_2, b_2, W_1, b_1

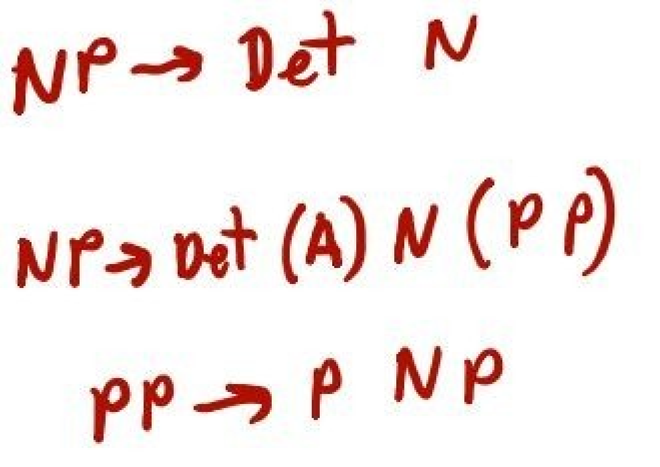
$$\frac{\partial J}{\partial b_1} = \frac{\partial J}{\partial z_1}\frac{\partial z_1}{\partial b_1} = \delta_2$$
$$\frac{\partial J}{\partial W_1} = x^T \delta_2 \in \mathbb{R}^{D_x \times D_z}$$
$$\frac{\partial J}{\partial z_1} = \delta_2$$
$$\frac{\partial J}{\partial b_2} = \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial b_2} = (\hat{y} - y)$$
$$\frac{\partial J}{\partial W_2} = h_1^T(\hat{y} - y) \in \mathbb{R}^{D_z \times D_y}$$
$$\frac{\partial J}{\partial \theta} = \hat{y} - y$$

Summary

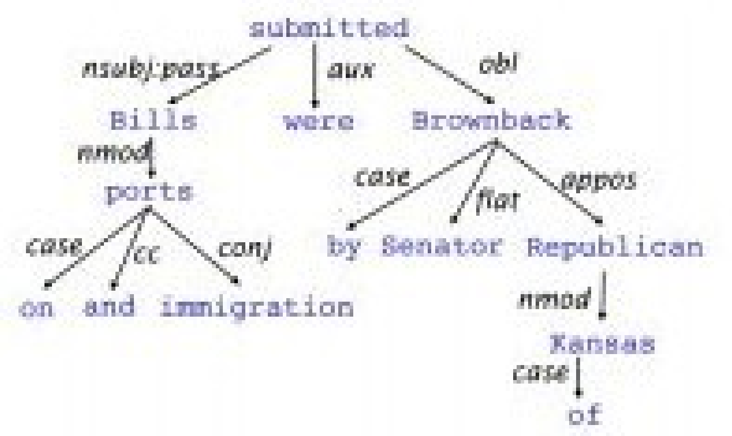
- Identify intermediate functions (forward prop)
- Compute local gradients from top to bottom
- Use Dimension Balancing to double check (or use it to achieve the final result in “hacky” way :))

Dependency Parsing

Two views of Linguistic Structure



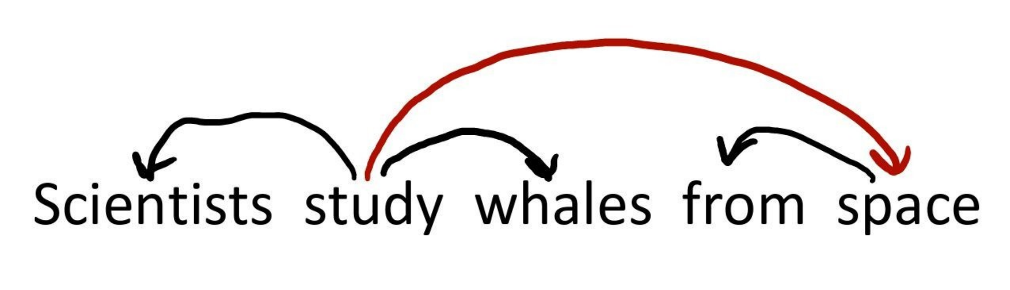
Constituency Structure uses phrase structure grammar to organize words into nested



Dependency Structure uses dependency grammar to identify which words depend on which other words (and how).

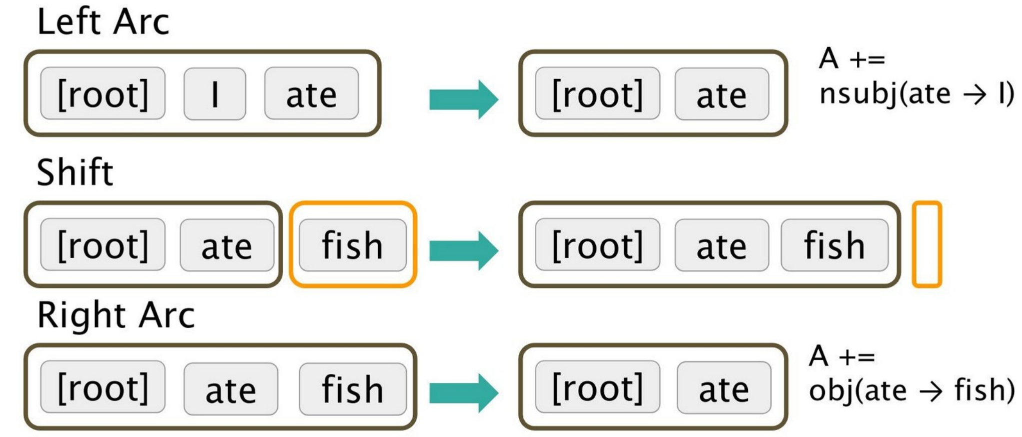
Dependency Parsing

- Asymmetric relations between words (head of the dependency to the dependent).
- Typed with the name of the grammatical relation.
- Usually forms a connected, single-head tree.
- Ambiguities exist



Greedy deterministic transition based parsing

- Bottom up actions analogous to shift-reduce parser
- States defined as a triple of words in buffer, words in stack and set of parsed dependencies.
- Discriminative classification
- Evaluation metrics: UAS, LA
- MaltParser



Projectivity

Projective arcs have no crossing arcs when the words are laid in linear order.

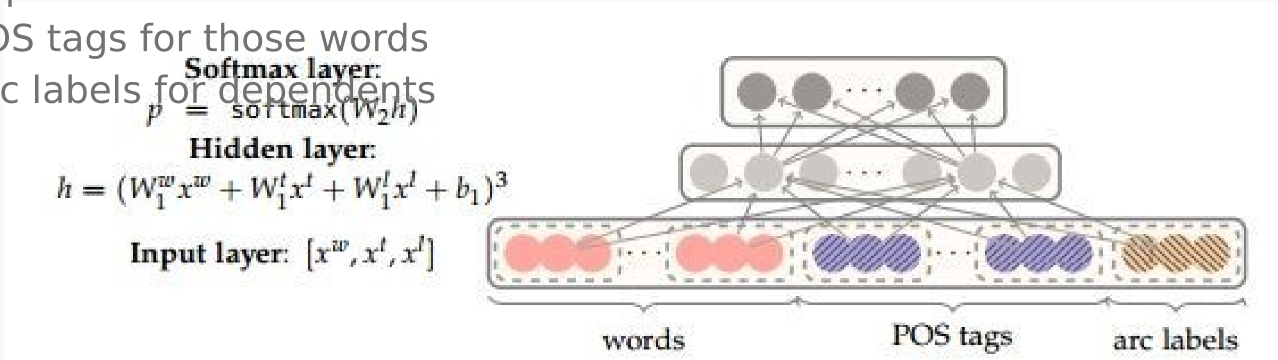
However, some sentences have non-projective dependency structu

Handling non-projectivity

- Declare defeat
- Use post-processor to identify and resolve these non-projective dependencies
- Add extra transitions
- Use a parsing mechanism that doesn’t have projectivity constraint.

Neural Dependency Parsing

- Instead of sparse, one-hot vector representations used in the previous methods, we use embedded vector representations for each feature.
- Features used:
    - Vector representation of first few words in buffer and stack and their dependents
    - POS tags for those words
    - Arc labels for dependents



RNNs

Overview

- Language models
- Applications of RNNs
- Backpropagation of RNNs
- Vanishing gradient problem
- GRUs and LSTMs

A fixed-window neural Language Model

output distribution

$$\hat{\boldsymbol{y}} = \mathrm{softmax}(\boldsymbol{U}\boldsymbol{h} + \boldsymbol{b}_2) \in \mathbb{R}^{|V|}$$

hidden layer

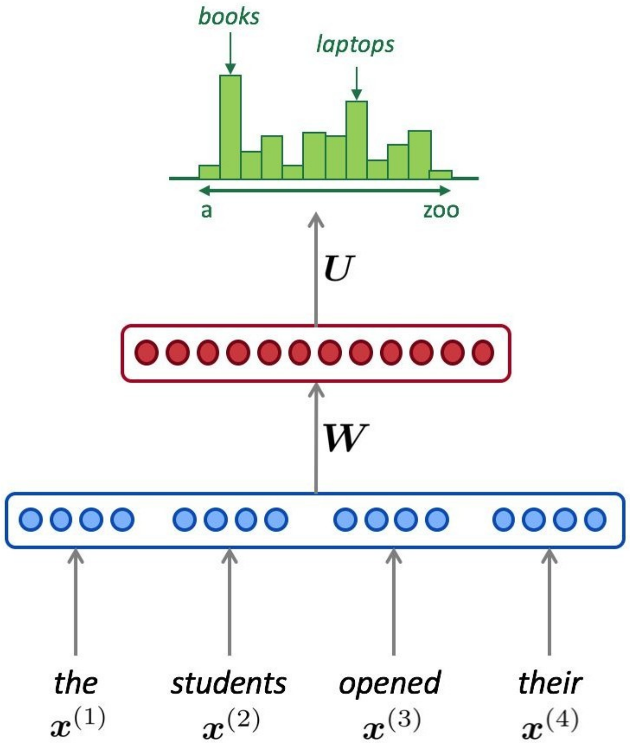
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{e} + b_1)$$

concatenated word

$$\boldsymbol{e} = [\boldsymbol{e}^{(1)}; \boldsymbol{e}^{(2)}; \boldsymbol{e}^{(3)}; \boldsymbol{e}^{(4)}]$$

embeddings words / one-

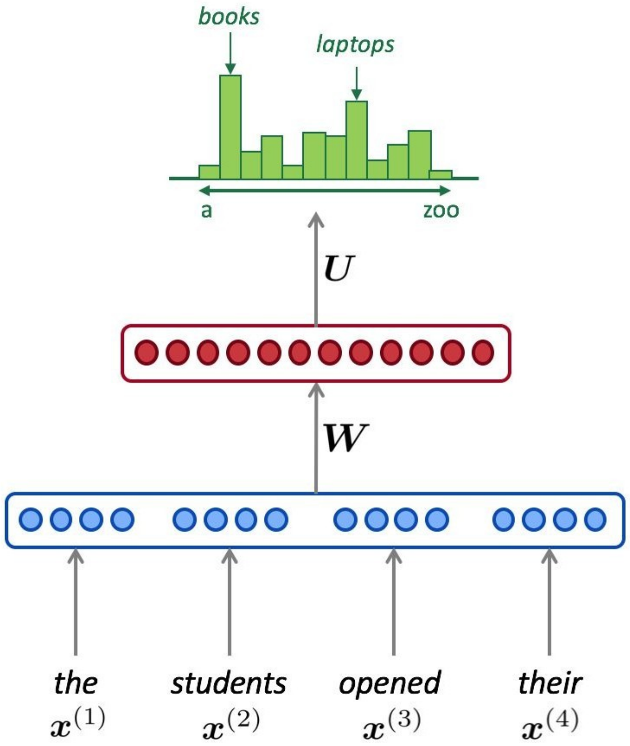
$$\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \boldsymbol{x}^{(3)}, \boldsymbol{x}^{(4)}$$



A fixed-window neural Language Model

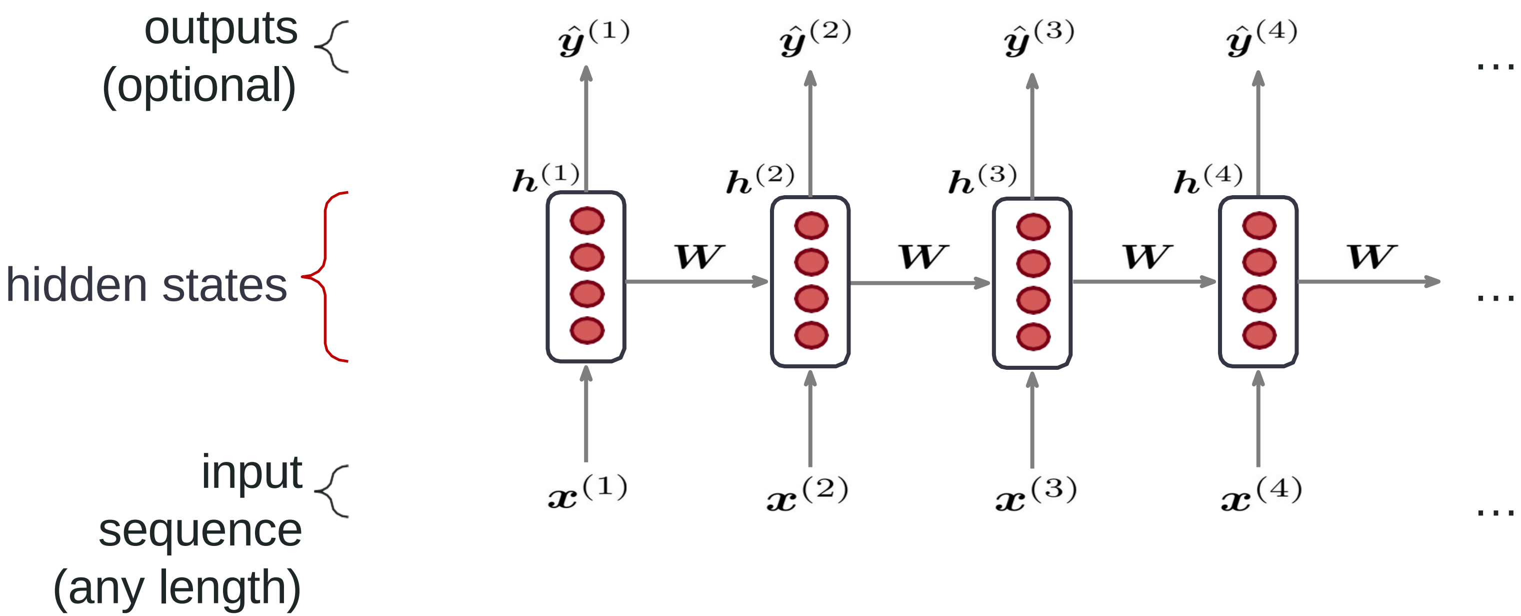
Improvements over *n*-gram LM:
- No sparsity problem
- Model size is O(*n*) not O(exp(*n*))

Remaining **problems**:
- Fixed window is too small
- Enlarging window enlarges
- Window can never be large enough!
- Each $\boldsymbol{x}^{(i)}$ uses different rows $\boldsymbol{W}$. We don't share weights across the window.

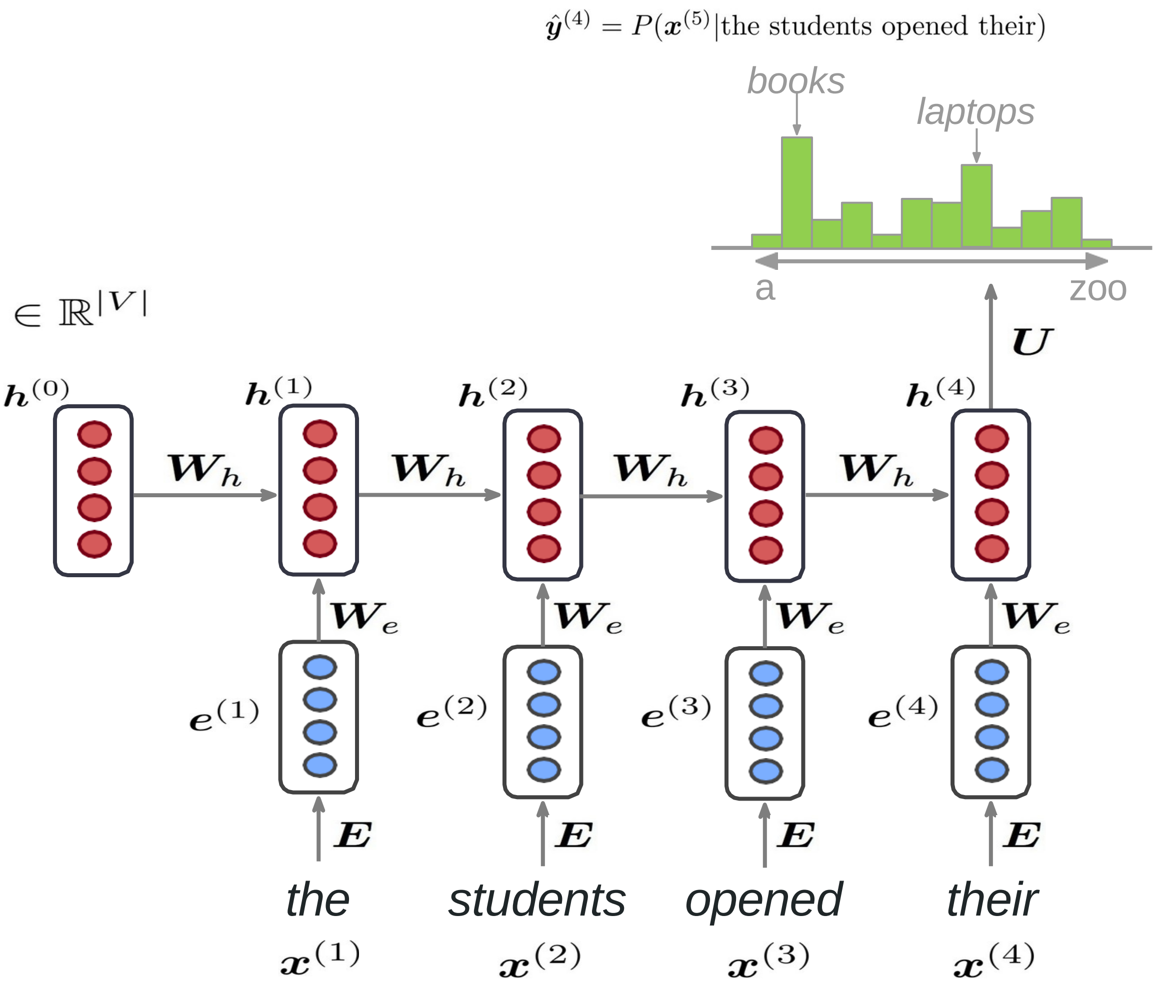


Recurrent Neural Networks (RNN)

<u>Core idea:</u> Apply the same weights *repeatedly*



RNN Language Model



output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

$$\boldsymbol{e}^{(t)} = \boldsymbol{E}\boldsymbol{x}^{(t)}$$

words / one-hot vectors

$$\boldsymbol{x}^{(t)} \in \mathbb{R}^{|V|}$$

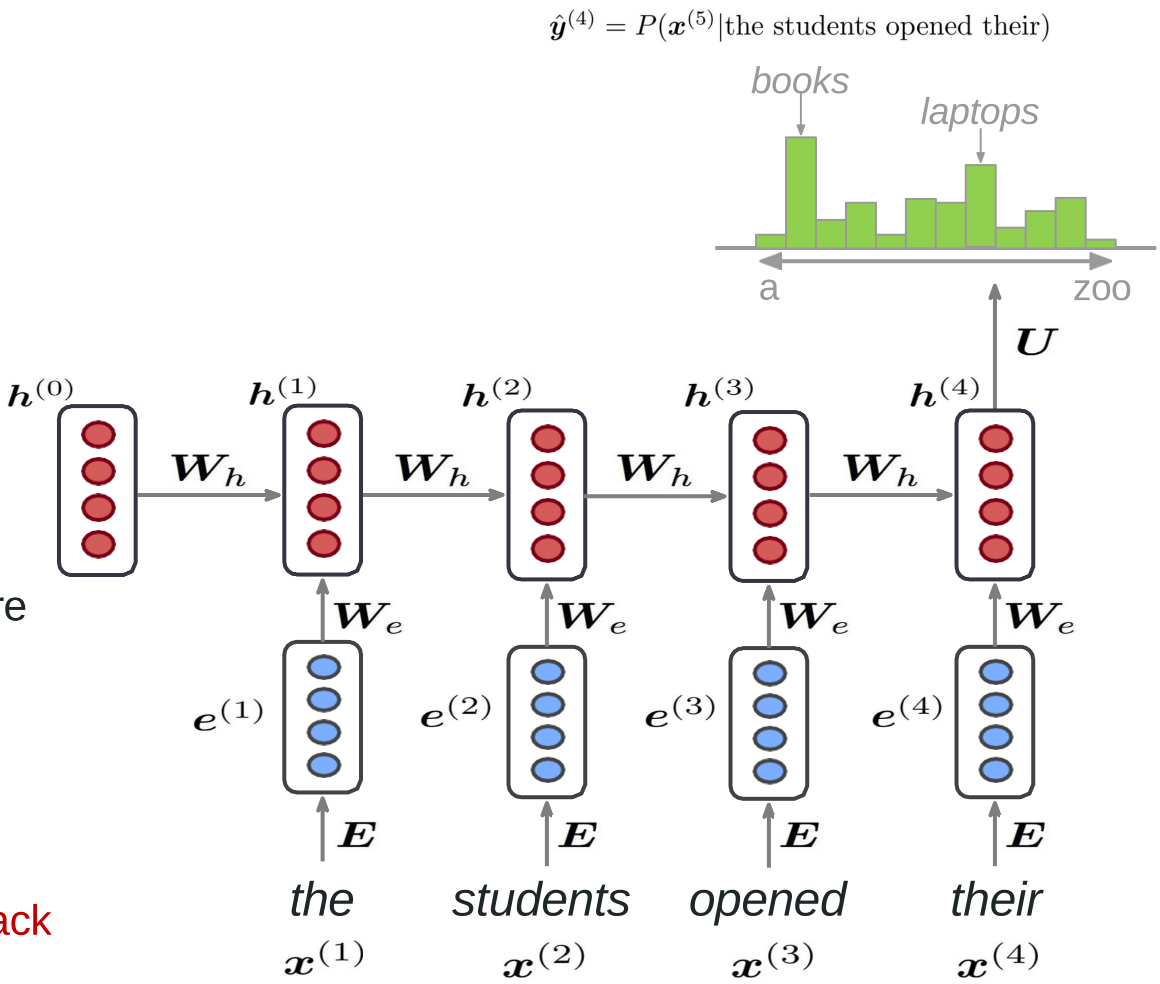
RNN Language Model

RNN **Advantages**:

- Can process any length input
- Model size doesn’t increase for longer input
- Computation for step *t* can (in theory) use information from many steps back
- Weights are shared across timesteps → representations are shared

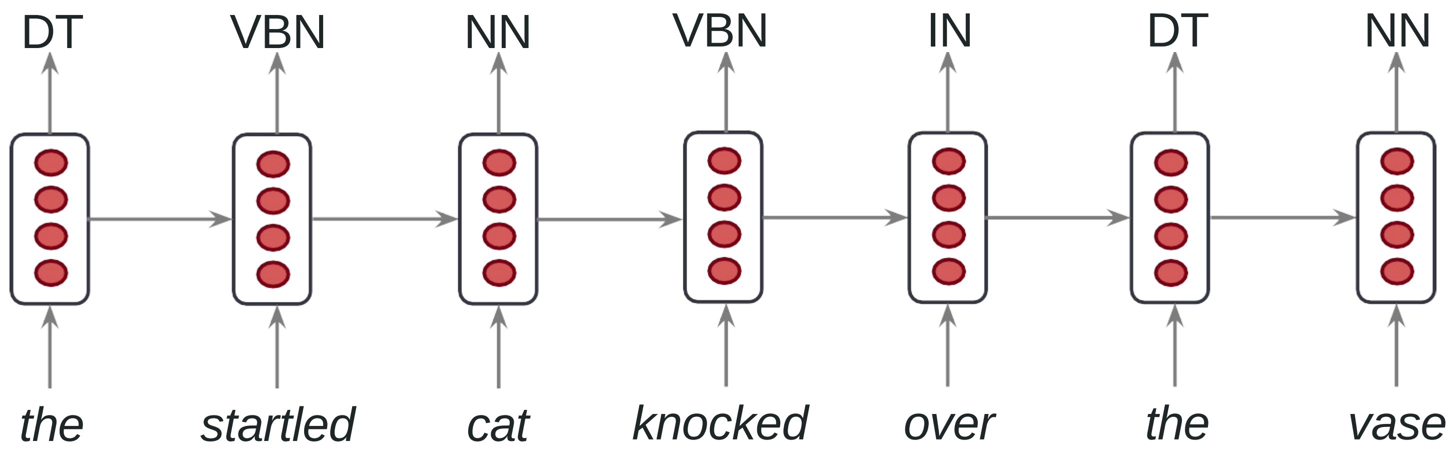
RNN **Disadvantages**:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back



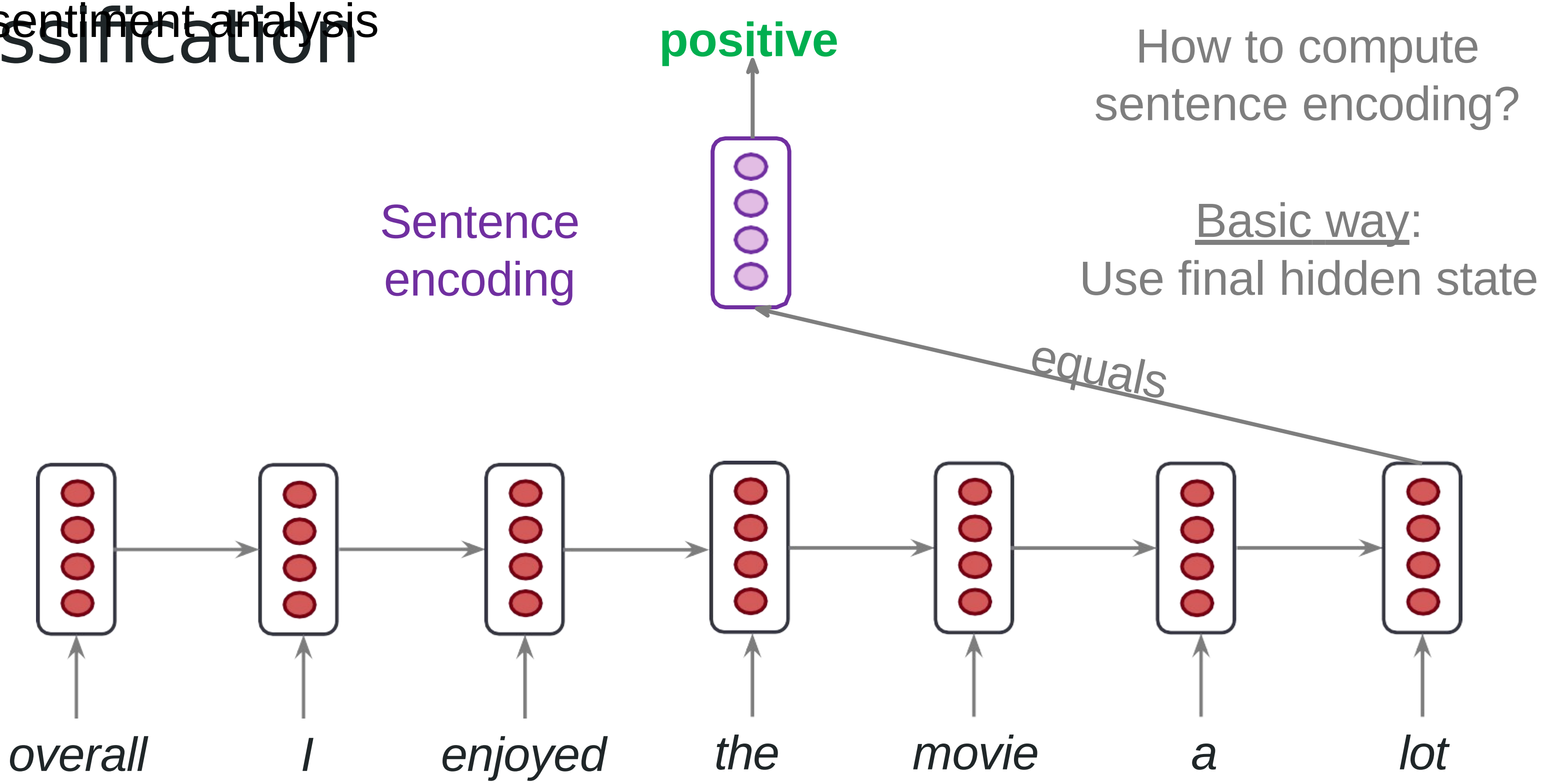
RNNs can be used for tagging

e.g. part-of-speech tagging, named entity recognition



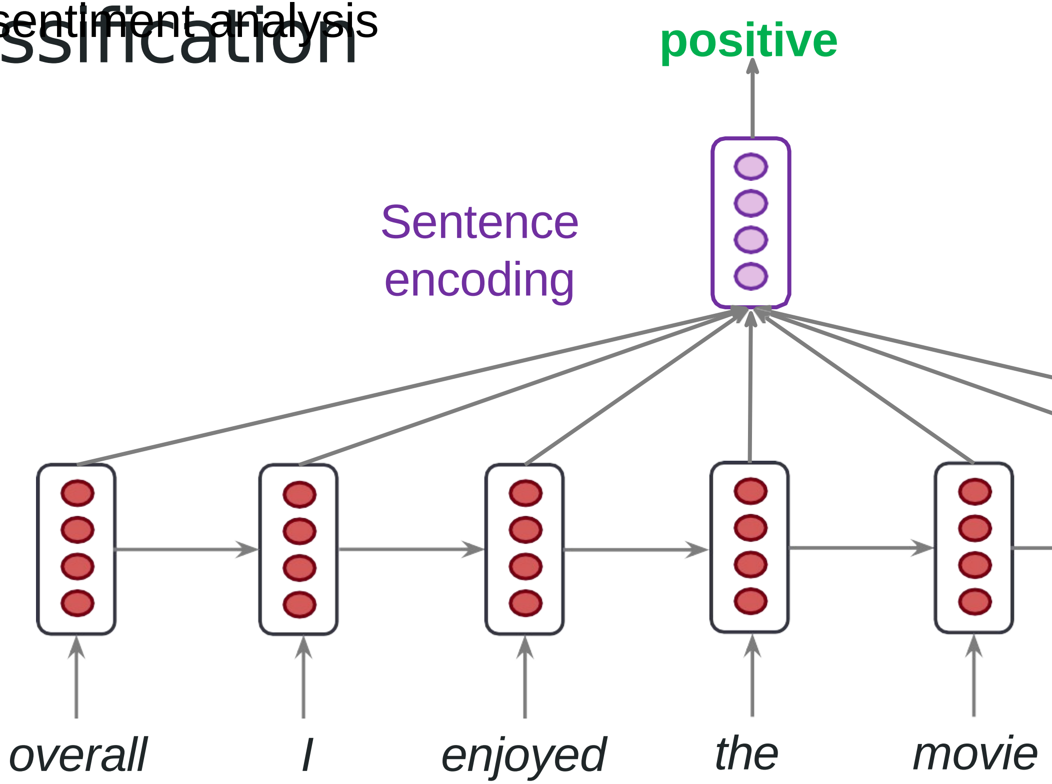
RNNs can be used for classification

e.g. sentiment analysis



RNNs can be used for classification

e.g. sentiment analysis

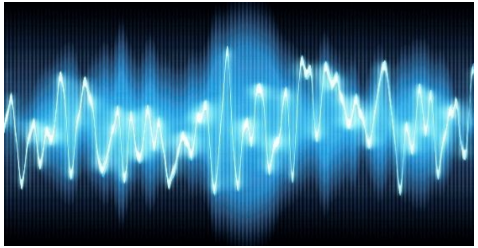


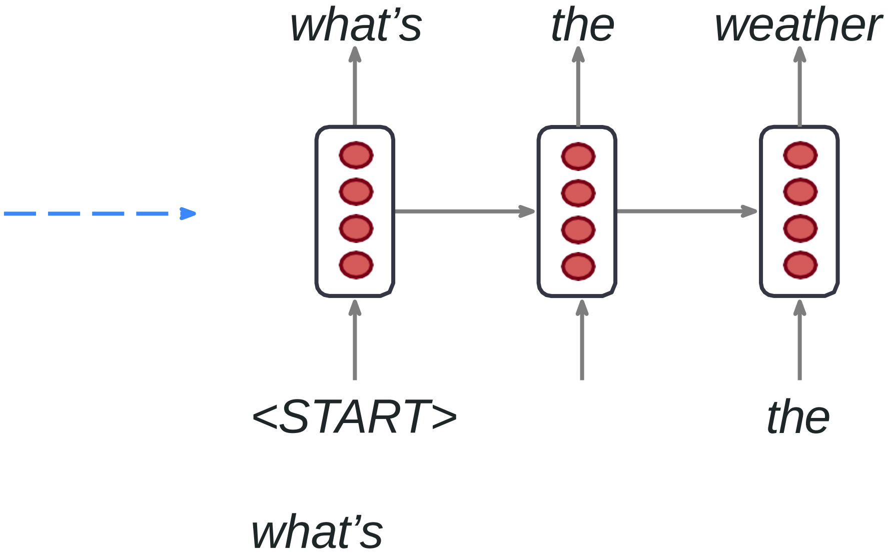
How to compute sentence encoding?

<u>Usually better</u>:
Take element-wise max or mean of all hidden states

RNNs can be used to generate text

e.g. speech recognition, machine translation, summarization



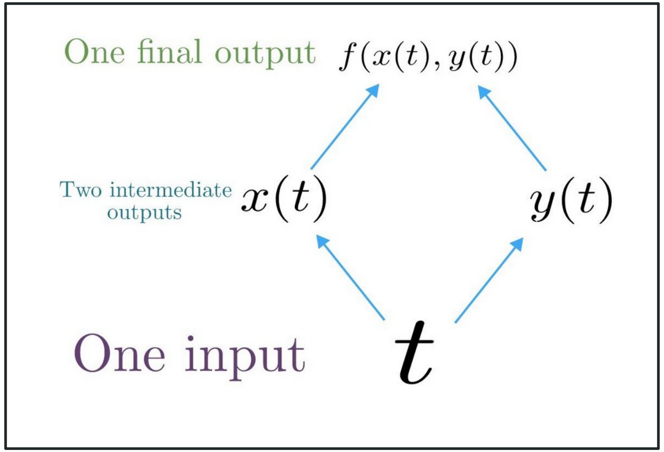


Can use a RNN Language Model to generate text by repeated sampling. Sampled output is next step's input,

Multivariable Chain Rule

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$



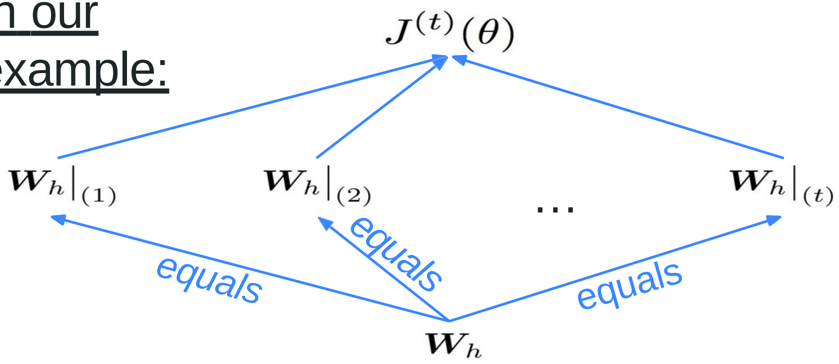
Source:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs

- Given a multivariable function $f(x, y)$, and two single variable functions $x(t)$ and $y(t)$, here's what the multivariable chain rule says:

$$\underbrace{\frac{d}{dt} f(x(t), y(t))}_{\text{Derivative of composition function}} = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

In our example:



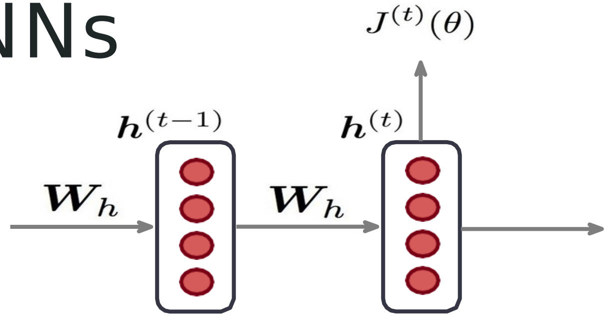
Apply the multivariable chain rule:

$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\right|_{(i)} \boxed{\frac{\partial \boldsymbol{W}_h|_{(i)}}{\partial \boldsymbol{W}_h}} \quad = 1$$

$$= \sum_{i=1}^{t} \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\right|_{(i)}$$

Source: https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/differentiating-vector-valued-functions/a/multivariable-chain-rule-simple-version

Backpropagation for RNNs



$$\hat{\boldsymbol{y}}^{(t)} = \operatorname{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

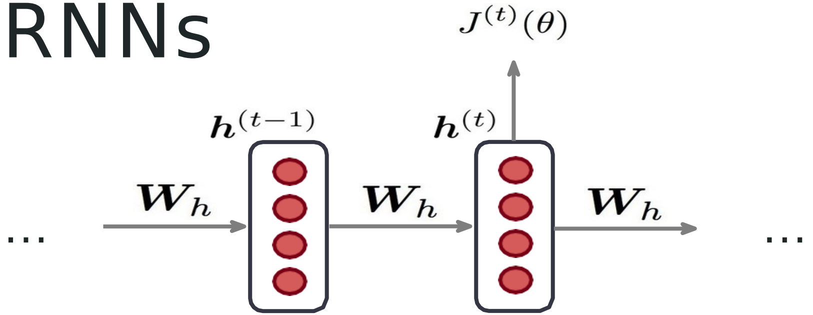
$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$$z^{(t)} = \boldsymbol{W}_h h^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1$$

$$\theta^{(t)} = \boldsymbol{U} h^{(t)} + \boldsymbol{b}_2$$

Recall $\boldsymbol{W}_h$ appears at every time step. Calculate the sum of gradients w.r.t each time it appears

Backpropagation for RNNs



$$\hat{\boldsymbol{y}}^{(t)} = \mathrm{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

$$z^{(t)} = \boldsymbol{W}_h h^{(t-1)} + \boldsymbol{W}_e \boldsymbol{e}^{(t)} + \boldsymbol{b}_1$$

$$\theta^{(t)} = \boldsymbol{U} h^{(t)} + \boldsymbol{b}_2$$

Recall $\boldsymbol{W}_h$ appears at every time step. Calculate the sum of gradients w.r.t each time it appears

Question: Consider only the last two time steps, t and t-1. What's the derivative $\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}$? Leave as a chain rule

Answer: $$\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h} = \sum_{i=t-1}^{t} \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W}_h}\right|_{(i)} = \frac{\partial J^{(t)}}{\partial \boldsymbol{\theta}^{(t)}} \frac{\partial \boldsymbol{\theta}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{z}^{(t)}} \frac{\partial \boldsymbol{z}^{(t)}}{\partial \boldsymbol{W}_h} + \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{z}^{(t-1)}} \frac{\partial \boldsymbol{z}^{(t-1)}}{\partial \boldsymbol{W}_h} \right)$$

Looks scary!

Gradient Problems

- Backprop in RNNs have a recursive gradient call for hidden layer
- Magnitude of gradients of typical activation functions (sigmoid, tanh) lie between 0 and 1. Also depends on repeated multiplications of W matrix.
- If gradient magnitude is small/big, increasing timesteps decreases/increases the final magnitude.
- RNNs fail to learn long term dependencies.

How to solve:

Exploding Gradients

- gradient clipping

Vanishing Gradients

- use GRUs or LSTMs

Gated Recurrent Units (GRU)

- Reset gate, r_t
- Update gate, z_t
- Intuition:
  - High r_t => Short-term dependencies
- High z_t => Long-term dependencies (solves vanishing gradients problem)

$$z_t = \sigma\left(W^{(z)}x_t + U^{(z)}h_{t-1}\right)$$

$$r_t = \sigma\left(W^{(r)}x_t + U^{(r)}h_{t-1}\right)$$

$$\widetilde{h}_t = \tanh\left(Wx_t + r_t \circ Uh_{t-1}\right)$$

$$h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \widetilde{h}_t$$

Long-Short-Term-Memories (LSTM)

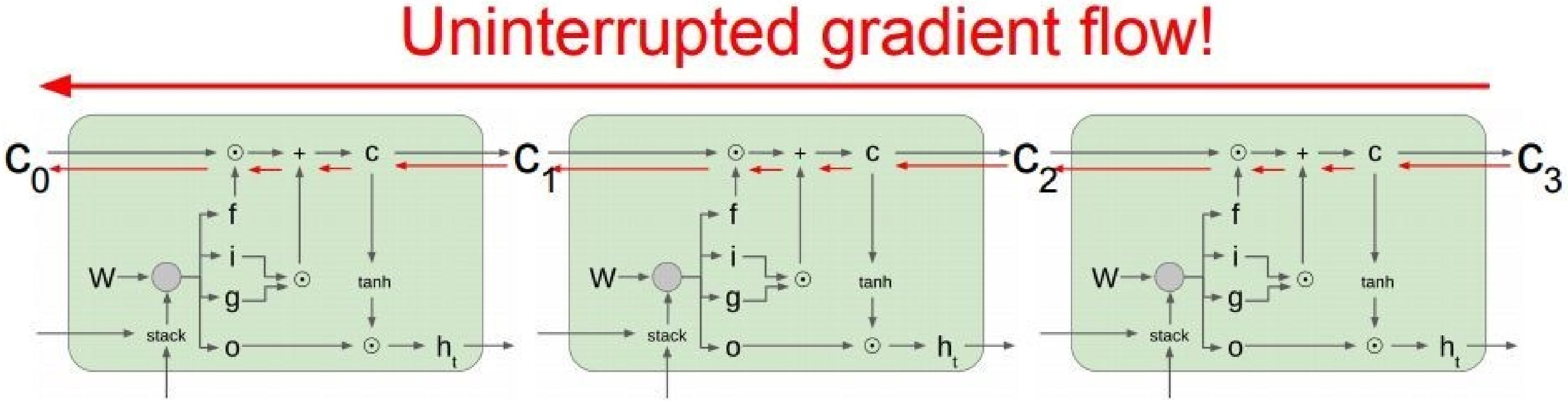
- i_t: Input gate - How much does current input matter
- f_t: Forget gate - How much does past matter
- o_t: Output gate - How much should current cell be exposed
- c_t: New memory - Memory from current cell

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$$
$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$$
$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$$
$$\widetilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t$$
$$h_t = o_t \circ \tanh\left(c_t\right)$$

Long-Short-Term-Memories (LSTM)

$$i_t = \sigma\left(W^{(i)}x_t + U^{(i)}h_{t-1}\right)$$
$$f_t = \sigma\left(W^{(f)}x_t + U^{(f)}h_{t-1}\right)$$
$$o_t = \sigma\left(W^{(o)}x_t + U^{(o)}h_{t-1}\right)$$
$$\widetilde{c}_t = \tanh\left(W^{(c)}x_t + U^{(c)}h_{t-1}\right)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t$$
$$h_t = o_t \circ \tanh\left(c_t\right)$$

Backpropagation from c_t to c_{t-1} only elementwise multiplication by f_t. No longer only depends on dh_t/dh_{t-1}



Source:
http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture10.pdf