NLP_WEEK 5

Lecture 5 - Recurrent Neural networks (RNNs)

Lecture plan

1. Nerual dependency parsing
2. A bit more about neural networks
3. Language modeling + RNNs

1. How do we gain from a neural dependency parser? Indicator Features Revisited

problem: sparse and incomplete

Neural Approach: learn a dense and compact feature representation

First win : Distributed Representations

- d-dimensional vector(word embedding)
  - similar words are expected to have close vector

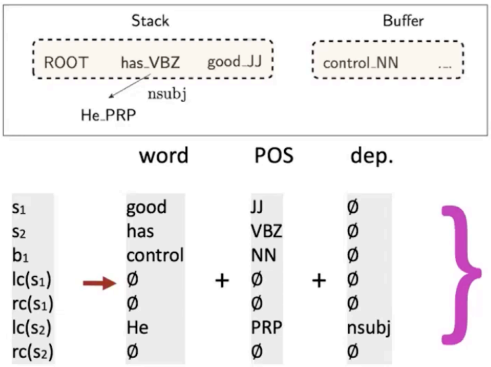
-Meanwhile POS and dependency labels are also represented as d-dimensional vectors
  - The smaller discrete sets also exhibit many semantical similarities

NNS (plural noun) should be close to NN (singular noun).

nummod (numerical modifier) should be close to amod (adjective modifier).

Extracting Tokens & vector representations from configuration

- We extract a set of tokens based on the stack / buffer positions:



Second win : Deep Learning classifiers are non-linear classifiers

- A softmax classifier assigns classes $y \in C$ based on inputs $x \in \mathbb{R}^d$ via the probability:

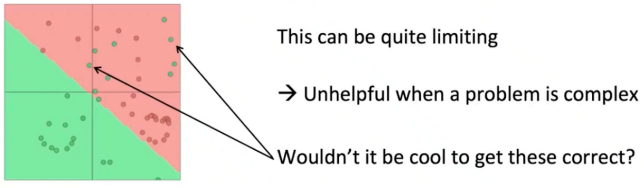
$$p(y|x) = \frac{\exp(W_{y\cdot}x)}{\sum_{c=1}^{C}\exp(W_{c\cdot}x)}$$

a.k.a. "cross entropy loss"

- We train the weight matrix $W \in \mathbb{R}^{C\times d}$ to minimize the neg. log loss : $\sum_i -\log p(y_i|x_i)$
- Traditional ML classifiers (including Naïve Bayes, SVMs, logistic regression and softmax classifier) are not very powerful classifiers: they only give linear decision boundaries

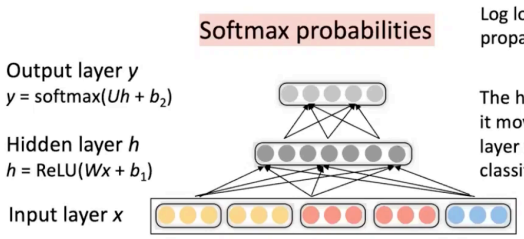
(softmax classifier is indeed a liear classifier)

->



neural networks can learn much more complex functions with nonlinear decision boundaries!

Simple feed-forward neural network multi-class classifier

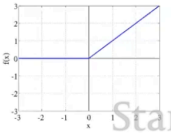


Log loss (cross-entropy error) will be back-propagated to the embeddings

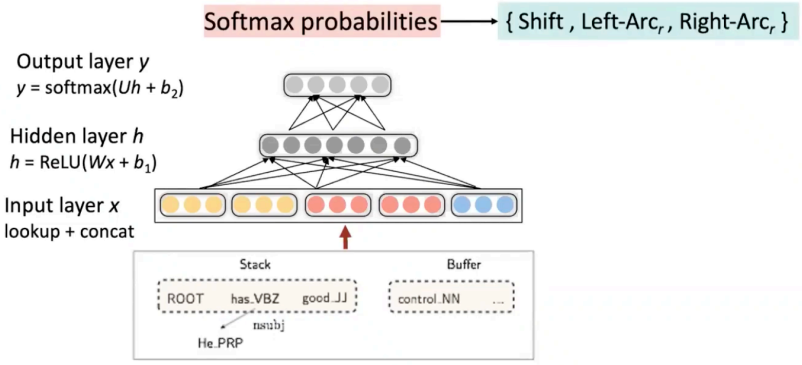
The hidden layer re-represents the input — it moves inputs around in an intermediate layer vector space—so it can be easily classified with a (linear) softmax

ReLU = Rectified Linear Unit

$\text{rect}(z) = \max(z, 0)$

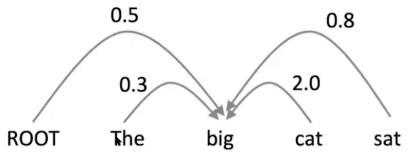


Neural Dependency Parser Model Architecture



Graph-based dependency parsers

- Compute a score for every possible dependency (choice of head) for each word
  - Doing this well requires more than just knowing the two words
  - We need good "contextual" representations of each word token, which we will develop in the coming lectures
- Repeat the same process for each other word; find the best parse (MST algorithm)



e.g., picking the head for "big"

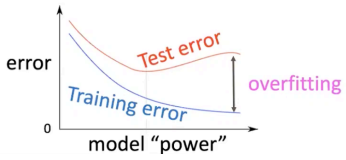
2. A bit more about neural networks

we have models with many parameters! Regularization!

- A full loss function includes regularization over all parameters θ, e.g., L2 regularization:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N} -\log\left(\frac{e^{f_{y_i}}}{\sum_{c=1}^{C} e^{f_c}}\right) + \lambda \sum_k \theta_k^2$$

- Classic view: Regularization works to prevent overfitting when we have a lot of features (or later a very powerful/deep model, etc.)



17

Dropout

- preventing Feature co-adaption = Good Regularization Method

Vectorization

- E.g., looping over word vectors versus concatenating them all into one large matrix and then multiplying the softmax weights with that matrix:

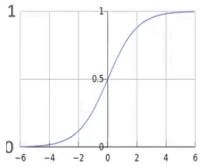
```
from numpy import random
N = 500 # number of windows to classify
d = 300 # dimensionality of each window
C = 5 # number of classes
W = random.rand(C,d)
wordvectors_list = [random.rand(d,1) for i in range(N)]
wordvectors_one_matrix = random.rand(d,N)

%timeit [W.dot(wordvectors_list[i]) for i in range(N)]
%timeit W.dot(wordvectors_one_matrix)
```

- 1000 loops, best of 3: **639 µs** per loop
  10000 loops, best of 3: **53.8 µs** per loop ← Now using a single a C x N matrix
- Matrices are awesome!!! Always try to use vectors and matrices rather than for loops!

Non-linearities, old and new

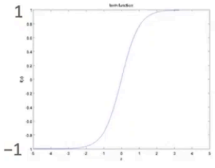
logistic ("sigmoid")

$$f(z) = \frac{1}{1+\exp(-z)}.$$



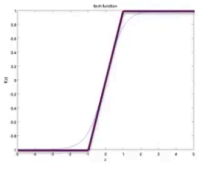
tanh

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$



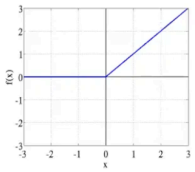
hard tanh

$$\text{HardTanh}(x) = \begin{cases} -1 & \text{if } x < -1 \\ x & \text{if } -1 <= x <= 1 \\ 1 & \text{if } x > 1 \end{cases}$$



ReLU (Rectified Linear Unit)

$$\text{rect}(z) = \max(z, 0)$$



tanh is just a rescaled and shifted sigmoid (2 × as steep, [−1,1]):

$$\tanh(z) = 2\,\text{logistic}(2z) - 1$$

Both logistic and tanh are still used in various places (e.g., to get a probability), but are no longer the defaults for making deep networks

Parameter Initialization

- You normally must initialie weights to small random values
- Initialie all other weights ~ Uniform(-r,r) with r chosen so numbers get neither too big or small

Optimizers

- Adagrad, RMSprop, Adam, SparseAdam

Learning Rates

3. Language Modeling + RNN

Language Modeling

- **Language Modeling** is the task of predicting what word comes next

the students opened their _____ → *books*, *laptops*, *exams*, *minds*

- More formally: given a sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$, compute the probability distribution of the next word $x^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)})$$

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|V|}\}$

- You can also think of a Language Model as a system that assigns probability to a piece of text

- For example, if we have some text $x^{(1)}, \ldots, x^{(T)}$, then the probability of this text (according to the Language Model) is:

$$P(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)}) = P(\boldsymbol{x}^{(1)}) \times P(\boldsymbol{x}^{(2)} | \boldsymbol{x}^{(1)}) \times \cdots \times P(\boldsymbol{x}^{(T)} | \boldsymbol{x}^{(T-1)}, \ldots, \boldsymbol{x}^{(1)})$$
$$= \prod_{t=1}^{T} P(\boldsymbol{x}^{(t)} | \boldsymbol{x}^{(t-1)}, \ldots, \boldsymbol{x}^{(1)})$$

This is what our LM provides

n-gram Language Models

Q : How to learn a Language Model?
A (pre-deep Learning) : learn a n-gram Language Model

n-gram is a chunk of n consecutive words

ex) bigrams : 'the students', 'students opened'

Idea : Collect statistics about how frequent different n-grams are and use these to predict next word.

- First we make a Markov assumption: $x^{(t+1)}$ depends only on the preceding *n*-1 words

$$P(\boldsymbol{x}^{(t+1)} | \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(1)}) = P(\boldsymbol{x}^{(t+1)} | \overbrace{\boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(t-n+2)}}^{n\text{-1 words}}) \quad \text{(assumption)}$$

$$= \frac{P(\boldsymbol{x}^{(t+1)}, \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(t-n+2)})}{P(\boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(t-n+2)})} \quad \text{(definition of conditional prob)}$$

prob of a n-gram / prob of a (n-1)-gram

- **Question:** How do we get these *n*-gram and (*n*-1)-gram probabilities?
- **Answer:** By counting them in some large corpus of text!

$$\approx \frac{\text{count}(\boldsymbol{x}^{(t+1)}, \boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(t-n+2)})}{\text{count}(\boldsymbol{x}^{(t)}, \ldots, \boldsymbol{x}^{(t-n+2)})} \quad \text{(statistical approximation)}$$

n-gram Language Models : Example

Suppose we are learning a 4-gram Language Model.

~~as the proctor started the clock, the~~ *students opened their* _____

discard

condition on this

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count(students opened their } \boldsymbol{w})}{\text{count(students opened their)}}$$

Sparsity Problems with n-gram Language Models

Sparsity Problem 1

Problem: What if *"students opened their w"* never occurred in data? Then *w* has probability 0!

(Partial) Solution: Add small δ to the count for every $w \in V$. This is called *smoothing*.

$$P(\boldsymbol{w}|\text{students opened their}) = \frac{\text{count(students opened their } \boldsymbol{w})}{\text{count(students opened their)}}$$

Sparsity Problem 2

Problem: What if *"students opened their"* never occurred in data? Then we can't calculate probability for *any w*!

(Partial) Solution: Just condition on *"opened their"* instead. This is called *backoff*.

Note: Increasing *n* makes sparsity problems *worse*. Typically, we can't have *n* bigger than 5.

n-gram Language Models in practice

You can also use a Language Model to generate text

today the price _____

condition on this

get probability distribution

of	0.308
for	0.050
it	0.046
to	0.046
is	0.031
…	

sample

You can also use a Language Model to generate text

today the price of _____

condition on this

get probability distribution

the	0.072
18	0.043
oil	0.043
its	0.036
gold	0.018
…	

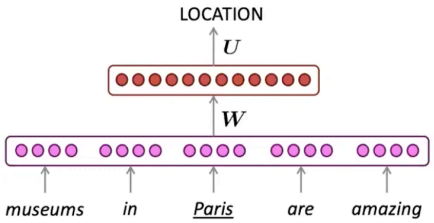
sample

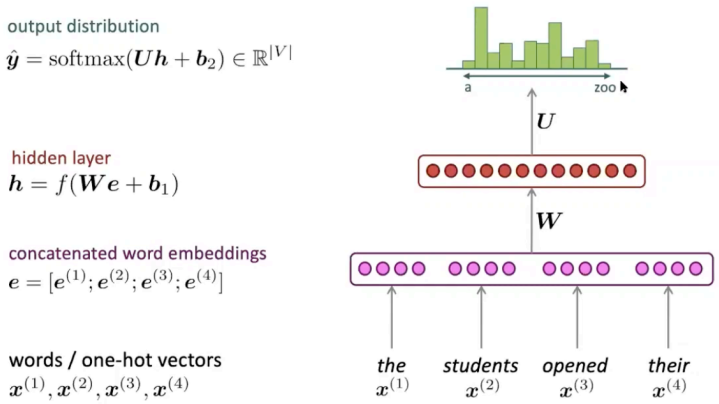
You can also use a Language Model to generate text

today the price of gold per ton , while production of shoe lasts and shoe industry , the bank intervened just after it considered and rejected an imf demand to rebuild depleted european stocks , sept 30 end primary 76 cts a share .

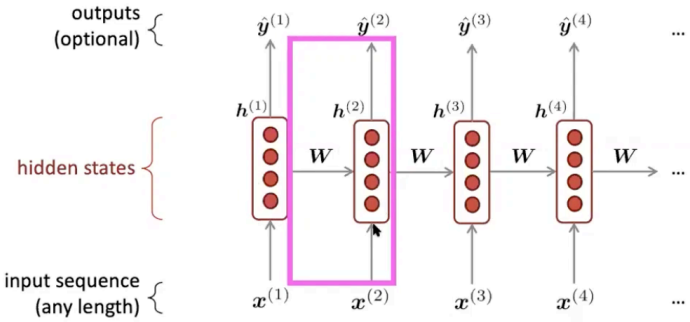
How to build a neural Language Model?

- Recall the Language Modeling task:
  - Input: sequence of words $x^{(1)}, x^{(2)}, \ldots, x^{(t)}$
  - Output: prob dist of the next word $P(x^{(t+1)} | x^{(t)}, \ldots, x^{(1)})$
- How about a window-based neural model?
  - We saw this applied to Named Entity Recognition in Lecture 3:





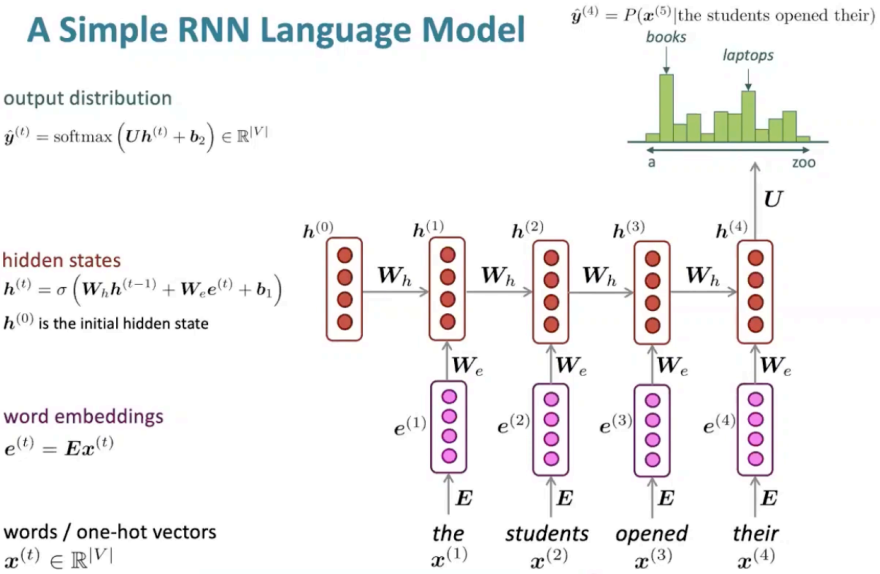
Recurrent Neural Networks(RNN)



Core idea : Apply the same weights W repeatedly

next hidden layer based on the next input word and the previous hidden state by updating it by mutiplying it by a matrix W

A Simple 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 **Advantages**:

- Can process any length input
- Computation for step *t* can (in theory) use information from many steps back
- Model size doesn't increase for longer input context
- Same weights applied on every timestep, so there is symmetry in how inputs are processed.

RNN **Disadvantages**:

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