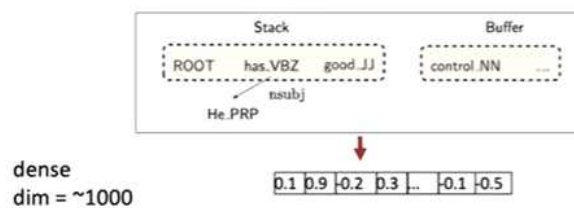


Lec 5 - Recurrent Neural networks(RNNs)

1. Neural dependency parsing

- problems: 1. sparse, 2. incomplete 3. expensive computation

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



Neural Approach:

learn a dense and compact feature representation

Parser	UAS	LAS	sent. / s
MaltParser	89.8	87.2	469
MSTParser	91.4	88.1	10
TurboParser	92.3	89.6	8
C & M 2014	92.0	89.7	654

POS and dependency labels are represented as d-dimensional vectors.

extracting tokens&vector representations from configuration

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

	word	POS	dep.
s_1	good	JJ	\emptyset
s_2	has	VBZ	\emptyset
b_1	control	NN	\emptyset
$lc(s_1)$	\emptyset	\emptyset	\emptyset
$rc(s_1)$	\emptyset	\emptyset	\emptyset
$lc(s_2)$	He	PRP	nsubj
$rc(s_2)$	\emptyset	\emptyset	\emptyset

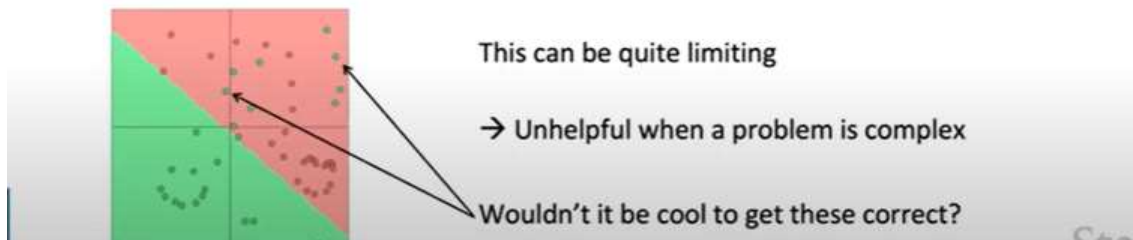
Second win: Deep Learning classifiers are non-linear classifiers

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

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

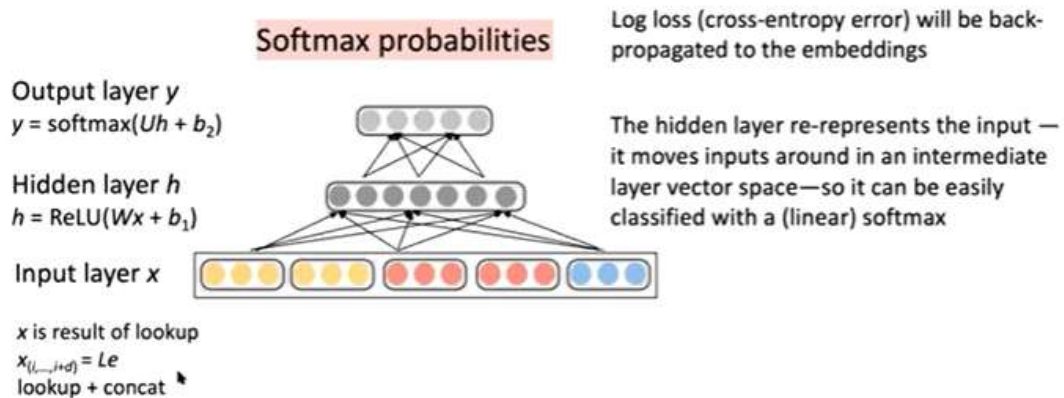
a.k.a. "cross entropy loss"

- 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**



Neural networks are more powerful: nonlinear decision boundaries (complex function)

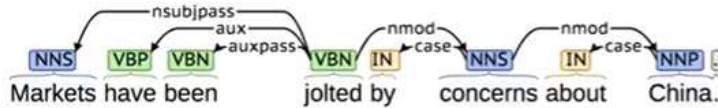
- simple feed-forward neural network multi-class classifier



-ReLU

Dependency parsing for sentence structure

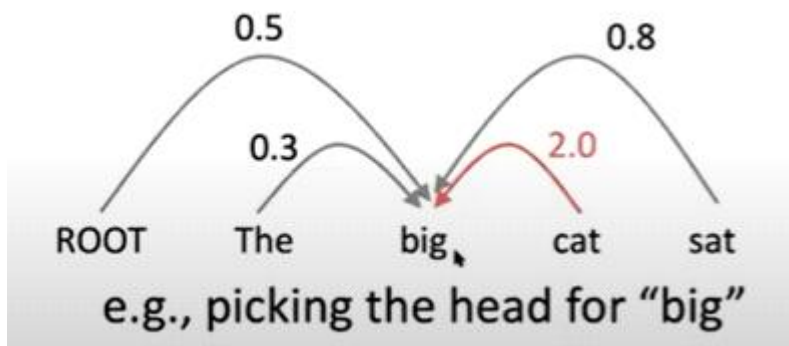
Chen and Manning (2014) showed that neural networks can accurately determine the structure of sentences, supporting meaning interpretation



It was the first simple, successful neural dependency parser

The dense representations (and non-linear classifier) let it outperform other greedy parsers in both accuracy and speed

- Dependency parsing for sentence structure
- Graph-based dependency parsers (eg. MST algorithm)



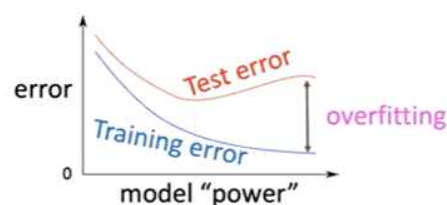
2. A bit more about neural networks

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.)



- 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 μ s per loop
- 10000 loops, best of 3: 53.8 μ s 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!
- The speed gain goes from 1 to 2 orders of magnitude with GPUs!

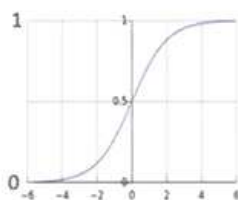
Sta

- Non-linearities, old and new

Non-linearities, old and new

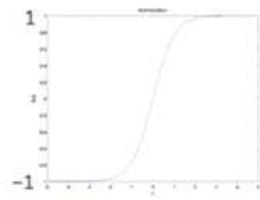
logistic ("sigmoid")

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



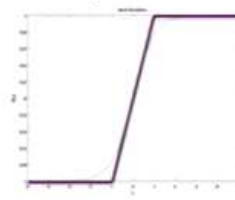
tanh

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



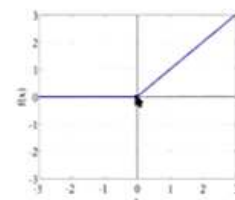
hard tanh

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



ReLU (Rectified Linear Unit)

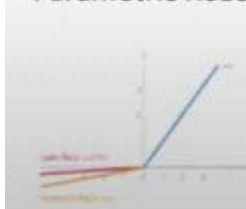
$$\text{rect}(z) = \max(z, 0)$$



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

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

Leaky ReLU /
Parametric ReLU



Swish [Ramachandran,
Zoph & Le 2017]



-parameter initialization

- You normally must initialize weights to small random values (i.e., not zero matrices!)
 - To avoid symmetries that prevent learning/specialization
- Initialize hidden layer biases to 0 and output (or reconstruction) biases to optimal value if weights were 0 (e.g., mean target or inverse sigmoid of mean target)
- Initialize **all other weights** $\sim \text{Uniform}(-r, r)$, with r chosen so numbers get neither too big or too small [later the need for this is removed with use of layer normalization]
- Xavier initialization has variance inversely proportional to fan-in n_{in} (previous layer size) and fan-out n_{out} (next layer size):

$$\text{Var}(W_i) = \frac{2}{n_{in} + n_{out}}$$

-Optimizers: 대부분 SGD 잘 작동(hand tuning learning rate 힘들)

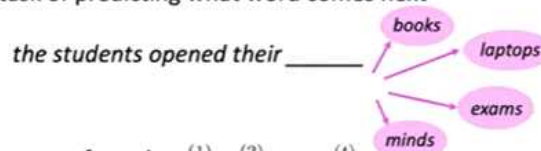
-Learning Rates

Learning Rates

- You can just use a constant learning rate. Start around $lr = 0.001$?
 - It must be order of magnitude right – try powers of 10
 - Too big: model may diverge or not converge
 - Too small: your model may not have trained by the assignment deadline
- Better results can generally be obtained by allowing learning rates to decrease as you train
 - By hand: halve the learning rate every k epochs
 - An epoch = a pass through the data (shuffled or sampled – not in same order each time)
 - By a formula: $lr = lr_0 e^{-kt}$, for epoch t
 - There are fancier methods like cyclic learning rates (q.v.)
- Fancier optimizers still use a learning rate but it may be an initial rate that the optimizer shrinks – so you may want to start with a higher learning rate

3. Language modeling+RNNs

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



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

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

where $x^{(t+1)}$ can be any word in the vocabulary $V = \{w_1, \dots, w_{|V|}\}$

What is language modeling? - is the task of predicting what word comes next/as a system that assigns probabilities to a piece of text

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

$$P(x^{(1)}, \dots, x^{(T)}) = P(x^{(1)}) \times P(x^{(2)} | x^{(1)}) \times \dots \times P(x^{(T)} | x^{(1)}, \dots, x^{(T-1)})$$

$$= \prod_{t=1}^T P(x^{(t)} | x^{(1)}, \dots, x^{(t-1)})$$

This is what our LM provides

- eg. 메시지 자동완성

- n-gram Language Models

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

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

prob of a n-gram $\rightarrow P(x^{(t+1)}, x^{(t)}, \dots, x^{(t-n+2)})$

prob of a (n-1)-gram $\rightarrow P(x^{(t)}, \dots, x^{(t-n+2)})$

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

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

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

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*.

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

Sparsity Problem 2

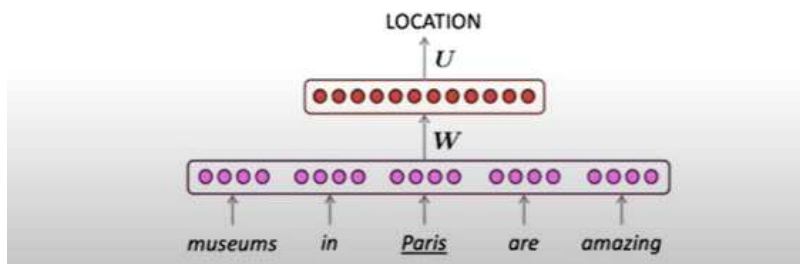
Problem: What if "students opened their" never occurred in data? Then we can't calculate probability for any w !

(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.

How to build a neural Language Model?

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



A fixed-window neural Language Model

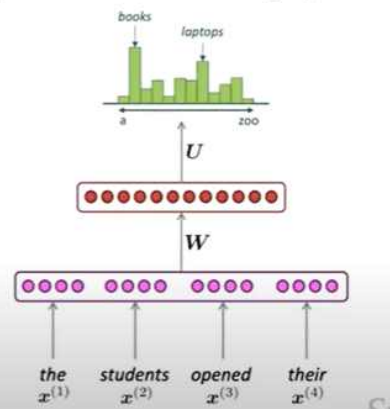
Approximately: Y. Bengio, et al. (2000/2003): A Neural Probabilistic Language Model

Improvements over n -gram LM:

- No sparsity problem
- Don't need to store all observed n -grams

Remaining **problems**:

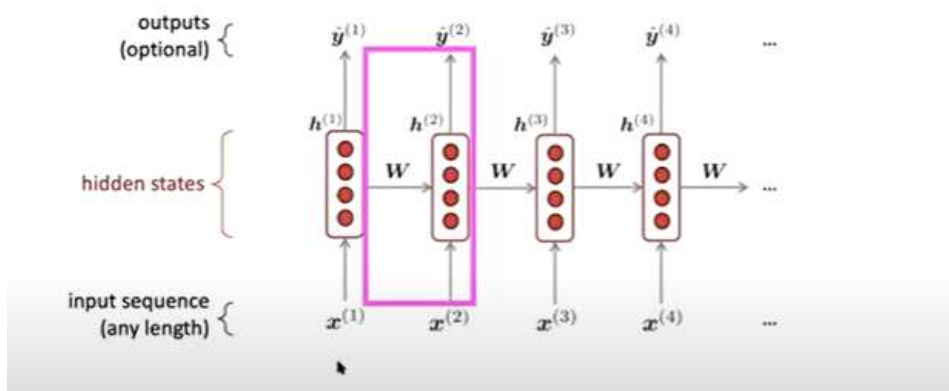
- Fixed window is **too small**
- Enlarging window enlarges W
- Window can never be large enough!
- $x^{(1)}$ and $x^{(2)}$ are multiplied by completely different weights in W .
- No symmetry** in how the inputs are processed.



Recurrent Neural Networks (RNN)

A family of neural architectures

Core idea: Apply the same weights W repeatedly



<A simple RNN Language Model>

A Simple RNN Language Model

$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$

output distribution

$$\hat{y}^{(t)} = \text{softmax}(U h^{(t)} + b_2) \in \mathbb{R}^{|V|}$$



hidden states

$$h^{(t)} = \sigma(W_h h^{(t-1)} + W_e e^{(t)} + b_1)$$

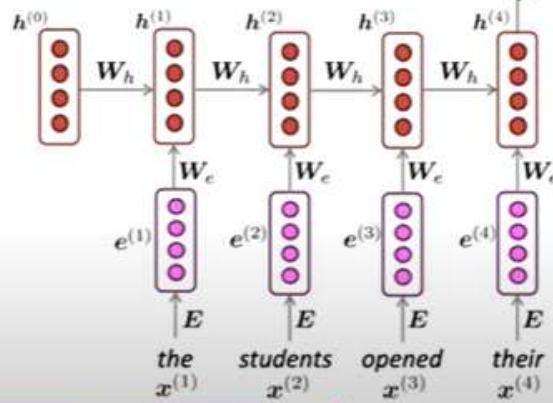
$h^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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



Note: this input sequence could be much longer now!

Sta