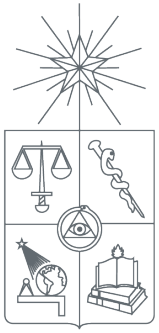
Deep Learning for Natural Language Processing

Felipe Bravo-Marquez

Department of Computer Science, University of Chile & IMFD

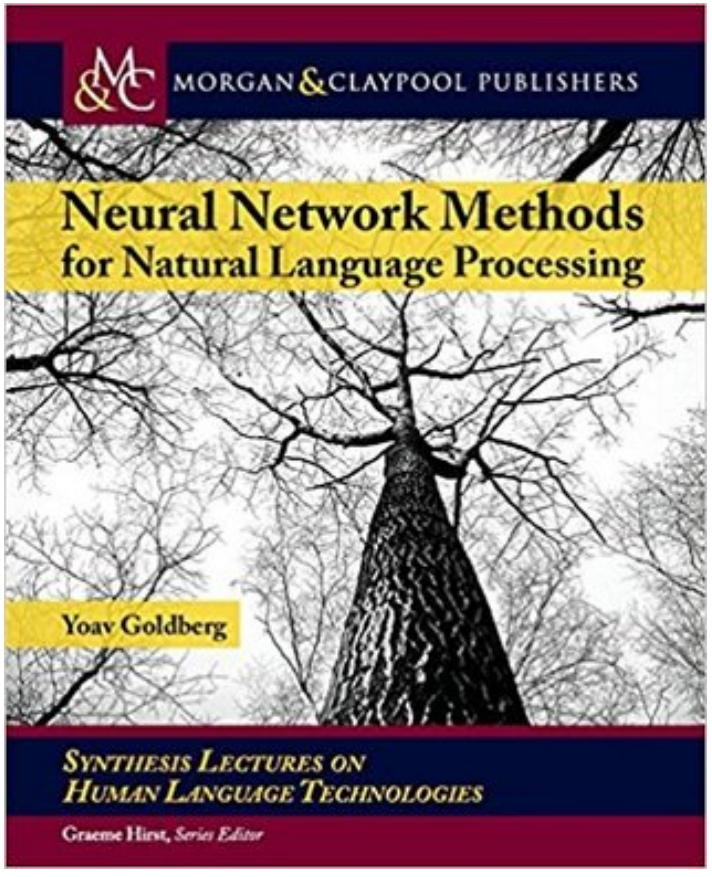
November 8, 2019



dcc CIENCIAS DE LA COMPUTACIÓN
UNIVERSIDAD DE CHILE

Disclaimer

This tutorial is heavily based on this book:



Natural Language Processing

- The amount of digitalized textual data being generated every day is huge (e.g, the Web, social media, medicar records, digitalized books).
- So does the need for translating, analyzing, and managing this flood of words and text.
- Natural language processing (NLP) is the field of designing methods and algorithms that take as input or produce as output unstructured, **natural language data**.

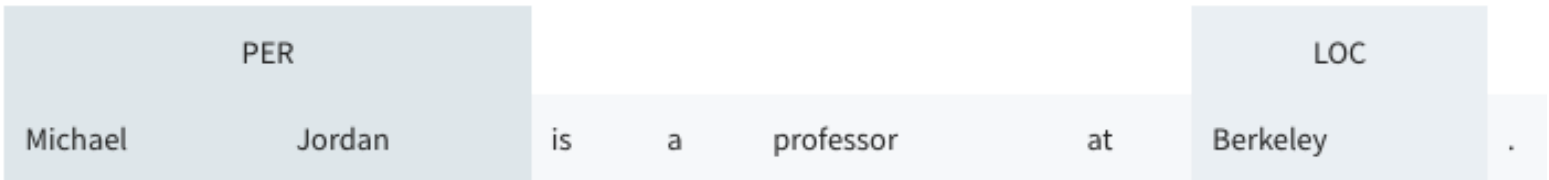


Figure: Example: Named Entity Recognition

- Human language is highly ambiguous: *I ate pizza with friends* vs. *I ate pizza with olives*.
- It is also ever changing and evolving (e.g, Hashtags in Twitter).

Natural Language Processing

- While we humans are great users of language, we are also very poor at formally understanding and describing the rules that govern language.
- Understanding and producing language using computers is highly challenging.
- The best known set of methods for dealing with language data rely on supervised machine learning.
- Supervised machine learning: attempt to infer usage patterns and regularities from a set of pre-annotated input and output pairs (a.k.a training dataset).

Training Dataset: CoNLL-2003 NER Data

Each line contains a token, a part-of-speech tag, a syntactic chunk tag, and a named-entity tag.

```
U.N.          NNP   I-NP   I-ORG
official      NN    I-NP   O
Ekeus         NNP   I-NP   I-PER
heads         VBZ   I-VP   O
for           IN    I-PP   O
Baghdad       NNP   I-NP   I-LOC
.             .     O      O
```

[1]Source:
https://www.clips.uantwerpen.be/conll2003/ner/

Challenges of Language

- Three challenging properties of language: discreteness , compositionality, and sparseness.
- **Discreteness**: we cannot infer the relation between two words from the letters they are made of (e.g., hamburger and pizza).
- **Compositionality**: the meaning of a sentence goes beyond the individual meaning of their words.
- **Sparseness**: The way in which words (discrete symbols) can be combined to form meanings is practically infinite.

Example of NLP Task: Topic Classification

- Classify a document into one of four categories: Sports, Politics, Gossip, and Economy.
- The words in the documents provide very strong hints.
- Which words provide what hints?
- Writing up rules for this task is rather challenging.
- However, readers can easily categorize a number of documents into its topic (data annotation).
- A supervised machine learning algorithm come up with the patterns of word usage that help categorize the documents.

Example 3: Sentiment Analysis

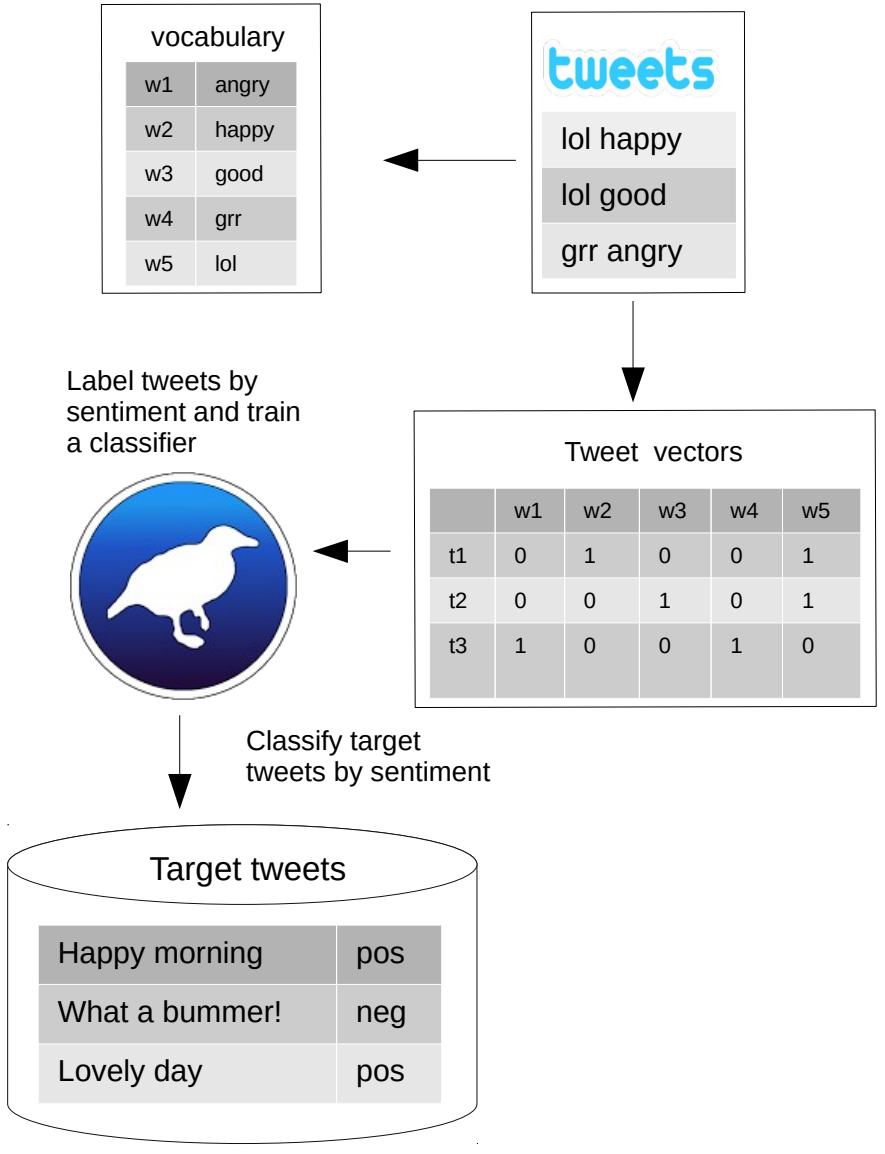
- Application of **NLP** techniques to identify and extract subjective information from textual datasets.

Main Problem: Message-level Polarity Classification (MPC)

1. Automatically classify a sentence to classes **positive**, **negative**, or **neutral**.

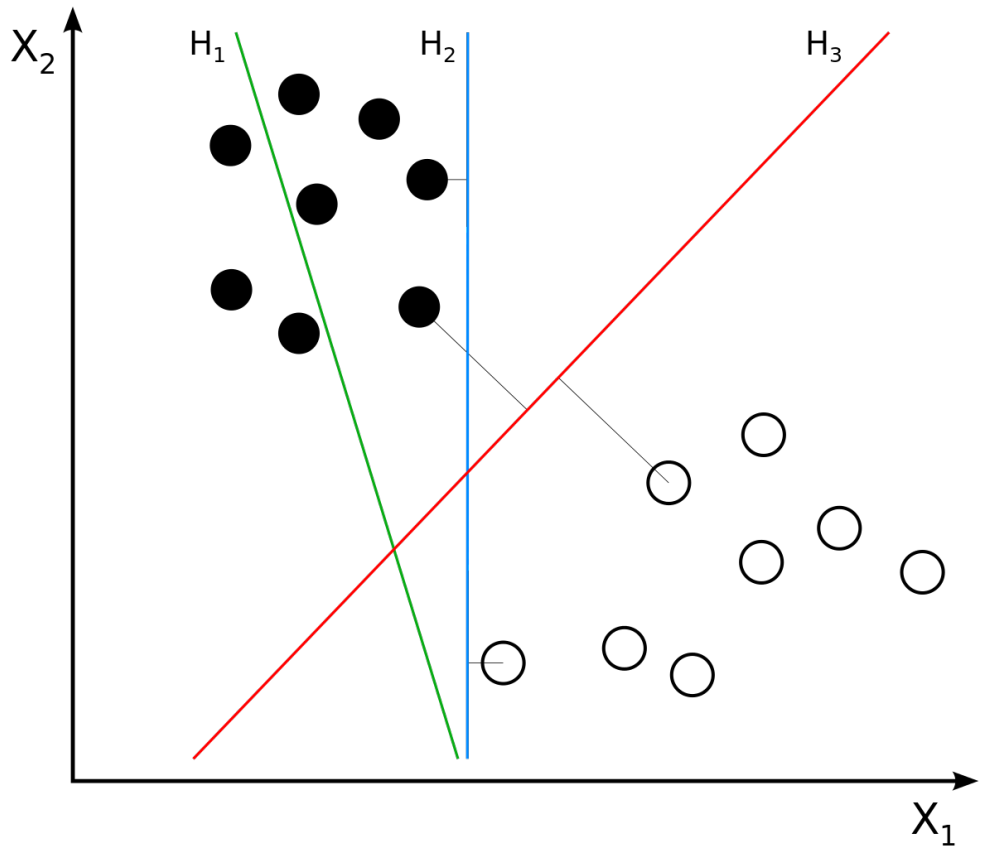
2. State-of-the-art solutions use **supervised** machine learning models trained from **manually** annotated examples [Mohammad et al., 2013].

Sentiment Classification via Supervised Learning and BoWs Vectors



Supervised Learning: Support Vector Machines (SVMs)

- Idea: Find a hyperplane that separates the classes with the maximum margin (largest separation).



- H_3 separates the classes with the maximum margin.

[2] Image source: Wikipedia

Challenges of NLP

- **Annotation Costs**: manual annotation is **labour-intensive** and **time-consuming**.
- **Domain Variations**: the pattern we want to learn can vary from one corpus to another (e.g., sports, politics).
- A model trained from data annotated for one domain will **not necessarily** work on another one!
- Trained models can become outdated over time (e.g., new hashtags).

Domain Variation in Sentiment

1. For me the queue was pretty **small** and it was only a 20 minute wait I think but was so worth it!!! :D @raynwise
2. Odd spatiality in Stuttgart. Hotel room is so **small** I can barely turn around but surroundings are inhumanly vast & long under construction.

Overcoming the data annotation costs

Distant Supervision

- Automatically **label** unlabeled data (**Twitter API**) using a heuristic method.
- **Emoticon-Annotation Approach (EAA)**: tweets with positive **:)** or negative **:(** emoticons are labelled according to the polarity indicated by the emoticon [Read, 2005].
- The emoticon is **removed** from the content.
- The same approach has been extended using hashtags #anger, and emojis.
- Is not trivial to find distant supervision techniques for all kind of NLP problems.

Crowdsourcing

- Rely on services like **Amazon Mechanical Turk** or **Crowdflower** to ask the **crowds** to annotate data.
- This can be expensive.
- It is hard to guarantee quality.

Sentiment Classification of Tweets

- In 2013, The Semantic Evaluation (SemEval) workshop organised the "Sentiment Analysis in Twitter task" [Nakov et al., 2013].
- The task was divided into two sub-tasks: the expression level and the message level.
- Expression-level: focused on determining the sentiment polarity of a message according to a marked entity within its content.
- Message-level: the polarity has to be determined according to the overall message.
- The organisers released training and testing datasets for both tasks. [Nakov et al., 2013]

The NRC System

- The team that achieved the highest performance in both tasks among 44 teams was the *NRC-Canada* team [Mohammad et al., 2013].
- The team proposed a supervised approach using a linear SVM classifier with the following hand-crafted features for representing tweets:
  1. Word *n*-grams.
  2. Character *n*-grams.
  3. Part-of-speech tags.
  4. Word clusters trained with the Brown clustering method [Brown et al., 1992].
  5. The number of elongated words (words with one character repeated more than two times).
  6. The number of words with all characters in uppercase.
  7. The presence of positive or negative emoticons.
  8. The number of individual negations.
  9. The number of contiguous sequences of dots, question marks and exclamation marks.
  10. Features derived from polarity lexicons [Mohammad et al., 2013]. Two of these lexicons were generated using the PMI method from tweets annotated with hashtags and emoticons.

Feature Engineering and Deep Learning

- Designing the features of a winning NLP system requires a lot of domain-specific knowledge.
- The NRC system was built before deep learning became popular in NLP.
- Deep Learning systems on the other hand rely on representation learning to automatically learn good representations.
- Large amounts of training data and faster multicore CPU/GPU machines are key in the success of deep learning.
- **Neural networks** and **word embeddings** play a key role in modern architectures for NLP.

Roadmap

In this course we will introduce modern concepts in natural language processing based on **neural networks.** The main concepts to be convered are listed below:

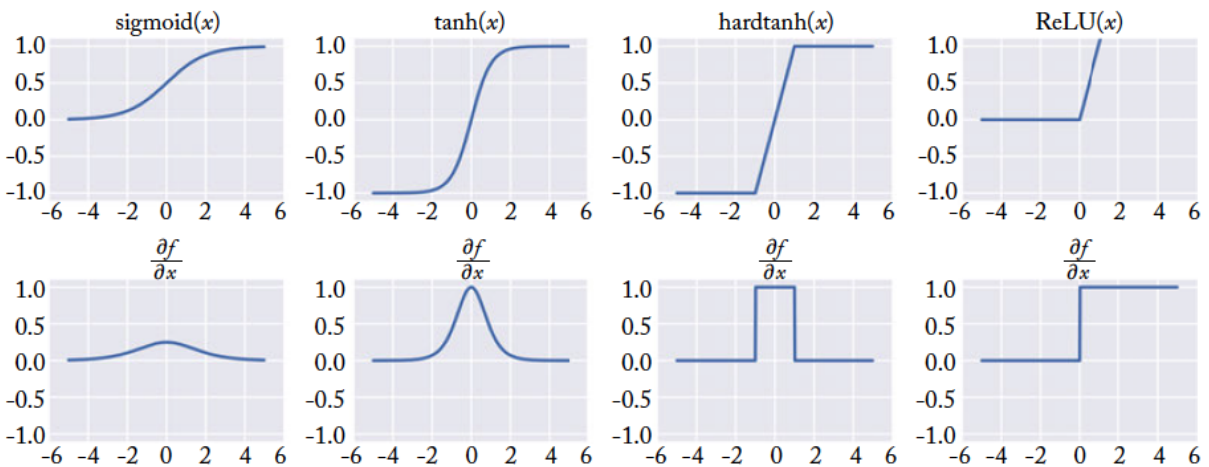
1. **Word embeddings**
2. **Convolutional Neural Networks** (CNNs)
3. **Recurrent Neural Networks**: Elman, LSTMs, GRUs.

Introduction to Neural Networks

- Very popular machine learning models formed by units called **neurons**.
- A neuron is a computational unit that has scalar inputs and outputs.
- Each input has an associated weight w.
- The neuron multiplies each input by its weight, and then sums them (other functions such as **max** are also possible).
- It applies an activation function g (usually non-linear) to the result, and passes it to its output.
- Multiple layers can be stacked.

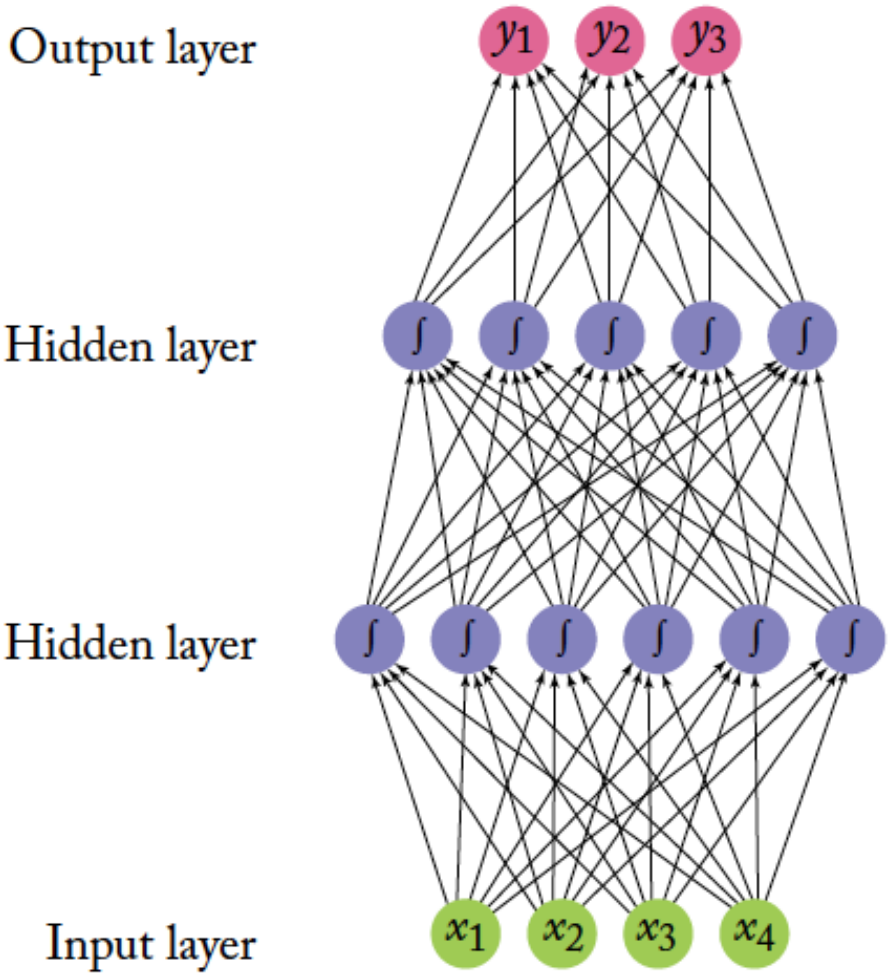
Activation Functions

- The nonlinear activation function *g* has a crucial role in the network's ability to represent complex functions.
- Without the nonlinearity in g, the neural network can only represent linear transformations of the input.



[2] Source:[Goldberg, 2016]

Feedforward Network with two Layers



[2] Source:[Goldberg, 2016]

Brief Introduction to Neural Networks

- The feedforward network from the picture is a stack of linear models separated by nonlinear functions.
- The values of each row of neurons in the network can be thought of as a vector.
- The input layer is a 4-dimensional vector ($\vec{x}$), and the layer above it is a 6-dimensional vector ($\vec{h}^1$).
- The fully connected layer can be thought of as a linear transformation from 4 dimensions to 6 dimensions.
- A fully connected layer implements a vector-matrix multiplication, $\vec{h} = \vec{x}W$.
- The weight of the connection from the i-th neuron in the input row to the j-th neuron in the output row is $W_{[i,j]}$.
- The values of $\vec{h}$ are transformed by a nonlinear function g that is applied to each value before being passed on as input to the next layer.

[2] Vectors are assumed to be row vectors and superscript indices correspond to network layers.

Brief Introduction to Neural Networks

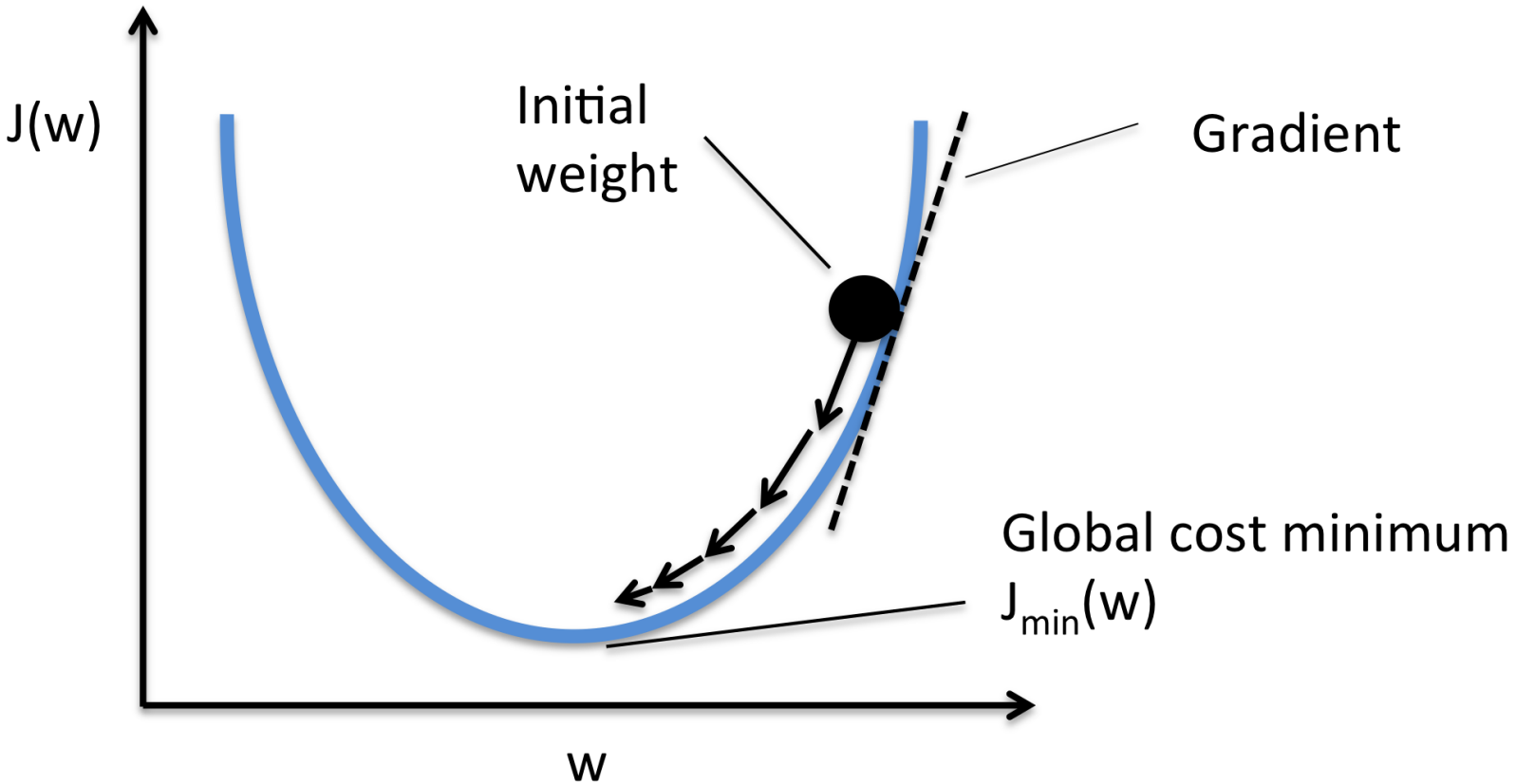
- The Multilayer Perceptron (MLP) from the figure can be written as the following mathematical function:

$$
\begin{aligned}
NN_{MLP2}(\vec{x}) &= \vec{y} \\
\vec{h}^1 &= g^1(\vec{x}W^1 + \vec{b}^1) \\
\vec{h}^2 &= g^2(\vec{h}^1 W^2 + \vec{b}^2) \\
\vec{y} &= \vec{h}^2 W^3 \\
\vec{y} &= (g^2(g^1(\vec{x}W^1 + \vec{b}^1)W^2 + \vec{b}^2))W^3.
\end{aligned} \tag{1}
$$

Network Training

- When training a neural network one defines a loss function $L(\hat{y}, y)$, stating the loss of predicting $\hat{y}$ when the true output is y.
- The training objective is then to minimize the loss across the different training examples.
- Networks are trained using gradient-based methods.
- They work by repeatedly computing an estimate of the loss L over the training set.
- They compute gradients of the parameters with respect to the loss estimate, and moving the parameters in the opposite directions of the gradient.
- Different optimization methods differ in how the error estimate is computed, and how moving in the opposite direction of the gradient is defined.

Gradient Descent



[2]Source: https://sebastianraschka.com/images/faq/closed-form-vs-gd/ball.png

Online Stochastic Gradient Descent

Algorithm 2.1 Online stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x}; \Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1}, \ldots, \boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1}, \ldots, \boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a training example $\boldsymbol{x_i}, \boldsymbol{y_i}$
3: Compute the loss $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$
4: $\hat{\boldsymbol{g}} \leftarrow$ gradients of $L(f(\boldsymbol{x_i}; \Theta), \boldsymbol{y_i})$ w.r.t Θ
5: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
6: **return** Θ

- The learning rate can either be fixed throughout the training process, or decay as a function of the time step t.
- The error calculated in line 3 is based on a single training example, and is thus just a rough estimate of the corpus-wide loss L that we are aiming to minimize.
- The noise in the loss computation may result in inaccurate gradients (single examples may provide noisy information).

[2] Source:[Goldberg, 2016]

Mini-batch Stochastic Gradient Descent

- A common way of reducing this noise is to estimate the error and the gradients based on a sample of *m* examples.
- This gives rise to the minibatch SGD algorithm

Algorithm 2.2 Minibatch stochastic gradient descent training.

Input:
- Function $f(\boldsymbol{x};\Theta)$ parameterized with parameters Θ.
- Training set of inputs $\boldsymbol{x_1},\ldots,\boldsymbol{x_n}$ and desired outputs $\boldsymbol{y_1},\ldots,\boldsymbol{y_n}$.
- Loss function L.

1: **while** stopping criteria not met **do**
2: Sample a minibatch of m examples $\{(\boldsymbol{x_1},\boldsymbol{y_1}),\ldots,(\boldsymbol{x_m},\boldsymbol{y_m})\}$
3: $\hat{\boldsymbol{g}} \leftarrow 0$
4: **for** $i = 1$ to m **do**
5: Compute the loss $L(f(\boldsymbol{x_i};\Theta),\boldsymbol{y_i})$
6: $\hat{\boldsymbol{g}} \leftarrow \hat{\boldsymbol{g}} +$ gradients of $\frac{1}{m}L(f(\boldsymbol{x_i};\Theta),\boldsymbol{y_i})$ w.r.t Θ
7: $\Theta \leftarrow \Theta - \eta_t \hat{\boldsymbol{g}}$
8: **return** Θ

- Higher values of *m* provide better estimates of the corpus-wide gradients, while smaller values allow more updates and in turn faster convergence.
- For modest sizes of *m* , some computing architectures (i.e., GPUs) allow an efficient parallel implementation of the computation in lines 3-6.

[2] Source:[Goldberg, 2016]

Some Loss Functions

- Hinge (or SVM loss): for binary classification problems, the classifier's output is a single scalar $\tilde{y}$ and the intended output y is in $\{+1, -1\}$. The classification rule is $\hat{y} = sign(\tilde{y})$, and a classification is considered correct if $y \cdot \tilde{y} > 0$.

$$L_{\text{hinge(binary)}}(\tilde{y}, y) = \max(0, 1 - y \cdot \tilde{y})$$

- Binary cross entropy (or logistic loss): is used in binary classification with conditional probability outputs. The classifier's output $\tilde{y}$ is transformed using the sigmoid function to the range $[0, 1]$, and is interpreted as the conditional probability $P(y = 1|x)$.

$$L_{\text{logistic}}(\hat{y}, y) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y})$$

Some Loss Functions

- Categorical cross-entropy loss: is used when a probabilistic interpretation of multi-class scores is desired. It measures the dissimilarity between the true label distribution y and the predicted label distribution $\tilde{y}$.

$$L_{\text{cross-entropy}}(\hat{y}, y) = -\sum_i y_{[i]} \log(\hat{y}_{[i]})$$

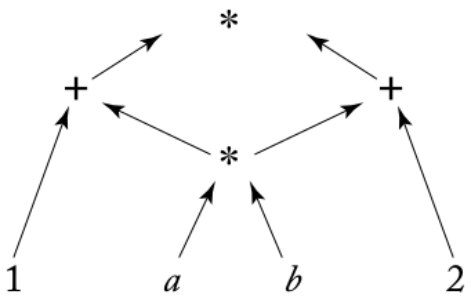
- The predicted label distribution of the categorical cross-entropy loss ($\hat{y}$) is obtained by applying the softmax function the last layer of the network $\tilde{y}$:

$$\hat{y}_{[i]} = \text{softmax}(\tilde{y})_{[i]} = \frac{e^{\tilde{y}_{[i]}}}{\sum_j e^{\tilde{y}_{[j]}}}$$

- The softmax function squashes the k-dimensional output to values in the range (0,1) with all entries adding up to 1. Hence, $\hat{y}_{[i]} = P(y = i|x)$ represent the class membership conditional distribution.

The Computation Graph Abstraction

- One can compute the gradients of the various parameters of a network by hand and implement them in code.
- This procedure is cumbersome and error prone.
- For most purposes, it is preferable to use automatic tools for gradient computation [Bengio, 2012].
- A computation graph is a representation of an arbitrary mathematical computation (e.g., a neural network) as a graph.
- Consider for example a graph for the computation of $(a * b + 1) * (a * b + 2)$:



- The computation of $a * b$ is shared.
- The graph structure defines the order of the computation in terms of the dependencies between the different components.

The Computation Graph Abstraction

- Te computation graph abstraction allows us to:
  1. Easily construct arbitrary networks.
  2. Evaluate their predictions for given inputs (forward pass)

Algorithm 5.3 Computation graph forward pass.

1: **for** i = 1 to N **do**
2: Let $a_1, \ldots, a_m = \pi^{-1}(i)$
3: $v(i) \leftarrow f_i(v(a_1), \ldots, v(a_m))$

  3. Compute gradients for their parameters with respect to arbitrary scalar losses (backward pass or backpropagation).

Algorithm 5.4 Computation graph backward pass (backpropagation).

1: $d(N) \leftarrow 1$ ▷ $\frac{\partial N}{\partial N} = 1$
2: **for** i = N-1 to 1 **do**
3: $d(i) \leftarrow \sum_{j \in \pi(i)} d(j) \cdot \frac{\partial f_j}{\partial i}$ ▷ $\frac{\partial N}{\partial i} = \sum_{j \in \pi(i)} \frac{\partial N}{\partial j} \frac{\partial j}{\partial i}$

- The backpropagation algorithm (backward pass) is essentially following the chain-rule of differentiation[3].

[3] A comprehensive tutorial on the backpropagation algorithm over the computational graph abstraction:
`https://colah.github.io/posts/2015-08-Backprop/`

Deep Learning Frameworks

Several software packages implement the computation-graph model. All these packages support all the essential components (node types) for defining a wide range of neural network architectures.

- TensorFlow (`https://www.tensorflow.org/`): an open source software library for numerical computation using data-flow graphs originally developed by the Google Brain Team.
- Keras: High-level neural network API that runs on top of Tensorflow as well as other backends (`https://keras.io/`).
- PyTorch: open source machine learning library for Python, based on Torch, developed by Facebook's artificial-intelligence research group. It supports dynamic graph construction, a different computation graph is created from scratch for each training sample. (`https://pytorch.org/`)

Word Vectors

- A major component in neural networks for language is the use of an embedding layer.
- A mapping of discrete symbols to continuous vectors.
- When embedding words, they transform from being isolated distinct symbols into mathematical objects that can be operated on.
- Distance between vectors can be equated to distance between words,
- This makes easier to generalize the behavior from one word to another.

Distributional Vectors

- **Distributional Hypothesis** [Harris, 1954]: words occurring in the same **contexts** tend to have similar meanings.
- Or equivalently: “a word is characterized by the **company** it keeps”.
- **Distributional representations**: words are represented by **high-dimensional vectors** based on the context’s where they occur.

Word-context Matrices

- Distributional vectors are built from word-context matrices M.
- Each cell (i, j) is a co-occurrence based association value between a **target word** w_i and a **context** c_j calculated from a corpus of documents.
- Contexts are commonly defined as windows of words surrounding w_i.
- The window length k is a parameter (between 1 and 8 words on both the left and the right sides of w_i).
- If the Vocabulary of the target words and context words is the same, M has dimensionality $|\mathcal{V}| \times |\mathcal{V}|$.
- Whereas shorter windows are likely to capture **syntactic information** (e.g, POS), longer windows are more likely to capture topical similarity [Goldberg, 2016, Jurafsky and Martin, 2008].

Distributional Vectors with context windows of size 1

Example 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

[3] Example taken from:
http://cs224d.stanford.edu/lectures/CS224d-Lecture2.pdf

Word-context Matrices

The associations between words and contexts can be calculated using different approaches:

1. Co-occurrence counts
2. Positive point-wise mutual information (PPMI)
3. The significance values of a paired t-test.

The most common of those according to [Jurafsky and Martin, 2008] is PPMI. Distributional methods are also referred to as count-based methods.

PPMI

- PPMI a filtered version of the traditional PMI measure in which negative values are set to zero:

$$\text{PPMI}(w, c) = \max(0, \text{PMI}(w, c)) \tag{2}$$

$$\text{PPMI}(w, c) = \max\left(0, \log_2\left(\frac{\text{count}(w, c) \times |D|}{\text{count}(w) \times \text{count}(c)}\right)\right). \tag{3}$$

- PMI calculates the log of the probability of word-context pairs occurring together over the probability of them being independent.
- Negative PMI values suggest that the pair co-occurs less often than chance.
- These estimates are unreliable unless the counts are calculated from very large corpora [Jurafsky and Martin, 2008].
- PPMI corrects this problem by replacing negative values by zero.

Distributed Vectors or Word embeddings

- Count-based distributional vectors increase in size with vocabulary i.e., can have a very high dimensionality.
- Explicitly storing the co-occurrence matrix can be memory-intensive.
- Some classification models don't scale well to high-dimensional data.
- The neural network community prefers using **distributed representations**[4] or **word embeddings**.
- Word **embeddings** are low-dimensional continuous dense word vectors trained from document corpora using **neural networks**.
- They have become a crucial component of Neural Network architectures for NLP.

[4] Idea: The meaning of the word is "distributed" over a combination of dimensions.

Distributed Vectors or Word embeddings (2)

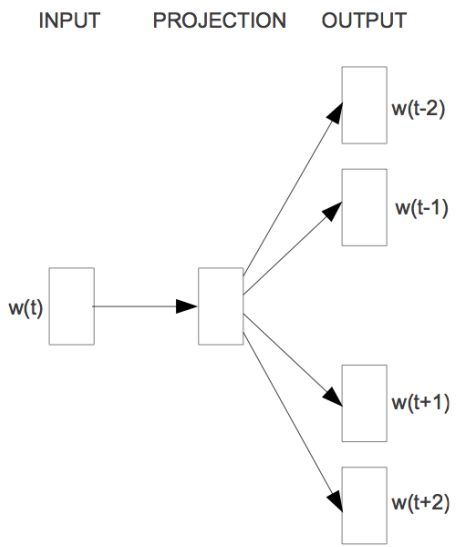
- They usually rely on an auxiliary predictive task (e.g., predict the following word).
- The dimensions are not directly interpretable i.e., represent latent features of the word, “hopefully capturing useful syntactic and semantic properties”[Turian et al., 2010].
- Most popular models are skip-gram [Mikolov et al., 2013], continuos bag-of-words [Mikolov et al., 2013], and Glove [Pennington et al., 2014].
- Word embeddings have shown to be more powerful than distributional approaches in many NLP tasks [Baroni et al., 2014].
- In [Amir et al., 2015], they were used as **features** in a regression model for determining the association between Twitter words and **positive sentiment**.

Word2Vec

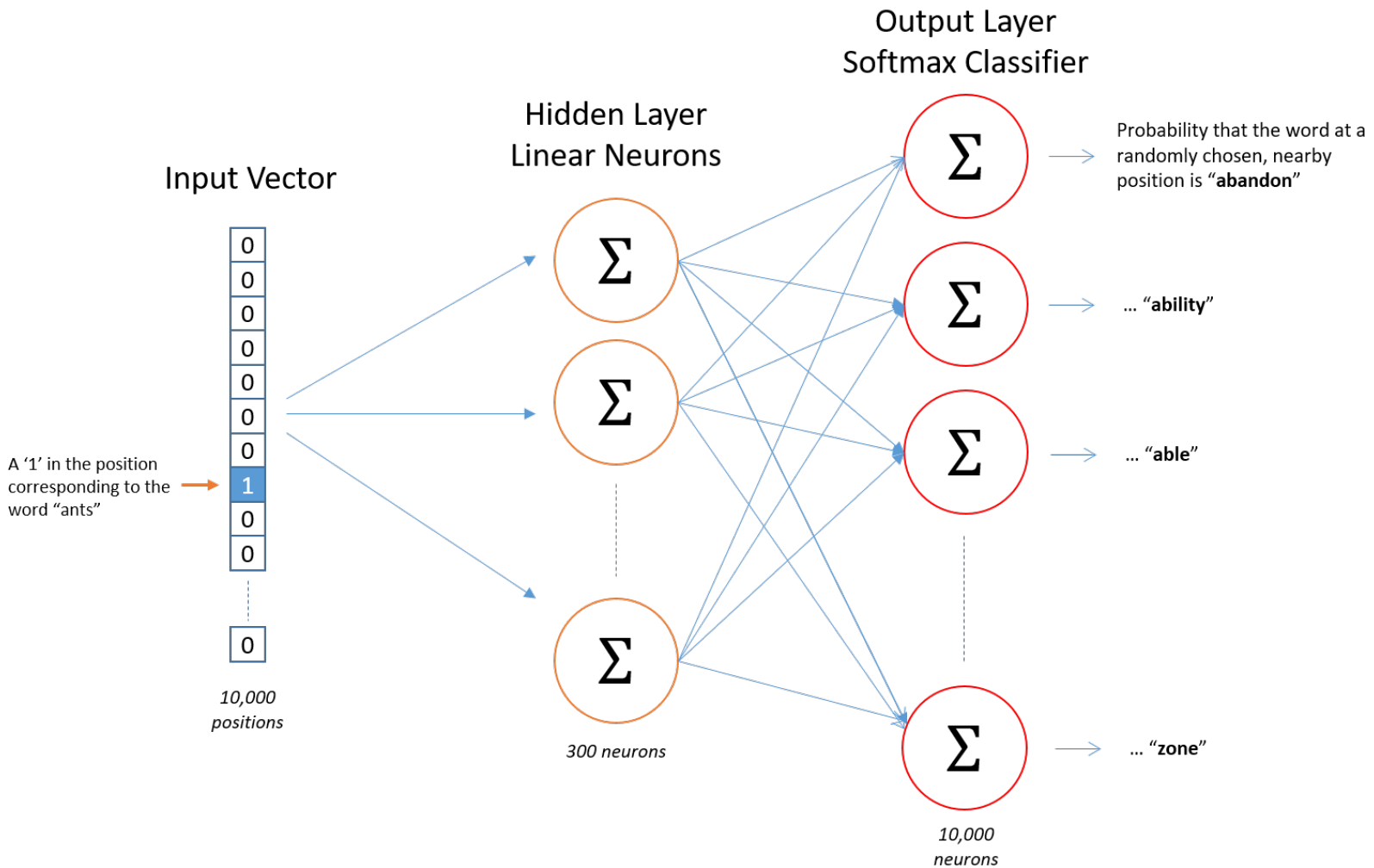
- Word2Vec is a software package that implements two neural network architectures for training word embeddings: Continuos Bag of Words (CBOW) and Skip-gram.
- It implements two optimization models: Negative Sampling and Hierarchical Softmax.
- These models are neural networks with one hidden layer that are trained to predict the contexts of words.

Skip-gram Model

- A neural network with one hidden layer is trained for predicting the words surrounding a center word, within a window of size k that is shifted along the input corpus.
- The center and surrounding k words correspond to the input and output layers of the network.
- Words are initially represented by 1-hot vectors: vectors of the size of the vocabulary ($|V|$) with zero values in all entries except for the corresponding word index that receives a value of 1.
- The output layer is formed by the concatenation of the k 1-hot vectors of the surrounding words.
- The hidden layer has a dimensionality d, which determines the size of the embeddings (normally $d \ll |V|$).



Skip-gram Model



[4] Picture taken from: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Parametrization of the Skip-gram model

- The conditional probability of the context word c given the center word w is modelled with a softmax (C is the set of all context words):

$$p(c|w) = \frac{e^{\vec{c}\cdot\vec{w}}}{\sum_{c'\in C} e^{\vec{c}'\cdot\vec{w}}}$$

- Model's parameters θ: $\vec{c}$ and $\vec{w}$ (vector representations of contexts and words).
- The optimization goal is to maximize the conditional likelihood of the contexts c:

$$arg\max_{\vec{c},\vec{w}} \sum_{(w,c)\in D} \log p(c|w) = \sum_{(w,c)\in D} (\log e^{\vec{c}\cdot\vec{w}} - \log \sum_{c'\in C} e^{\vec{c}'\cdot\vec{w}}) \quad (4)$$

- Assumption: maximising this function will result in good embeddings $\vec{w}$ i.e., similar words will have similar vectors.
- The term $p(c|w)$ is computationally expensive because of the summation $\sum_{c'\in C} e^{\vec{c}'\cdot\vec{w}}$ over all the contexts c'
- Fix: replace the softmax with a hierarchical softmax (the vocabulary is represented with a Huffman binary tree).
- Huffman trees assign short binary codes to frequent words, reducing the number of output units to be evaluated.

Skip-gram with Negative Sampling

- Negative-sampling (NS) is presented as a more efficient model for calculating skip-gram embeddings.
- However, it optimises a different objective function [Goldberg and Levy, 2014].
- Let D be the set of correct word-context pairs.
- NS maximizes the probability that a word-context pair a word-context pair (w, c) came from the input corpus D using a sigmoid function:

$$P(D = 1|w, c_i) = \frac{1}{1 + e^{-\vec{w} \cdot \vec{c_i}}}$$

- Assumption: the contexts words c_i are indepedent from each other:

$$P(D = 1|w, c_{1:k}) = \prod_{i=1}^{k} P(D = 1|w, c_i) = \prod_{i=1}^{k} \frac{1}{1 + e^{-\vec{w} \cdot \vec{c_i}}}$$

- This leads to the following target function (log-likelihood):

$$\arg\max_{\vec{c}, \vec{w}} \quad \log P(D = 1|w, c_{1:k}) = \sum_{i=1}^{k} \log \frac{1}{1 + e^{-\vec{w} \cdot \vec{c_i}}} \tag{5}$$

Skip-gram with Negative Sampling (2)

- This objective has a trivial solution if we set $\vec{w}$,$\vec{c}$ such that $p(D = 1|w, c) = 1$ for every pair (w, c) from D.
- This is achieved by setting $\vec{w} = \vec{c}$ and $\vec{w} \cdot \vec{c} = K$ for all $\vec{w}, \vec{c}$, where K is a large number.
- We need a mechanism that prevents all the vectors from having the same value, by disallowing some (w, c) combinations.
- One way to do so, is to present the model with some (w, c) pairs for which $p(D = 1|w, c)$ must be low, i.e. pairs which are not in the data.
- This is achieved sampling negative samples from $\tilde{D}$.

Skip-gram with Negative Sampling (3)

- Sample m words for each word-context pair $(w, c) \in D$.
- Add each sampled word w_i together with the original context c as a negative example to $\tilde{D}$.
- Final objective function:

$$\arg\max_{\vec{c},\vec{w}} \sum_{(w,c)\in D} \log P(D = 1|w, c_{1:k}) + \sum_{(w,c)\in \tilde{D}} \log P(D = 0|w, c_{1:k}) \quad (6)$$

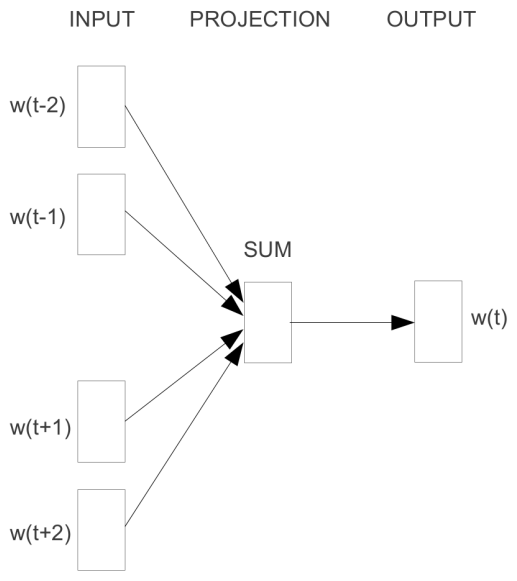
- The negative words are sampled from smoothed version of the corpus frequencies:

$$\frac{\#(w)^{0.75}}{\sum_{w'} \#(w')^{0.75}}$$

- This gives more relative weight to less frequent words.

Continuos Bag of Words: CBOW

- Similar to the skip-gram model but now the center word is predicted from the surrounding context.



CBOW

GloVe

- GloVe (from global vectors) is another popular method for training word embeddings [Pennington et al., 2014].
- It constructs an explicit word-context matrix, and trains the word and context vectors $\vec{w}$ and $\vec{c}$ attempting to satisfy:

$$w \cdot c + b_{[w]} + b_{[c]} = \log \#(w, c) \quad \forall (w, c) \in D \tag{7}$$

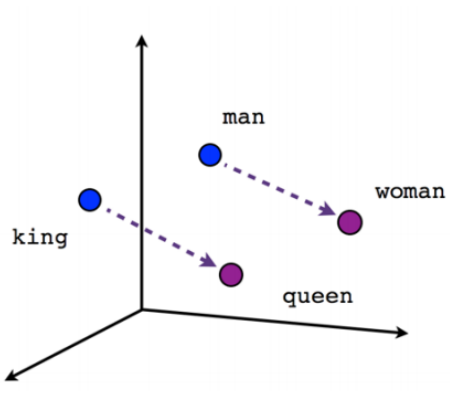
- where $b_{[w]}$ and $b_{[c]}$ are word-specific and context-specific trained biases.

GloVe (2)

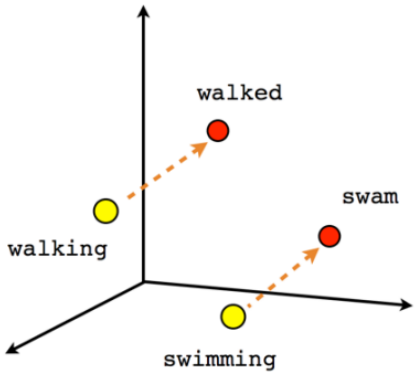
- In terms of matrix factorization, if we fix $b_{[w]} = \log \#(w)$ and $b_{[c]} = \log \#(c)$ we'll get an objective that is very similar to factorizing the word-context PMI matrix, shifted by $\log(|D|)$.
- In GloVe the bias parameters are learned and not fixed, giving it another degree of freedom.
- The optimization objective is weighted least-squares loss, assigning more weight to the correct reconstruction of frequent items.
- When using the same word and context vocabularies, the model suggests representing each word as the sum of its corresponding word and context embedding vectors.

Word Analogies

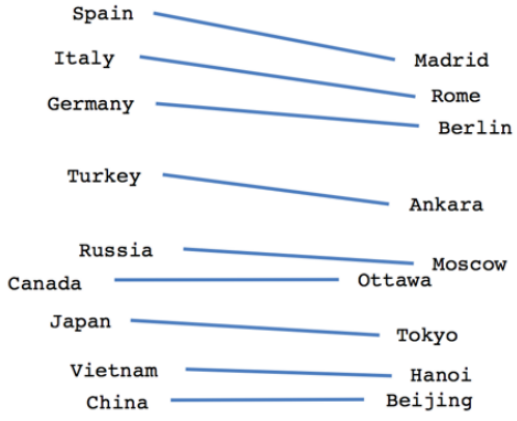
- Word embeddings can capture certain semantic relationships, e.g. male-female, verb tense and country-capital relationships between words.
- For example, the following relationship is found for word embeddings trained using Word2Vec: $\vec{w}_{king} - \vec{w}_{man} + \vec{w}_{woman} \approx \vec{w}_{queen}$.



Male-Female



Verb tense



Country-Capital

[5] Source: https://www.tensorflow.org/tutorials/word2vec

Correspondence between Distributed and Distributional Models

- Both the distributional “count-based” methods and the distributed “neural” ones are based on the distributional hypothesis.
- The both attempt to capture the similarity between words based on the similarity between the contexts in which they occur.
- Levy and Goldebrg showed in [Levy and Goldberg, 2014] that Skip-gram negative sampling (SGNS) is implicitly factorizing a word-context matrix, whose cells are the pointwise mutual information (PMI) of the respective word and context pairs, shifted by a global constant.
- This ties the neural methods and the traditional “count-based” suggesting that in a deep sense the two algorithmic families are equivalent.

FastText

- FastText embeedings extend the skipgram model to take into account the internal structure of words while learning word representations [Bojanowski et al., 2016].
- A vector representation is associated to each character *n*-gram.
- Words are represented as the sum of these representations.
- Taking the word *where* and $n = 3$, it will be represented by the character *n*-grams: $<$wh, whe, her, ere, re$>$, and the special sequence $<$where$>$.
- Note that the sequence $<$*her*$>$, corresponding to the word "her" is different from the tri-gram "her" form the word "here".
- FastText is useful for morphologically rich languages. For example, the words "amazing" and "amazingly" share information in FastText through their shared *n*-grams, whereas in Word2Vec these two words are completely unrelated.

FastText (2)

- Let $\mathcal{G}_w$ be the set of n-grams appearing in w.
- FastText associates a vector $\vec{g}$ to each n-gram in $\mathcal{G}_w$.
- In FastText the probability that a word-context pair (w, c) came from the input corpus D is calculated as follows:

$$P(D|w, c) = \frac{1}{1 + e^{-s(w,c)}}$$

where,

$$s(w, c) = \sum_{g \in G_w} \vec{g} \cdot \vec{c}.$$

- The negative sampling algorithm can be calculated in the same form as in the skip-gram model with this formulation.

Sentiment-Specific Phrase Embeddings

- Problem of word embeddings: antonyms can be used in similar contexts e.g., my car is nice vs my car is ugly.
- In [Tang et al., 2014] **sentiment-specific** word embeddings are proposed by combining the skip-gram model with emoticon-annotated tweets :) :(.
- These embeddings are used for **training** a word-level polarity classifier.
- The model integrates sentiment information into the continuous representation of phrases by developing a tailored neural architecture.
- Input: $\{w_i, s_j, pol_j\}$, where w_i is a phrase (or word), s_j the sentence, and pol_j the sentence's polarity.

Sentiment-Specific Phrase Embeddings (2)

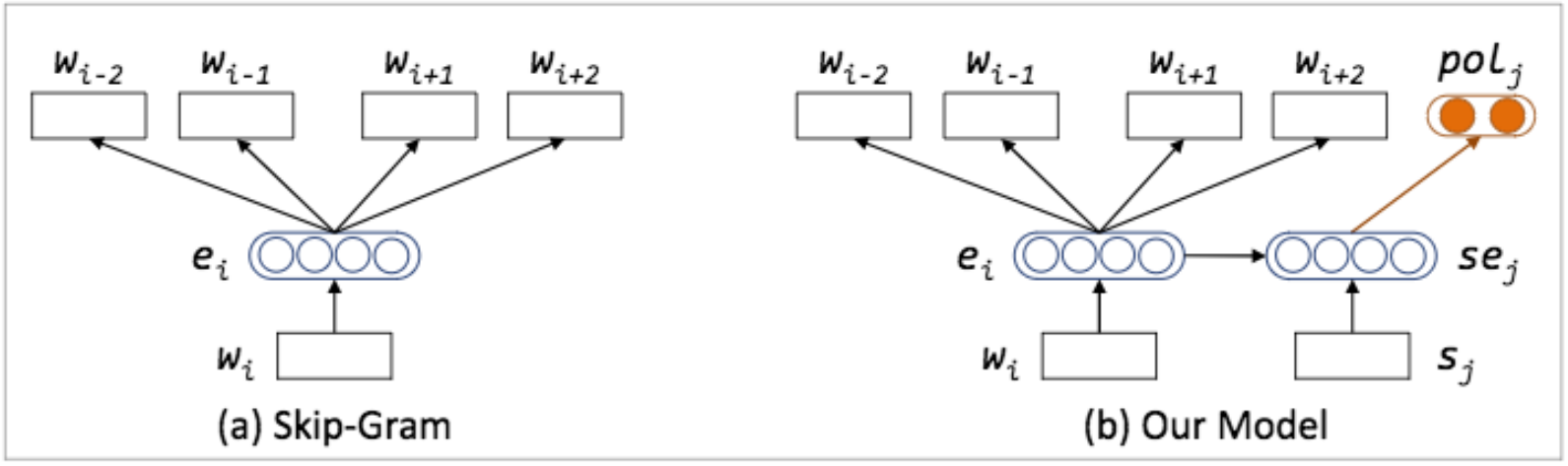
- The training objective uses the embedding of w_i to predict its context words (in the same way as the skip-gram model), and uses the sentence representation se_j to predict pol_j.
- Sentences (se_j) are represented by averaging the word vectors of their words.
- The objective of the sentiment part is to maximize the average of log sentiment probability:

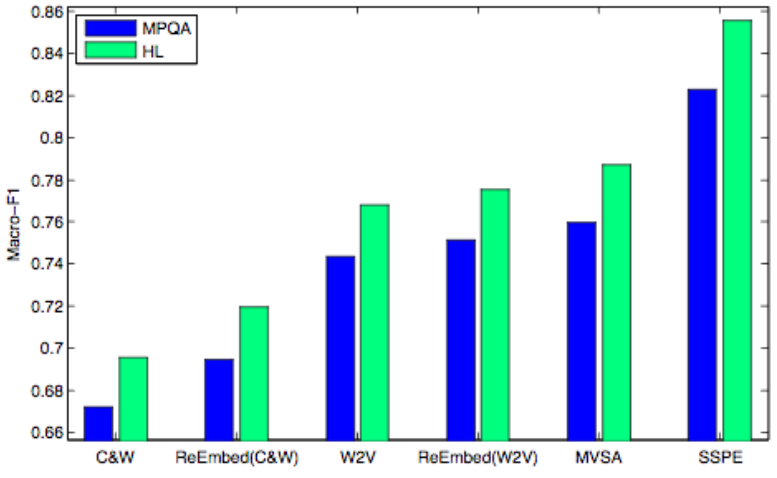
$$f_{sentiment} = \frac{1}{S} \sum_{j=1}^{S} \log p(pol_j | se_j)$$

- The final training objective is to maximize the linear combination of the skip-gram and sentiment objectives:

$$f = \alpha f_{skipgram} + (1 - \alpha) f_{sentiment}$$

Sentiment-Specific Phrase Embeddings





(b) Sentiment classification of lexicons with different embedding learning algrithms.

Gensim

Gensim is an open source Python library for natural language processing that implements many algorithms for training word embeddings.

- `https://radimrehurek.com/gensim/`
- `https://machinelearningmastery.com/develop-word-embeddings-python-gensim/`

Convolutional Neural Networks

- Convolutional neural networks (CNNs) became very popular in the computer vision community due to its success for detecting objects (“cat”,“bicycles”) regardless of its position in the image.
- They identify indicative local predictors in a structure (e.g., images, sentences).
- These predictors are combined to produce a fixed size vector representation for the structure.
- When used in NLP, the network captures the n-grams that are most informative for the target predictive task.
- For sentiment classification, these local aspects correspond to n-grams conveying sentiment (e.g., not bad, very good).
- The fundamental idea of CNNs [LeCun et al., 1998] is to consider feature extraction and classification as one jointly trained task.

Basic Convolution + Pooling

- Sentences are usually modelled as sequences of word embeddings.
- The CNN applies nonlinear (learned) functions or “filters” mapping windows of k words into scalar values.
- Several filters can be applied, resulting in an l-dimensional vector (one dimension per filter).
- The filters capture relevant properties of the words in the window.
- These filters correspond to the “convolution layer” of the network.

Basic Convolution + Pooling

- The “pooling” layer is used to combine the vectors resulting from the different windows into a single l-dimensional vector.
- This is done by taking the max or the average value observed in each of the dimensions over the different windows.
- The goal is to capture the most important “features” in the sentence, regardless of the position.
- The resulting l-dimensional vector is then fed further into a network that is used for prediction (e.g., softmax).
- The gradients are propagated back from the network’s loss tuning the parameters of the filter.
- The filters learn to highlight the aspects of the data (n-grams) that are important for the target task.

Basic Convolution + Pooling

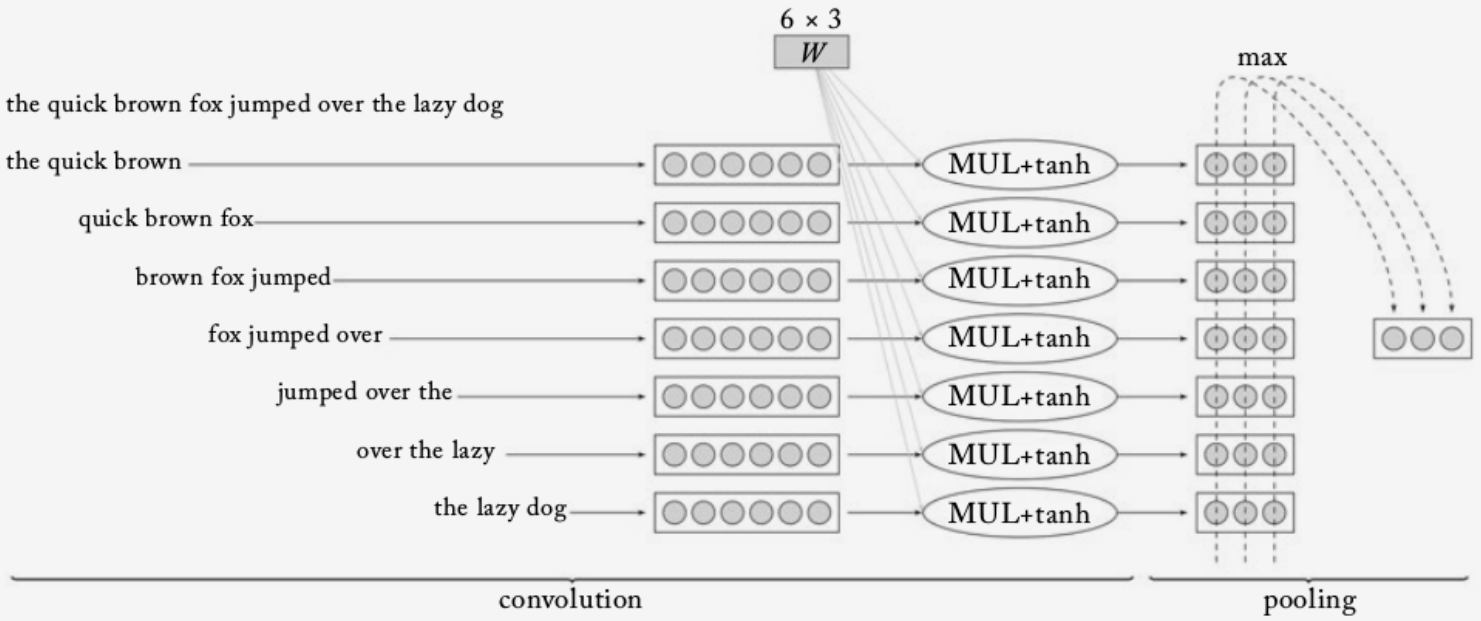


Figure 13.2: 1D convolution+pooling over the sentence "the quick brown fox jumped over the lazy dog." This is a narrow convolution (no padding is added to the sentence) with a window size of 3. Each word is translated to a 2-dim embedding vector (not shown). The embedding vectors are then concatenated, resulting in 6-dim window representations. Each of the seven windows is transfered through a 6×3 filter (linear transformation followed by element-wise tanh), resulting in seven 3-dimensional filtered representations. Then, a max-pooling operation is applied, taking the max over each dimension, resulting in a final 3-dimensional pooled vector.

[6]Source: [Goldberg, 2016]

1D Convolutions over Text

- We focus on the one-dimensional convolution operation[7].
- Consider a sequence of words $w_{1:n} = w_1, \ldots, w_n$ each with their corresponding d_{emb} dimensional word embedding $E_{[w_i]} = \vec{w}_i$.
- A 1D convolution of width k works by moving a sliding-window of size k over the sentence, and applying the same filter to each window in the sequence.
- A filter is a dot-product with a weight vector $\vec{u}$, which is often followed by a nonlinear activation function.

[7] 1D here refers to a convolution operating over 1-dimensional inputs such as sequences, as opposed to 2D convolutions which are applied to images.

1D Convolutions over Text

- Define the operator $\oplus(w_{i:i+k-1})$ to be the concatenation of the vectors $\vec{w}_i, \ldots, \vec{w}_{i+k-1}$.
- The concatenated vector of the i-th window is $\vec{x}_i = \oplus(w_{i:i+k-1}) = [\vec{w}_i; \vec{w}_{i+1}; \ldots; \vec{w}_{i+k-1}]$, $x_i \in \mathcal{R}^{k \cdot d_{emb}}$.
- We then apply the filter to each window vector resulting in scalar values $p_i = g(\vec{x}_i \cdot \vec{u})$. ($p_i \in \mathcal{R}$)
- It is customary to use l different filters, $\vec{u}_1, \ldots, \vec{u}_l$, which can be arranged into a matrix U, and a bias vector $\vec{b}$ is often added: $\vec{p}_i = g(\vec{x}_i \cdot U + \vec{b})$.
- Each vector $\vec{p}_i$ is a collection of l values that represent (or summarise) the i-th window ($\vec{p}_i \in \mathcal{R}^l$).
- Ideally, each dimension captures a different kind of indicative information.

Narrow vs. Wide Convolutions

- How many vectors $\vec{p}_i$ do we have?
- For a sentence of length n with a window of size k, there are $n - k + 1$ positions in which to start the sequence.
- We get $n - k + 1$ vectors $\vec{p}_{1:n-k+1}$.
- This approach is called **narrow convolution**.
- An alternative is to pad the sentence with $k - 1$ padding-words to each side, resulting in $n + k + 1$ vectors $\vec{p}_{1:n+k+1}$.
- This is called a **wide convolution**.

1D Convolutions over Text

- The main idea behind the convolution layer is to apply the same parameterised function over all k-grams in the sequence.
- This creates a sequence of m vectors, each representing a particular k-gram in the sequence.
- The representation is sensitive to the identity and order of the words within a k-gram.
- However, the same representation will be extracted for a k-gram regardless of its position within the sequence.

Vector Pooling

- Applying the convolution over the text results in m vectors $\vec{p}_{1:m}$, each $\vec{p}_i \in \mathcal{R}^l$.
- These vectors are then combined (pooled) into a single vector $c \in \mathcal{R}^l$ representing the entire sequence.
- Max pooling: this operator takes the maximum value across each dimension (most common pooling operation).

$$c_{[j]} = \max_{1<i\leq m} p_{i[j]} \quad \forall j \in [1, l]$$

- Average Pooling (second most common): takes the average value of each index:

$$c = \frac{1}{m} \sum_{i=1}^{m} p_i$$

- Ideally, the vector c will capture the essence of the important information in the sequence.

Vector Pooling

- The nature of the important information that needs to be encoded in the vector c is task dependent.
- If we are performing sentiment classification, the essence are informative ngrams that indicate sentiment.
- If we are performing topic-classification, the essence are informative n-grams that indicate a particular topic.
- During training, the vector c is fed into downstream network layers (i.e., an MLP), culminating in an output layer which is used for prediction.
- The training procedure of the network calculates the loss with respect to the prediction task, and the error gradients are propagated all the way back through the pooling and convolution layers, as well as the embedding layers.
- The training process tunes the convolution matrix U, the bias vector $\vec{b}$, the downstream network, and potentially also the embeddings matrix E[8] such that the vector c resulting from the convolution and pooling process indeed encodes information relevant to the task at hand.

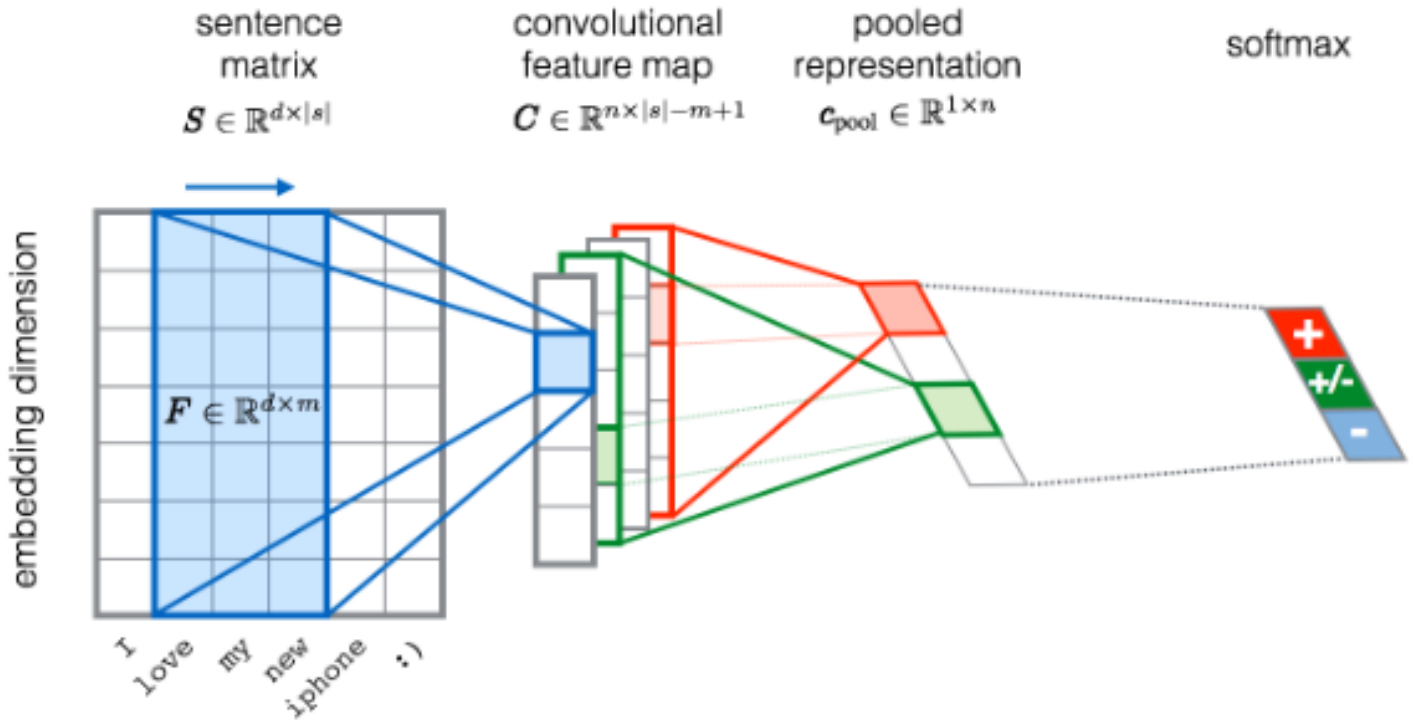
[8] While some people leave the embedding layer fixed during training, others allow the parameters to change.

Twitter Sentiment Classification with CNN

- A convolutional neural network architecture for Twitter sentiment classification is developed in [Severyn and Moschitti, 2015].
- Each tweet is represented as a matrix whose columns correspond to the words in the tweet, preserving the order in which they occur.
- The words are represented by dense vectors or embeddings trained from a large corpus of unlabelled tweets using word2vec.
- The network is formed by the following layers: an input layer with the given tweet matrix, a single convolutional layer, a rectified linear activation function, a max pooling layer, and a soft-max classification layer.

Twitter Sentiment Classification with CNN

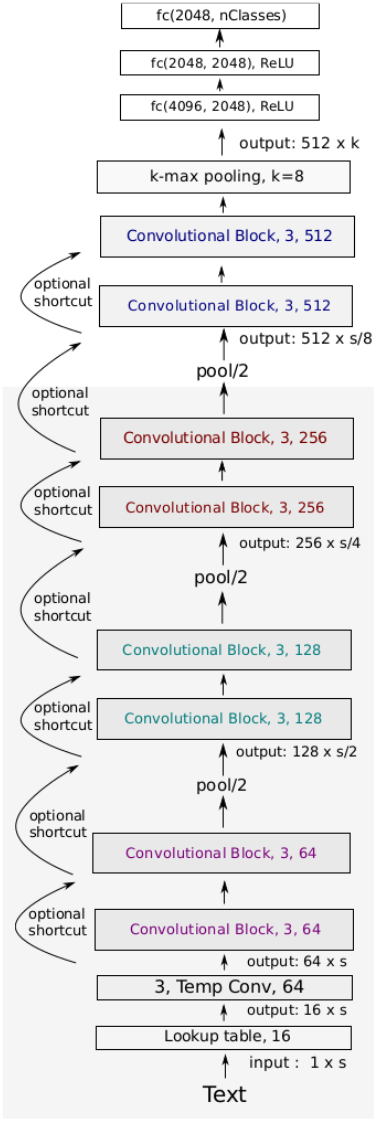
- The weights of the neural network are pre-trained using emoticon-annotated data, and then trained with the hand-annotated tweets from the SemEval competition.
- Experimental results show that the pre-training phase allows for a proper initialisation of the network's weights, and hence, has a positive impact on classification accuracy.



Very Deep Convolutional Networks for Text Classification

- CNNs architectures for NLP are rather shallow in comparison to the deep convolutional networks which have pushed the state-of-the-art in computer vision.
- A new architecture (VDCNN) for text processing which operates directly at the character level and uses only small convolutions and pooling operations is proposed in [Conneau et al., 2017].
- Character-level embeddings are used instead of word-embeddings.
- Characters are the lowest atomic representation of text.
- The performance of this model increases with depth: using up to 29 convolutional layers, authors report improvements over the state-of-the-art on several public text classification tasks.
- Most notorious improvements are achieved on large datasets.
- First paper showing the the benefits of depth architectures for NLP.

Very Deep Convolutional Networks for Text Classification



Recurrent Neural Networks

- While representations derived from convolutional networks offer some sensitivity to word order, their order sensitivity is restricted to mostly local patterns, and disregards the order of patterns that are far apart in the sequence.
- Recurrent neural networks (RNNs) allow representing arbitrarily sized sequential inputs in fixed-size vectors, while paying attention to the structured properties of the inputs [Goldberg, 2016].
- RNNs, particularly ones with gated architectures such as the LSTM and the GRU, are very powerful at capturing statistical regularities in sequential inputs.

The RNN Abstraction

- We use $\vec{x}_{i:j}$ to denote the sequence of vectors $\vec{x}_i, \ldots, \vec{x}_j$.
- On a high-level, the RNN is a function that takes as input an arbitrary length ordered sequence of n d_{in}-dimensional vectors $\vec{x}_{i:n} = \vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n$ ($\vec{x}_i \in \mathcal{R}^{d_{in}}$) and returns as output a single d_{out} dimensional vector $\vec{y}_n \in \mathcal{R}^{d_{out}}$:

$$\begin{aligned} \vec{y}_n &= RNN(\vec{x}_{1:n}) \\ \vec{x}_i \in \mathcal{R}^{d_{in}} \quad \vec{y}_n &\in \mathcal{R}^{d_{out}} \end{aligned} \tag{8}$$

- This implicitly defines an output vector $\vec{y}_i$ for each prefix $\vec{x}_{1:i}$ of the sequence $\vec{x}_{i:n}$.
- We denote by RNN^* the function returning this sequence:

$$\begin{aligned} \vec{y}_{1:n} &= RNN^*(\vec{x}_{1:n}) \\ \vec{y}_i &= RNN(\vec{x}_{1:i}) \\ \vec{x}_i \in \mathcal{R}^{d_{in}} \quad \vec{y}_n &\in \mathcal{R}^{d_{out}} \end{aligned} \tag{9}$$

The RNN Abstraction

- The output vector $\vec{y}_n$ is then used for further prediction.
- For example, a model for predicting the conditional probability of an event e given the sequence $\vec{x}_{1:n}$ can be defined as the j-th element in the output vector resulting from the softmax operation over a linear transformation of the RNN encoding:

$$p(e = j|\vec{x}_{1:n}) = \text{softmax}(RNN(\vec{x}_{1:n}) \cdot W + b)_{[j]}$$

- The RNN function provides a framework for conditioning on the entire history without resorting to the Markov assumption which is traditionally used for modeling sequences.

The RNN Abstraction

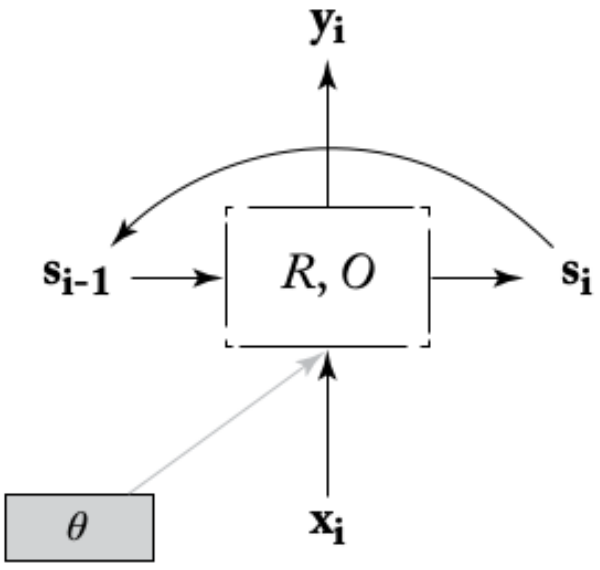
- The RNN is defined recursively, by means of a function R taking as input a state vector $\vec{s}_{i-1}$ and an input vector $\vec{x}_i$ and returning a new state vector $\vec{s}_i$.
- The state vector $\vec{s}_i$ is then mapped to an output vector $\vec{y}_i$ using a simple deterministic function $O(\cdot)$.
- The base of the recursion is an initial state vector, $\vec{s}_0$, which is also an input to the RNN.
- For brevity, we often omit the initial vector s_0 , or assume it is the zero vector.
- When constructing an RNN, much like when constructing a feed-forward network, one has to specify the dimension of the inputs $\vec{x}_i$ as well as the dimensions of the outputs $\vec{y}_i$.

The RNN Abstraction

$$
\begin{aligned}
RNN^*(\vec{x}_{1:n}; \vec{s}_0) &= \vec{y}_{1:n} \\
\vec{y}_i &= O(\vec{s}_i) \\
\vec{s}_i &= R(\vec{s}_{i-1}, \vec{x}_i) \\
\vec{x}_i \in \mathcal{R}^{d_{in}}, \quad \vec{y}_i \in \mathcal{R}^{d_{out}}, \quad \vec{s}_i &\in \mathcal{R}^{f(d_{out})}
\end{aligned}
\tag{10}
$$

- The functions R and O are the same across the sequence positions.
- The RNN keeps track of the states of computation through the state vector $\vec{s}_i$ that is kept and being passed across invocations of R.

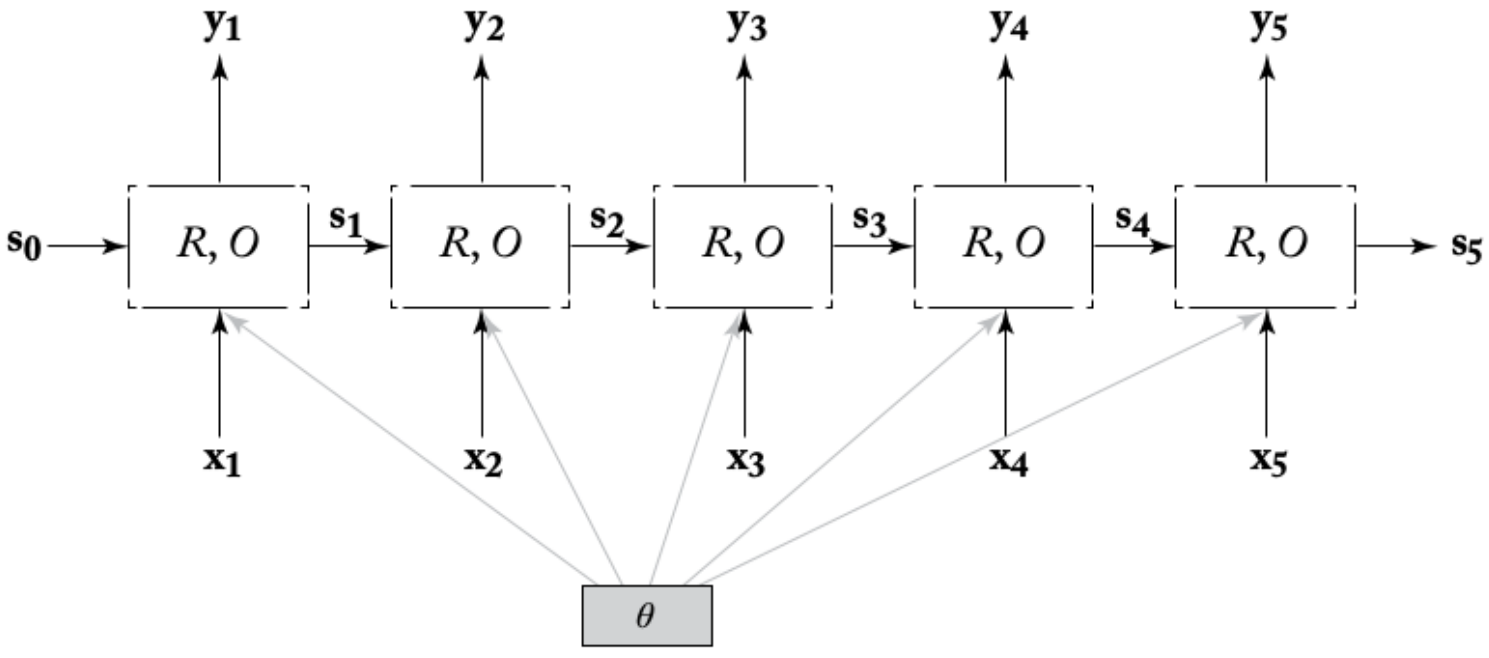
The RNN Abstraction



- This presentation follows the recursive definition, and is correct for arbitrarily long sequences.

The RNN Abstraction

- For a finite sized input sequence (and all input sequences we deal with are finite) one can unroll the recursion.



- The parameters θ highlight the fact that the same parameters are shared across all time steps.
- Different instantiations of R and O will result in different network structures.

The RNN Abstraction

- We note that the value of $\vec{s}_i$ (and hence $\vec{y}_i$) is based on the entire input $\vec{x}_1, \ldots, \vec{x}_i$.
- For example, by expanding the recursion for $i = 4$ we get:

$$
\begin{aligned}
s_4 &= R(s_3, x_4) \\
&= R(\overbrace{R(s_2, x_3)}^{s_3}, x_4) \\
&= R(R(\overbrace{R(s_1, x_2)}^{s_2}, x_3), x_4) \\
&= R(R(R(\overbrace{R(s_0, x_1)}^{s_1}, x_2), x_3), x_4).
\end{aligned}
$$

- Thus, $\vec{s}_n$ and $\vec{y}_n$ can be thought of as encoding the entire input sequence.
- The job of the network training is to set the parameters of R and O such that the state conveys useful information for the task we are tying to solve.

RNN Training

- An unrolled RNN is just a very deep neural network.
- The same parameters are shared across many parts of the computation.
- Additional input is added at various layers.
- To train an RNN network, need to create the unrolled computation graph for a given input sequence, add a loss node to the unrolled graph.
- Then use the backward (backpropagation) algorithm to compute the gradients with respect to that loss.

RNN Training

- This procedure is referred to in the RNN literature as backpropagation through time (BPTT).
- The RNN does not do much on its own, but serves as a trainable component in a larger network.
- The final prediction and loss computation are performed by that larger network, and the error is back-propagated through the RNN.
- This way, the RNN learns to encode properties of the input sequences that are useful for the further prediction task.
- The supervision signal is not applied to the RNN directly, but through the larger network.

Bidirectional RNNS (BIRNN)

- A useful elaboration of an RNN is a bidirectional-RNN (also commonly referred to as biRNN)
- Consider the task of sequence tagging over a sentence.
- An RNN allows us to compute a function of the i-th word x_i based on the past words $x_{1:i}$ up to and including it.
- However, the following words $x_{i+1:n}$ may also be useful for prediction,
- The biRNN allows us to look arbitrarily far at both the past and the future within the sequence.

Bidirectional RNNS (BIRNN)

- Consider an input sequence $\vec{x}_{1:n}$.
- The biRNN works by maintaining two separate states, s_i^f and s_i^b for each input position i.
- The forward state s_i^f is based on $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_i$, while the backward state s_i^b is based on $\vec{x}_n, \vec{x}_{n-1}, \ldots, \vec{x}_i$.
- The forward and backward states are generated by two different RNNs.
- The first RNN(R^f, O^f) is fed the input sequence $\vec{x}_{1:n}$ as is, while the second RNN(R^b, O^b) is fed the input sequence in reverse.
- The state representation $\vec{s}_i$ is then composed of both the forward and backward states.

Bidirectional RNNS (BIRNN)

- The output at position i is based on the concatenation of the two output vectors:

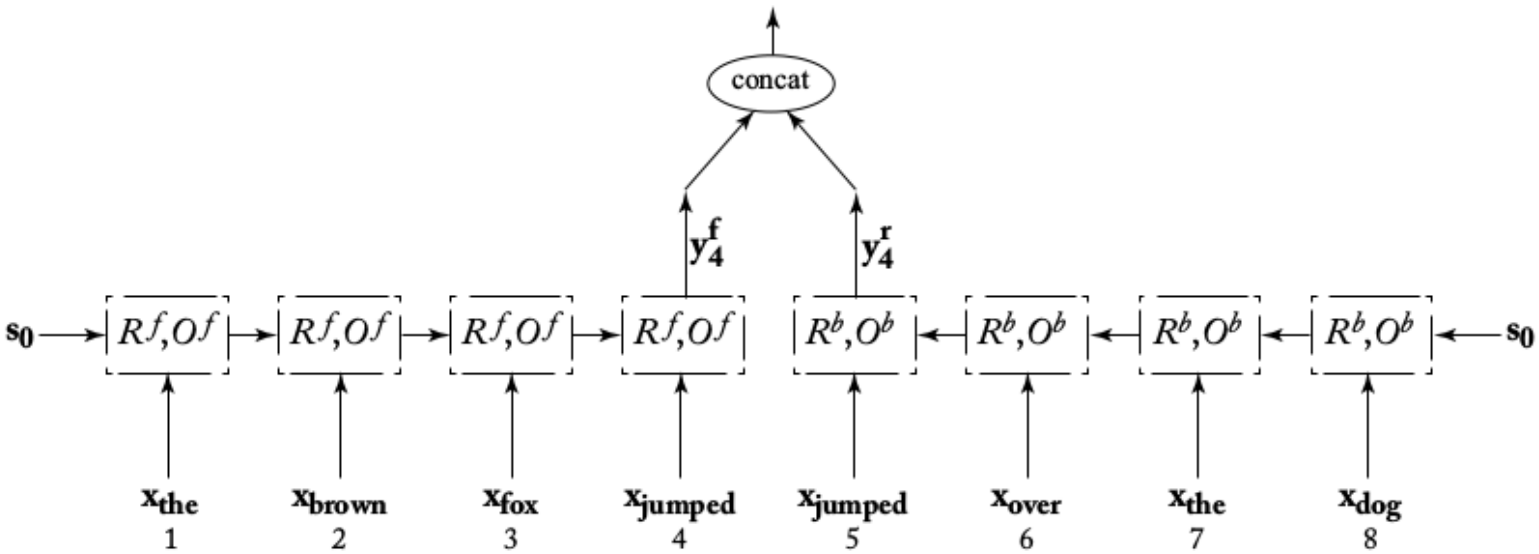
$$\vec{y}_i = [\vec{y}_i^f; \vec{y}_i^b] = [O^f(s_i^f); O^b(s_i^b)]$$

- The output takes into account both the past and the future.
- The biRNN encoding of the ith word in a sequence is the concatenation of two RNNs, one reading the sequence from the beginning, and the other reading it from the end.
- We define $biRNN(\vec{x}_{1:n}, i)$ to be the output vector corresponding to the ith sequence position:

$$biRNN(\vec{x}_{1:n}, i) = \vec{y}_i = [RNN^f(\vec{x}_{1:i}); RNN^b(\vec{x}_{n:i})$$

Bidirectional RNNS (BIRNN)

- The vector $\vec{y}_i$ can then be used directly for prediction, or fed as part of the input to a more complex network.
- While the two RNNs are run independently of each other, the error gradients at position i will flow both forward and backward through the two RNNs.
- Feeding the vector $\vec{y}_i$ through an MLP prior to prediction will further mix the forward and backward signals.



- Note how the vector $\vec{y}_4$, corresponding to the word **jumped**, encodes an infinite window around (and including) the focus vector $\vec{x}_{jumped}$.

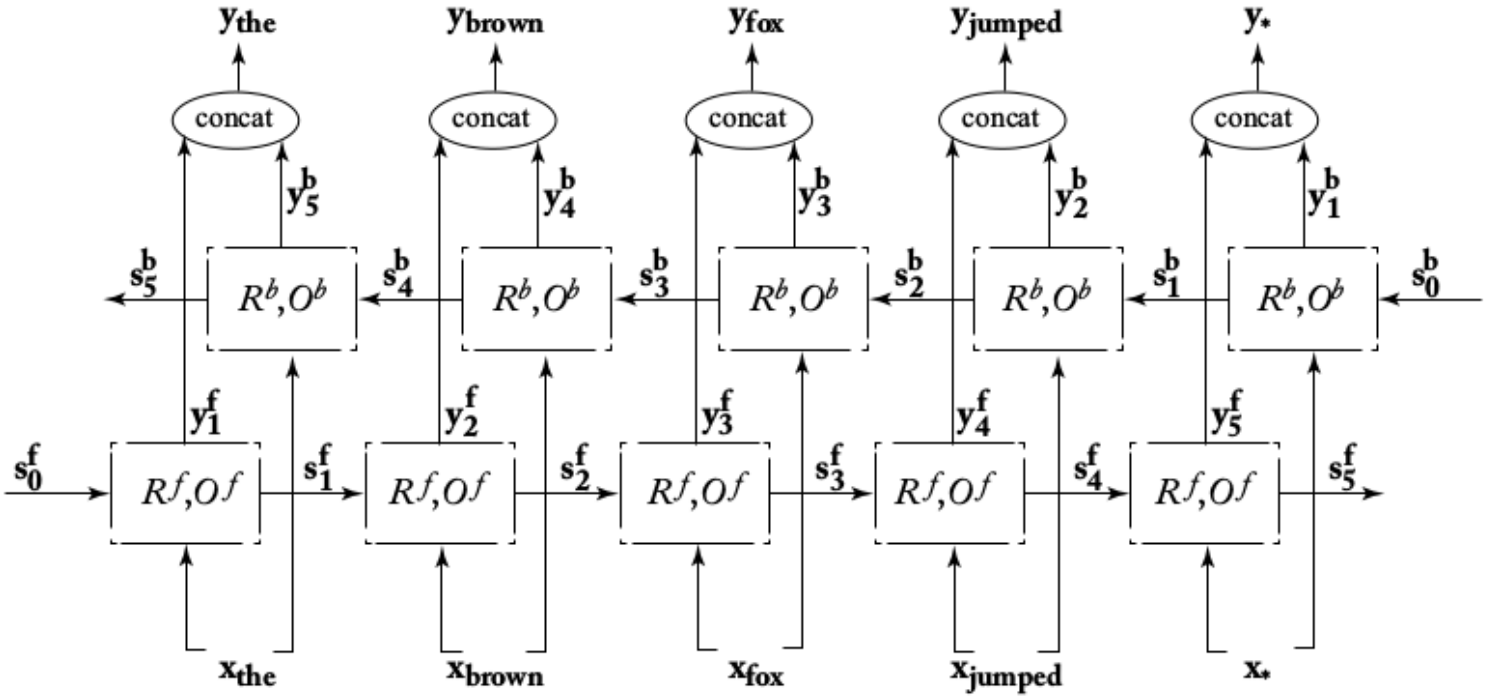
Bidirectional RNNS (BIRNN)

- Similarly to the *RNN* case, we also define $biRNN^*(\vec{x}_{1:n})$ as the sequence of vectors $\vec{y}_{1:n}$:

$$biRNN^*(\vec{x}_{1:n}) = \vec{y}_{1:n} = biRNN(\vec{x}_{1:n}, 1), \ldots, biRNN(\vec{x}_{1:n}, n)$$

- The n output vectors $\vec{y}_{1:n}$ can be efficiently computed in linear time by first running the forward and backward RNNs, and then concatenating the relevant outputs.

Bidirectional RNNS (BIRNN)

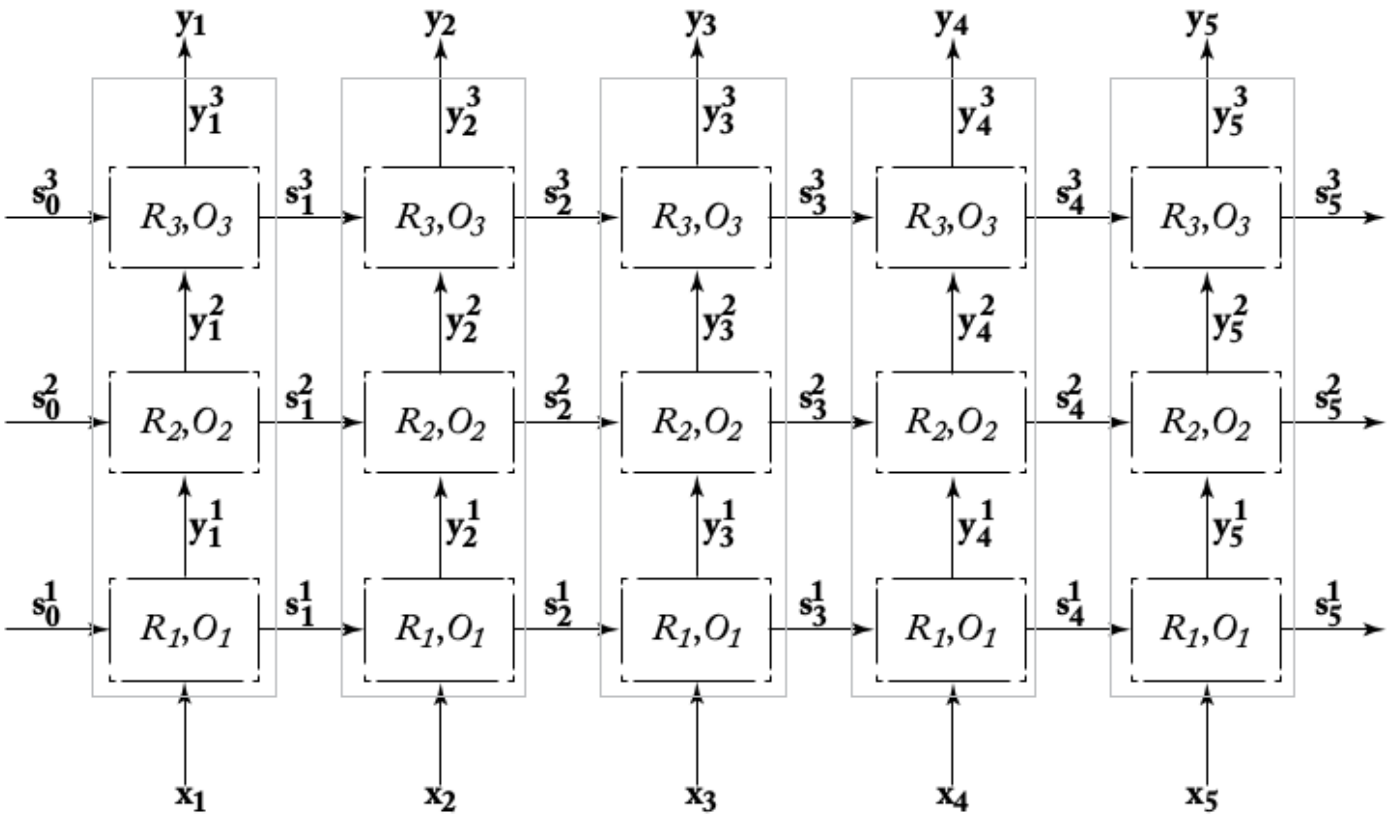


- The biRNN is very effective for tagging tasks, in which each input vector corresponds to one output vector.
- It is also useful as a general-purpose trainable feature-extracting component, that can be used whenever a window around a given word is required.

Multi-layer (stacked) RNNS

- RNNs can be stacked in layers, forming a grid.
- Consider k RNNs, $RNN_1, \ldots, RNN_k$, where the jth RNN has states $\vec{s}^j_{1:n}$ and outputs $\vec{y}^j_{1:n}$.
- The input for the first RNN are $\vec{x}_{1:n}$.
- The input of the jth RNN ($j \geq 2$) are the outputs of the RNN below it, $\vec{y}^{j-1}_{1:n}$.
- The output of the entire formation is the output of the last RNN, $\vec{y}^k_{1:n}$.
- Such layered architectures are often called deep RNNs.

Multi-layer (stacked) RNNS



- It is not theoretically clear what is the additional power gained by the deeper architecture.
- It was observed empirically that deep RNNs work better than shallower ones on some tasks (e.g., machine translation).

Elman Network or Simple-RNN

- After describing the RNN abstraction, we are now in place to discuss specific instantiations of it.
- Recall that we are interested in a recursive function $\vec{s}_i = R(\vec{x}_i, \vec{s}_{i-1})$ such that $\vec{s}_i$ encodes the sequence $\vec{x}_{1:n}$.
- The simplest RNN formulation is known as an Elman Network or Simple-RNN (S-RNN).

Elman Network or Simple-RNN

$$\begin{aligned} \vec{s}_i &= R_{SRNN}(\vec{x}_i, \vec{s}_{i-1}) = g(\vec{s}_{i-1} W^s + \vec{x}_i W^x + \vec{b}) \\ \vec{y}_i &= O_{SRNN}(\vec{s}_i) = \vec{s}_i \end{aligned} \tag{11}$$

$$\vec{s}_i, \vec{y}_i \in \mathcal{R}^{d_s}, \quad \vec{x}_i \in \mathcal{R}^{d_x}, \quad W^x \in \mathcal{R}^{d_x \times d_s}, \quad W^s \in \mathcal{R}^{d_s \times d_s}, \vec{b} \in \mathcal{R}^{d_s}$$

- The state $\vec{s}_i$ and the input $\vec{x}_i$ are each linearly transformed.
- The results are added (together with a bias term) and then passed through a nonlinear activation function g (commonly tanh or ReLU).
- The Simple RNN provides strong results for sequence tagging as well as language modeling.

Gated Architectures

- The S-RNN is hard to train effectively because of the **vanishing gradients** problem.
- Error signals (gradients) in later steps in the sequence diminish quickly in the backpropagation process.
- Thus, they do not reach earlier input signals, making it hard for the S-RNN to capture long-range dependencies.
- Gating-based architectures, such as the LSTM [Hochreiter and Schmidhuber, 1997] and the GRU [Cho et al., 2014b] are designed to solve this deficiency.

Gated Architectures

- Consider the RNN as a general purpose computing device, where the state $\vec{s}_i$ represents a finite memory.
- Each application of the function R reads in an input $\vec{x}_{i+1}$, reads in the current memory $\vec{s}_i$, operates on them in some way, and writes the result into memory.
- This resuls in a new memory state $\vec{s}_{i+1}$.
- An apparent problem with the S-RNN architecture is that the memory access is not controlled.
- At each step of the computation, the entire memory state is read, and the entire memory state is written.

Gated Architectures

- How does one provide more controlled memory access?
- Consider a binary vector $\vec{g} \in 0, 1^n$.
- Such a vector can act as a **gate** for controlling access to n-dimensional vectors, using the hadamard-product operation $\vec{x} \odot \vec{g}$.
- The hadamard operation is the same as the element-wise multiplication of two vectors:

$$\vec{x} = \vec{u} \odot \vec{v} \Leftrightarrow \vec{x}_{[i]} = \vec{u}_{[i]} \cdot \vec{v}_{[i]} \quad \forall i \in [1, n]$$

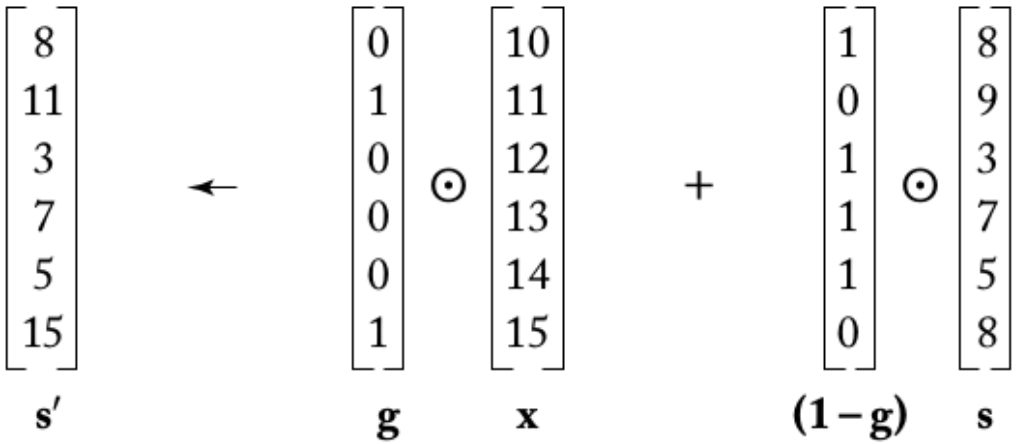
Gated Architectures

- Consider a memory $\vec{s} \in \mathcal{R}^d$, an input $\vec{x} \in \mathcal{R}^d$ and a gate $\vec{g} \in [0,1]^d$.
- The following computation:

$$\vec{s}' \leftarrow \vec{g} \odot \vec{x} + (\vec{1} - \vec{g}) \odot (\vec{s})$$

- Reads the entries in $\vec{x}$ that correspond to the $\vec{1}$ values in $\vec{g}$, and writes them to the new memory $\vec{s}'$.
- Locations that weren't read to are copied from the memory $\vec{s}$ to the new memory $\vec{s}'$ through the use of the gate $(\vec{1} - \vec{g})$.

Gated Architectures



- This gating mechanism can serve as a building block in our RNN.
- Gate vectors can be used to control access to the memory state $\vec{s}_i$.
- We are still missing two important (and related) components:
  1. The gates should not be static, but be controlled by the current memory state and the input.
  2. Their behavior should be learned.
- This introduced an obstacle, as learning in our framework entails being differentiable (because of the backpropagation algorithm).
- The binary 0-1 values used in the gates are not differentiable.

Gated Architectures

- A solution to this problem is to approximate the hard gating mechanism with a soft—but differentiable—gating mechanism.
- To achieve these differentiable gates , we replace the requirement that $\vec{g} \in 0, 1^n$ and allow arbitrary real numbers, $\vec{g}' \in \mathcal{R}^n$.
- These real numbers are then pass through a sigmoid function $\sigma(\vec{g}')$.
- This bounds the value in the range $(0, 1)$, with most values near the borders.

Gated Architectures

- When using the gate $\sigma(g') \odot \vec{x}$, indices in $\vec{x}$ corresponding to near-one values in $\sigma(\vec{g}')$ are allowed to pass.
- While those corresponding to near-zero values are blocked.
- The gate values can then be conditioned on the input and the current memory.
- And can be trained using a gradient-based method to perform a desired behavior.
- This controllable gating mechanism is the basis of the LSTM and the GRU architectures.
- At each time step, differentiable gating mechanisms decide which parts of the inputs will be written to memory and which parts of memory will be overwritten (forgotten).

LSTM

- The Long Short-Term Memory (LSTM) architecture [Hochreiter and Schmidhuber, 1997] was designed to solve the vanishing gradients problem.
- It was the the first architecture to introduce the gating mechanism.
- The LSTM architecture explicitly splits the state vector $\vec{s}_i$ into two halves: 1) memory cells and 2) working memory.
- The memory cells are designed to preserve the memory, and also the error gradients, across time, and are controlled through differentiable gating components[9].
- At each input state, a gate is used to decide how much of the new input should be written to the memory cell, and how much of the current content of the memory cell should be forgotten.

[9] Smooth mathematical functions that simulate logical gates.

LSTM

- Mathematically, the LSTM architecture is defined as:

$$
\begin{aligned}
\boldsymbol{s}_j = R_{\text{LSTM}}(\boldsymbol{s}_{j-1}, \boldsymbol{x}_j) &= [\boldsymbol{c}_j; \boldsymbol{h}_j] \\
\boldsymbol{c}_j &= \boldsymbol{f} \odot \boldsymbol{c}_{j-1} + \boldsymbol{i} \odot \boldsymbol{z} \\
\boldsymbol{h}_j &= \boldsymbol{o} \odot \tanh(\boldsymbol{c}_j) \\
\boldsymbol{i} &= \sigma(\boldsymbol{x}_j \boldsymbol{W}^{xi} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hi}) \\
\boldsymbol{f} &= \sigma(\boldsymbol{x}_j \boldsymbol{W}^{xf} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hf}) \\
\boldsymbol{o} &= \sigma(\boldsymbol{x}_j \boldsymbol{W}^{xo} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{ho}) \\
\boldsymbol{z} &= \tanh(\boldsymbol{x}_j \boldsymbol{W}^{xz} + \boldsymbol{h}_{j-1} \boldsymbol{W}^{hz})
\end{aligned}
$$

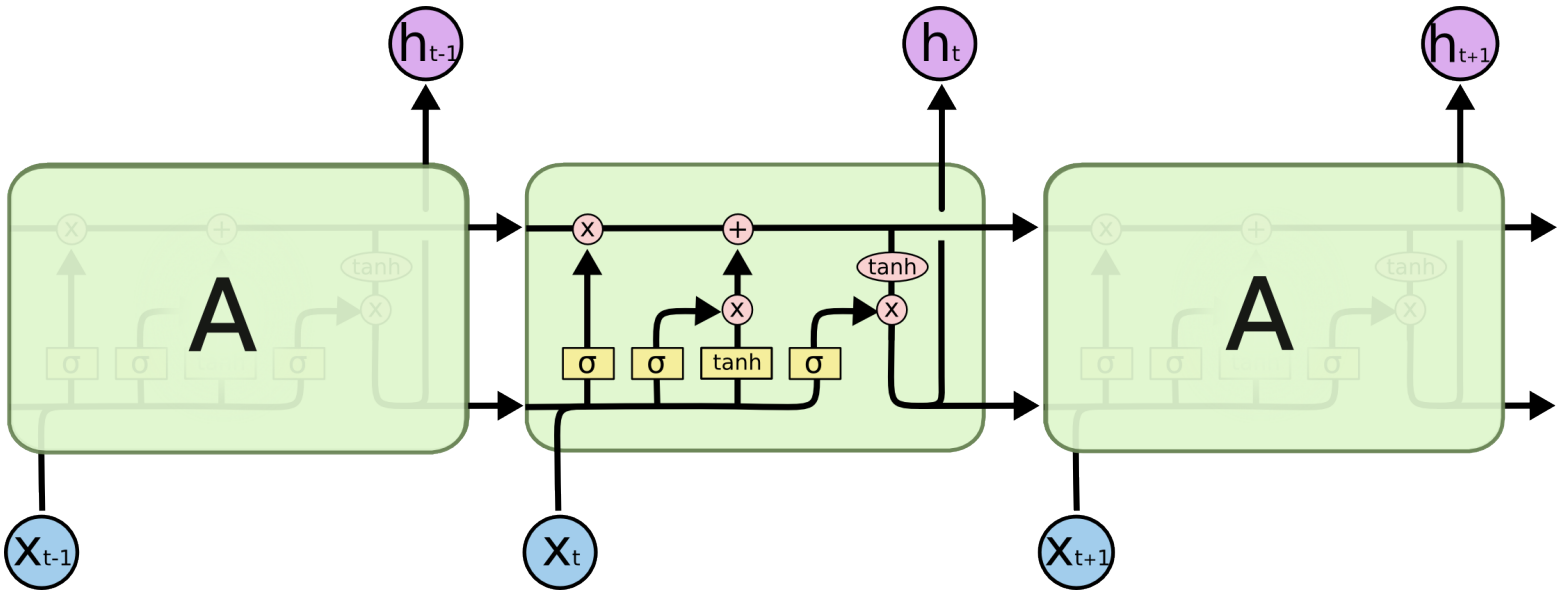
$$
\boldsymbol{y}_j = O_{\text{LSTM}}(\boldsymbol{s}_j) = \boldsymbol{h}_j
$$

$$
\boldsymbol{s}_j \in \mathbb{R}^{2 \cdot d_h}, \quad \boldsymbol{x}_i \in \mathbb{R}^{d_x}, \quad \boldsymbol{c}_j, \boldsymbol{h}_j, \boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \boldsymbol{z} \in \mathbb{R}^{d_h}, \quad \boldsymbol{W}^{xo} \in \mathbb{R}^{d_x \times d_h}, \quad \boldsymbol{W}^{ho} \in \mathbb{R}^{d_h \times d_h}.
$$

LSTM

- The state at time j is composed of two vectors, $\vec{c}_j$ and h_j, where $\vec{c}_j$ is the memory component and $\vec{h}_j$ is the hidden state component.
- There are three gates, $\vec{i}$, $\vec{f}$, and $\vec{o}$, controlling for **i**nput, **f**orget, and **o**utput.
- The gate values are computed based on linear combinations of the current input $\vec{x}_j$ and the previous state $\vec{h}_{j-1}$, passed through a sigmoid activation function.
- An update candidate $\vec{z}$ is computed as a linear combination of $\vec{x}_j$ and $\vec{h}_{j-1}$, passed through a tanh activation function (to push the values to be between -1 and 1).
- The memory $\vec{c}_j$ is then updated: the forget gate controls how much of the previous memory to keep ($\vec{f} \odot \vec{c}_{j-1}$), and the input gate controls how much of the proposed update to keep ($\vec{i} \odot \vec{z}$).
- Finally, the value of $\vec{h}_j$ (which is also the output $\vec{y}_j$) is determined based on the content of the memory $\vec{c}_j$, passed through a tanh nonlinearity and controlled by the output gate.
- The gating mechanisms allow for gradients related to the memory part $\vec{c}_j$ to stay high across very long time ranges.

LSTM



[10] source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM

- Intuitively, recurrent neural networks can be thought of as very deep feed-forward networks, with shared parameters across different layers.
- For the Simple-RNN, the gradients then include repeated multiplication of the matrix W.
- This makes the gradient values to vanish or explode.
- The gating mechanism mitigate this problem to a large extent by getting rid of this repeated multiplication of a single matrix.
- LSTMs are currently the most successful type of RNN architecture, and they are responsible for many state-of-the-art sequence modeling results.
- The main competitor of the LSTM RNN is the GRU, to be discussed next.

GRU

- The LSTM architecture is very effective, but also quite complicated.
- The complexity of the system makes it hard to analyze, and also computationally expensive to work with.
- The gated recurrent unit (GRU) was recently introduced by Cho et al. [2014b] as an alternative to the LSTM.
- It was subsequently shown by Chung et al. [2014] to perform comparably to the LSTM on several (non textual) datasets.
- Like the LSTM, the GRU is also based on a gating mechanism, but with substantially fewer gates and without a separate memory component.

GRU

$$
\begin{aligned}
s_j = R_{\text{GRU}}(s_{j-1}, x_j) &= (1 - z) \odot s_{j-1} + z \odot \tilde{s_j} \\
z &= \sigma(x_j W^{xz} + s_{j-1} W^{sz}) \\
r &= \sigma(x_j W^{xr} + s_{j-1} W^{sr}) \\
\tilde{s_j} &= \tanh(x_j W^{xs} + (r \odot s_{j-1}) W^{sg})
\end{aligned}
$$

$$
y_j = O_{\text{GRU}}(s_j) = s_j
$$

$$
s_j, \tilde{s_j} \in \mathbb{R}^{d_s}, \; x_i \in \mathbb{R}^{d_x}, \; z, r \in \mathbb{R}^{d_s}, \; W^{x\circ} \in \mathbb{R}^{d_x \times d_s}, \; W^{s\circ} \in \mathbb{R}^{d_s \times d_s}.
$$

GRU

- One gate $\vec{r}$ is used to control access to the previous state $\vec{s}_{j-1}$ and compute a proposed update $\vec{\tilde{s}}_j$.
- The updated state $\vec{s}_j$ (which also serves as the output $\vec{y}_j$) is then determined based on an interpolation of the previous state $\vec{s}_{j-1}$ and the proposal $\vec{\tilde{s}}_j$.
- The proportions of the interpolation are controlled using the gate $\vec{z}$.
- The GRU was shown to be effective in language modeling and machine translation.
- However, the jury is still out between the GRU, the LSTM and possible alternative RNN architectures, and the subject is actively researched.
- For an empirical exploration of the GRU and the LSTM architectures, see Jozefowicz et al. [2015].

Sentiment Classification with RNNs

- The simplest use of RNNs is as acceptors: read in an input sequence, and produce a binary or multi-class answer at the end.
- RNNs are very strong sequence learners, and can pick-up on very intricate patterns in the data.
- An example of naturally occurring positive and negative sentences in the movie-reviews domain would be the following:
- Positive: It's not life-affirming—it's vulgar and mean, but I liked it.
- Negative: It's a disappointing that it only manages to be decent instead of dead brilliant.

Sentiment Classification with RNNs

- Note that the positive example contains some negative phrases (not life affirming, vulgar, and mean).
- While the negative example contains some positive ones (dead brilliant).
- Correctly predicting the sentiment requires understanding not only the individual phrases but also the context in which they occur, linguistic constructs such as negation, and the overall structure of the sentence.

Sentiment Classification with RNNs

- The sentence-level sentiment classification task is modelled using an RNN-acceptor.
- After tokenization, the RNN reads in the words of the sentence one at a time.
- The final RNN state is then fed into an MLP followed by a softmax-layer with two outputs (positive and negative).
- The network is trained with cross-entropy loss based on the gold sentiment labels.

$$
\begin{aligned}
p(label = k|\vec{w}_{1:n}) &= \hat{\vec{y}}_{[k]} \\
\hat{\vec{y}} &= softmax(MLP(RNN(\vec{x}_{1:n}))) \\
\vec{x}_{1:n} &= E_{[w_1]}, \ldots, E_{[w_n]}
\end{aligned}
\tag{12}
$$

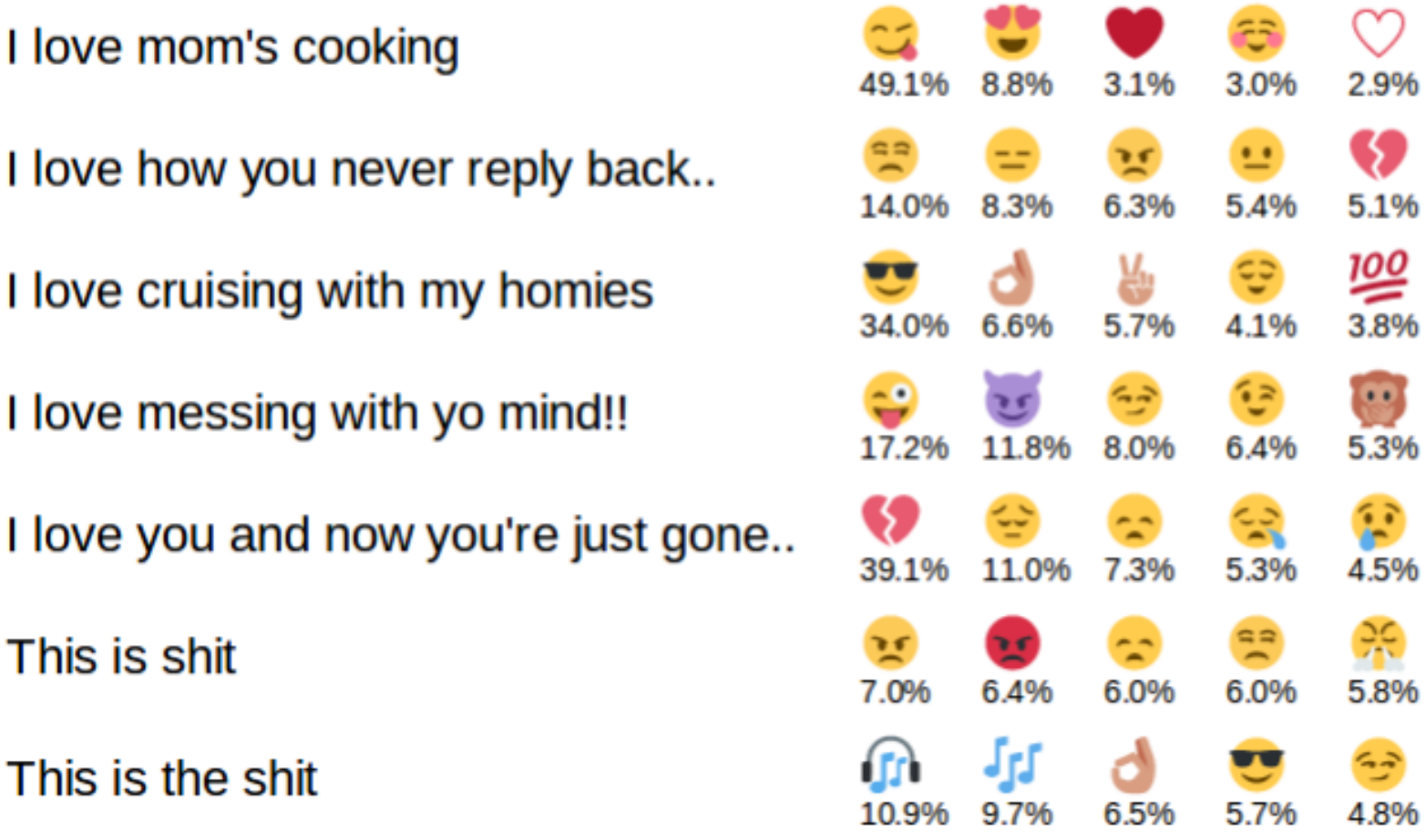
Sentiment Classification with RNNs

- The word embeddings matrix E is initialized using pre-trained embeddings learned over a large external corpus using an algorithm such as word2vec or Glove with a relatively wide window.
- It is often helpful to extend the model by considering bidirectional RNNs-
- For longer sentences, Li et al. [2015] found it useful to use a hierarchical architecture, in which the sentence is split into smaller spans based on punctuation.
- Then, each span is fed into a bidirectional RNN.
- Sequence of resulting vectors (onefor each span) are then fed into an RNN acceptor.
- A similar hierarchical architecture was used for document-level sentiment classification in Tang et al. [2015].

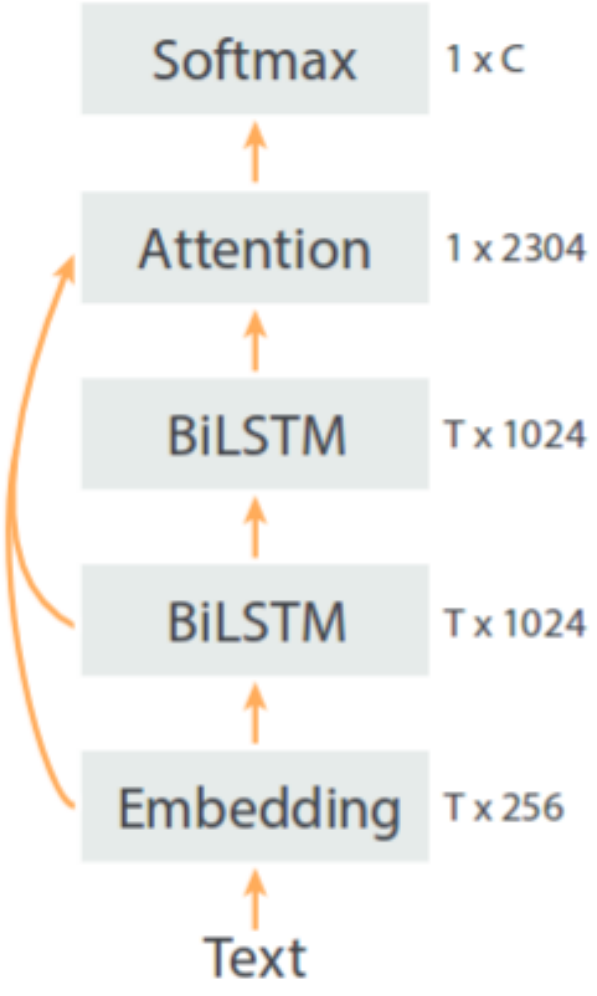
Twitter Sentiment Classification with LSTMS Emojis

- An emoji-based distant supervision model for detecting sentiment and other affective states from short social media messages was proposed in [Felbo et al., 2017].
- Emojis are used as a distant supervision approach for various affective detection tasks (e.g., emotion, sentiment, sarcasm) using a large corpus of 634M tweets with 64 emojis.
- A neural network architecture is pretrained with this corpus.
- The network is an LSTM variant formed by an embedding layer, 2 bidirectional LSTM layers with normal skip connections and temporal average pooling-skip connections.

DeepEmoji


I love mom's cooking
49.1% 8.8% 3.1% 3.0% 2.9%
I love how you never reply back..
14.0% 8.3% 6.3% 5.4% 5.1%
I love cruising with my homies
34.0% 6.6% 5.7% 4.1% 3.8%
I love messing with yo mind!!
17.2% 11.8% 8.0% 6.4% 5.3%
I love you and now you're just gone..
39.1% 11.0% 7.3% 5.3% 4.5%
This is shit
7.0% 6.4% 6.0% 6.0% 5.8%
This is the shit
10.9% 9.7% 6.5% 5.7% 4.8%

DeepEmoji


Softmax
1 x C
Attention
1 x 2304
BiLSTM
T x 1024
BiLSTM
T x 1024
Embedding
T x 256
Text

Twitter Sentiment Classification with LSTMS Emojis

- Authors propose the chain-thaw transfer-learning approach in which the pretrained network is fine-tuned for the target task.
- Here, each layer is individually fine-tuned in each step with the target gold data, and then they are all fine-tuned together.
- The model achieves state-of-the-art results the detection of emotion, sentiment, and sarcasm.
- The pretrained network is released to the public.
- A demo of the model: `https://github.com/bfelbo/DeepMoji`.

DeepEmoji

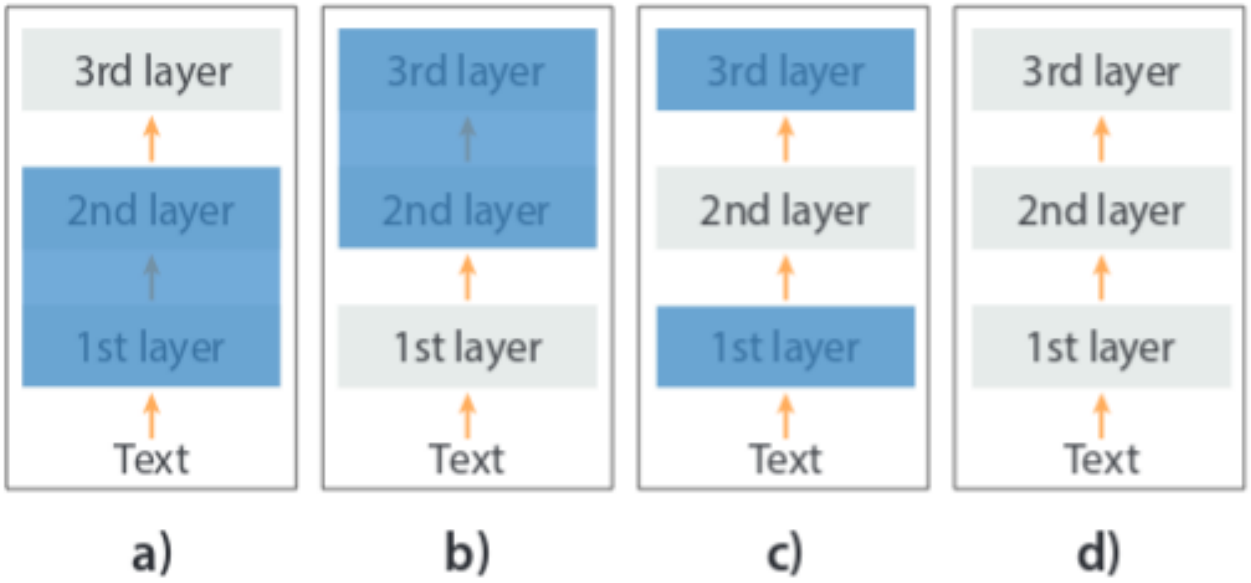


Figure 2: Illustration of the chain-thaw transfer learning approach, where each layer is fine-tuned separately. Layers covered with a blue rectangle are frozen. Step a) tunes any new layers, b) then tunes the 1st layer and c) the next layer until all layers have been fine-tuned individually. Lastly, in step d) all layers are fine-tuned together.

Language Models and Language Generation

- Language modeling is the task of assigning a probability to sentences in a language.
- Example: what is the probability of seeing the sentence “the lazy dog barked loudly”?
- The task can be formulated as the task of predicting the probability of seing a word conditioned on previous words:

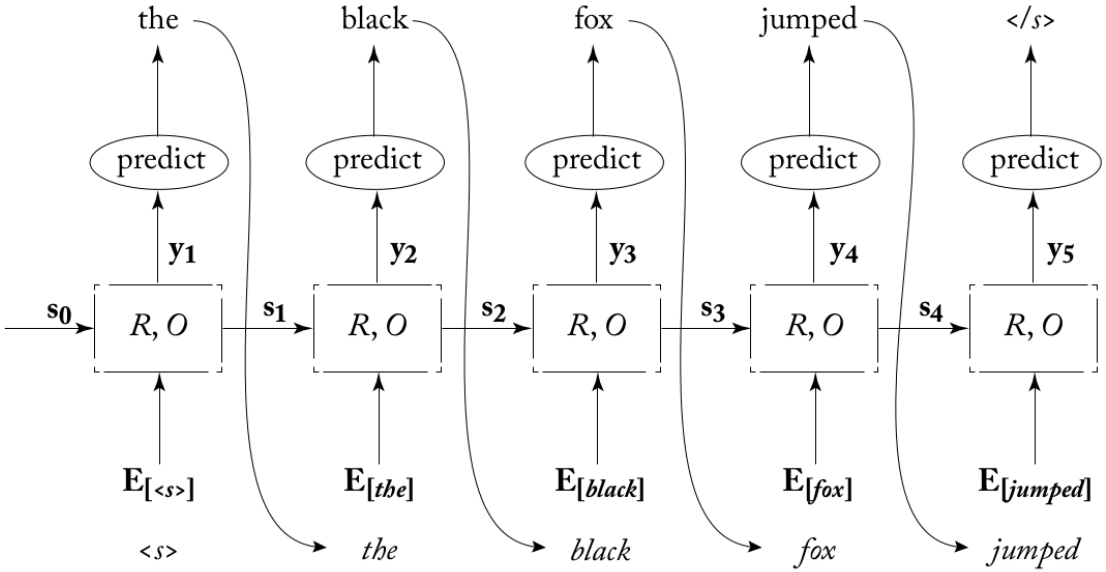
$$P(w_i|w_1, w_2, \cdots, w_{i-1}) = \frac{P(w_1, w_2, \cdots, w_{i-1}, w_i)}{P(w_1, w_2, \cdots, w_{i-1})}$$

Language Models and Language Generation

- RNNs can be used to train language models by tying the output at time *i* with its input at time $i + 1$.
- This network can be used to generate sequences of words or random sentences.
- Generation process: predict a probability distribution over the first word conditioned on the start symbol, and draw a random word according to the predicted distribution.
- Then predict a probability distribution over the second word conditioned on the first, and so on, until predicting the end-of-sequence $< /\text{s}>$ symbol.

Language Models and Language Generation

- After predicting a distribution over the next output symbols $P(t_i = k|t_{1:i-1})$, a token t_i is chosen and its corresponding embedding vector is fed as the input to the next step.



- Teacher-forcing: during **training** the generator is fed with the ground-truth previous word even if its own prediction put a small probability mass on it.
- It is likely that the generator would have generated a different word at this state in **test time**.

Sequence to Sequence Problems

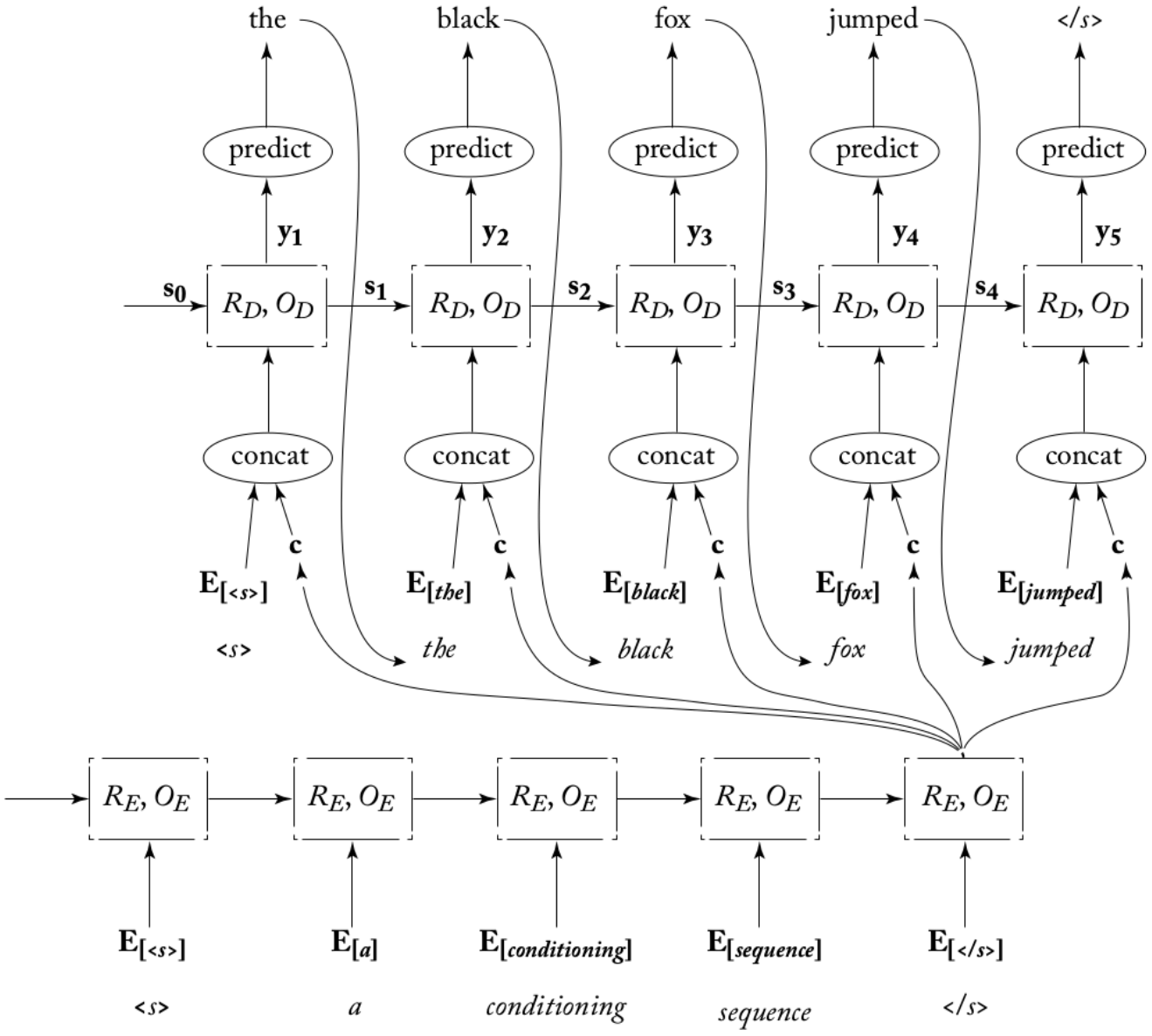
Nearly any task in NLP can be formulated as a sequence to sequence (or condionated generation) task i.e., generate output sequences from input ones. Input and output sequences can have different lengths.

- Machine Translation: source language to target language.
- Summarization: long text to short text.
- Dialogue (chatbots): previous utterances to next utterance.

Conditioned Generation

- While using the RNN as a generator is a cute exercise for demonstrating its strength, the power of RNN gnerator is really revealed when moving to a conditioned generation or enconder-decoder framework.
- Core idea: using two RNNs.
- Encoder: One RNN is used to encode the source input into a vector $\overrightarrow{c}$.
- Decoder: Another RNN is used to decode the encoder's output and generate the target output.
- At each stage of the generation process the context vector $\overrightarrow{c}$ is concatenated to the input $\hat{t}_j$ and the concatenation is fed into the RNN.

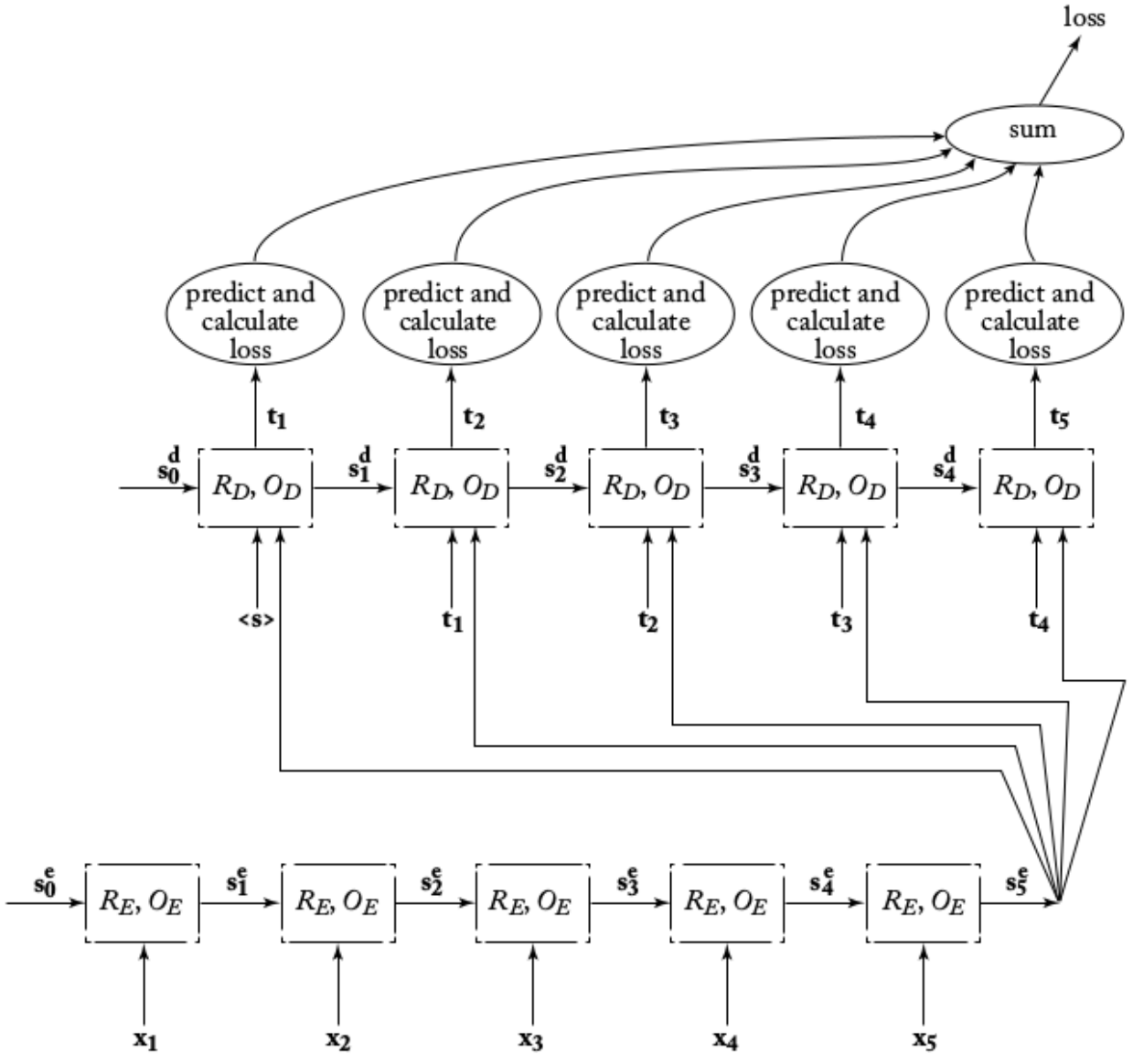
Encoder Decoder Framework



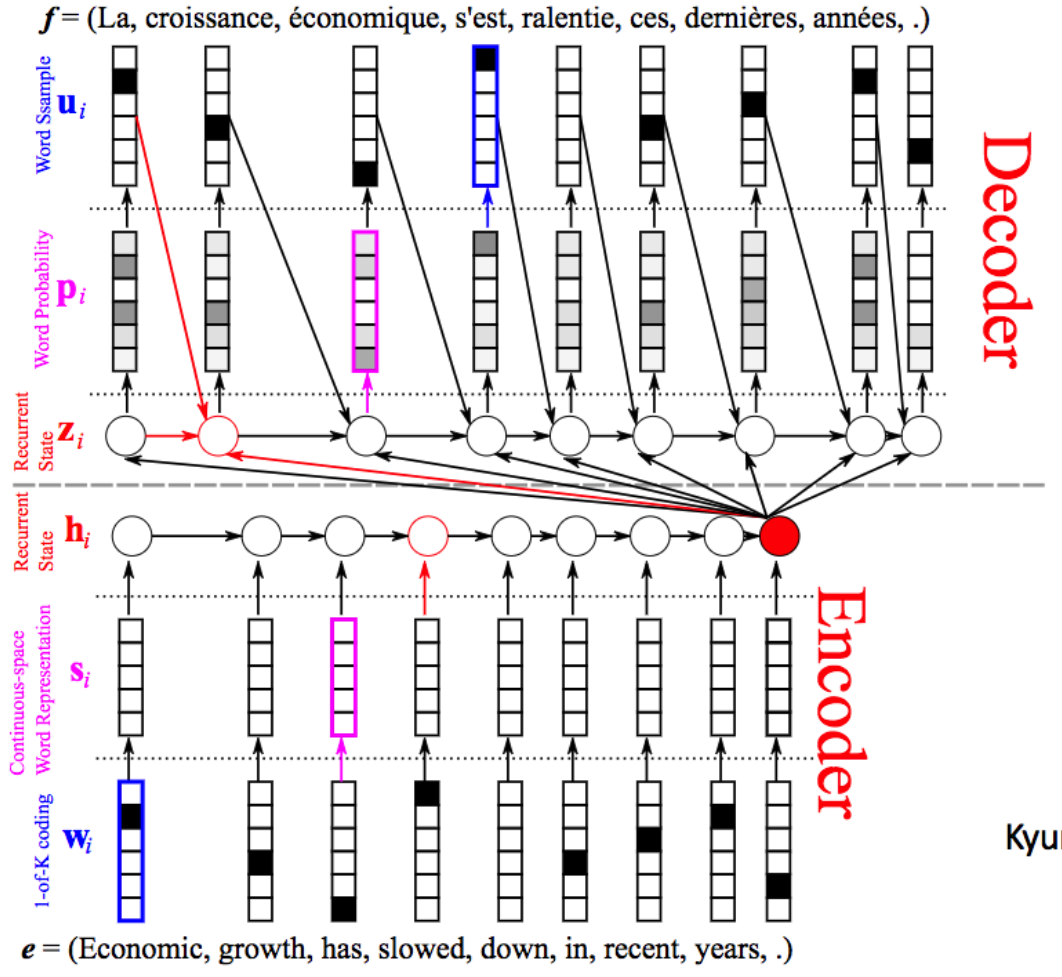
Conditioned Generation

- This setup is useful for mapping sequences of length n to sequences of length m.
- The encoder summarizes the source sentence as a vector $\vec{c}$.
- The decoder RNN is then used to predict (using a language modeling objective) the target sequence words conditioned on the previously predicted words as well as the encoded sentence $\vec{c}$.
- The encoder and decoder RNNs are trained jointly.
- The supervision happens only for the decoder RNN, but the gradients are propagated all the way back to the encoder RNN.

Sequence to Sequence Training Graph

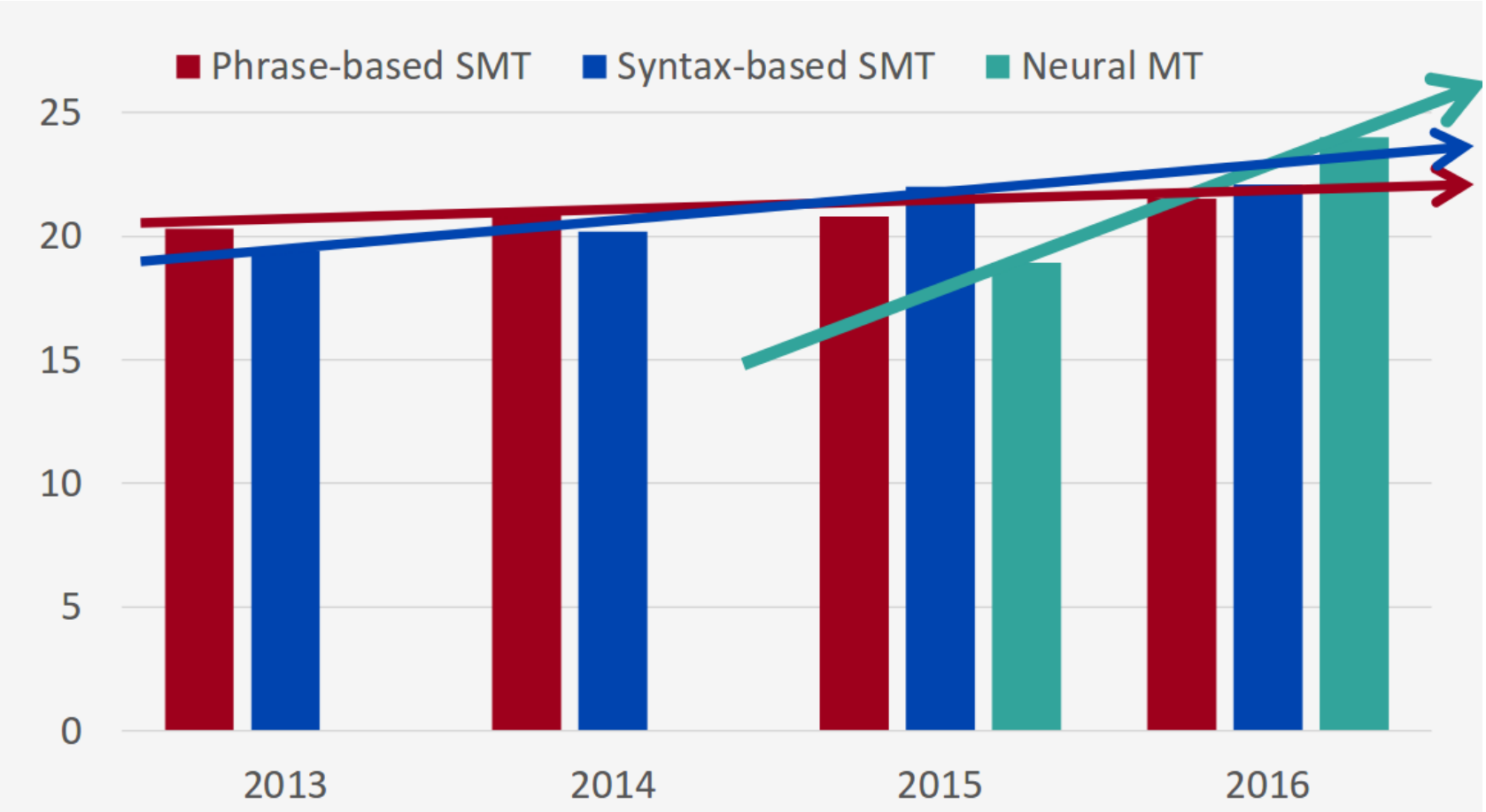


Neural Machine Translation



Kyunghyun Cho et al. 2014

Machine Translation BLEU progress over time



[Edinburgh En-De WMT]

[10] source: http://www.meta-net.eu/events/meta-forum-2016/slides/09_sennrich.pdf

Decoding Approaches

- The decoder aims to generate the output sequence with maximal score (or maximal probability), i.e., such that $\sum_{i=1}^{n} P(\hat{t}_i|\hat{t}_{1:i-1})$ is maximized.
- The non-markovian nature of the RNN means that the probability function cannot be decomposed into factors that allow for exact search using standard dynamic programming.
- Exact search: finding the optimum sequence requires evaluating every possible sequence (computationally prohibitive).
- Thus, it only makes sense to solving the optimization problem above approximately.
- Greedy search: choose the highest scoring prediction (word) at each step.
- This may result in sub-optimal overall probability leading to prefixes that are followed by low-probability events.

Greedy Search

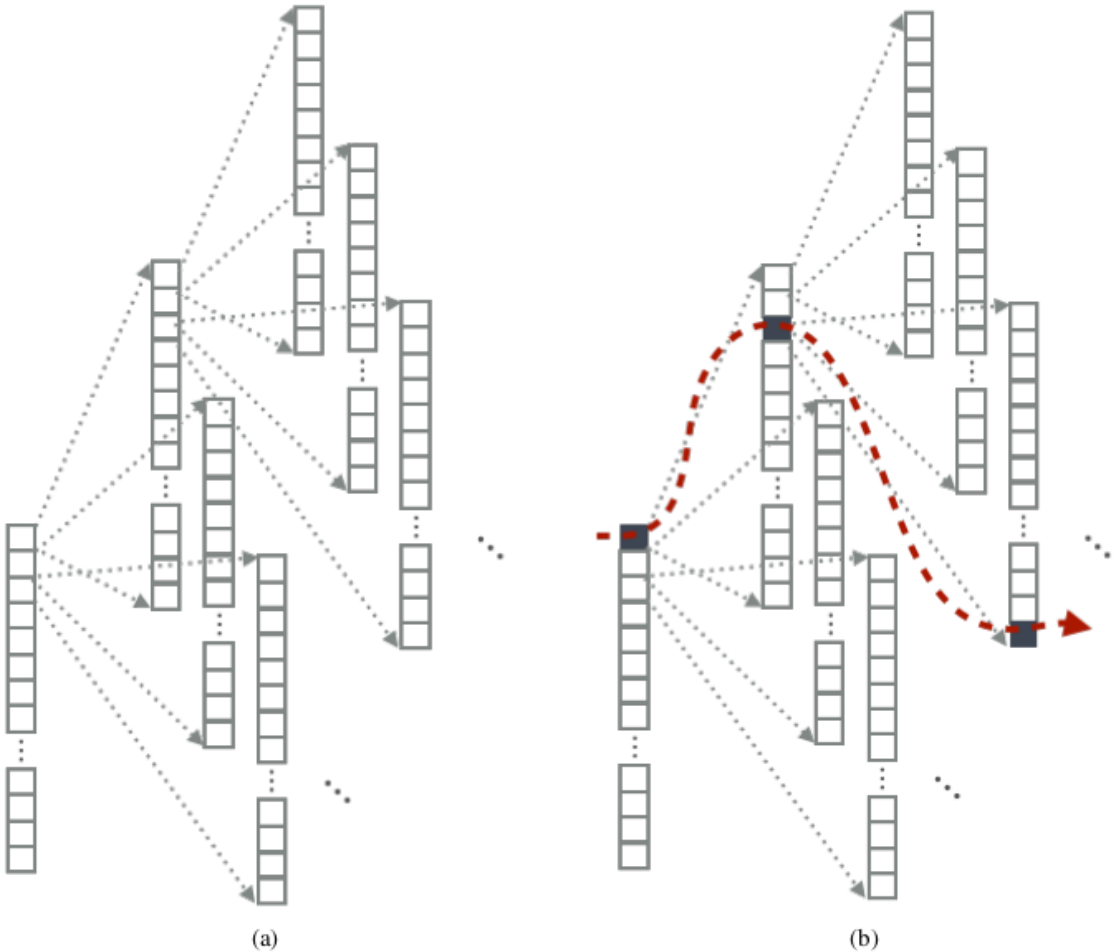


Figure 6.4: (a) Search space depicted as a tree. (b) Greedy search.

[10] Source: [Cho, 2015]

Beam Search

- Beam search interpolates between the exact search and the greedy search by changing the size K of hypotheses maintained throughout the search procedure [Cho, 2015].
- We first pick the K starting words with the highest probability
- At each step, each candidate sequence is expanded with all possible next steps.
- Each candidate step is scored.
- The K sequences with the most likely probabilities are selected and all other candidates are pruned.
- The search process can halt for each candidate separately either by reaching a maximum length, by reaching an end-of-sequence token, or by reaching a threshold likelihood.
- The sentence with the highest overall probability is selected.

[10] More info at: `https://machinelearningmastery.com/beam-search-decoder-natural-language-processing/`

Conditioned Generation with Attention

- In the encoder-decoder networks the input sentence is encoded into a single vector, which is then used as a conditioning context for an RNN-generator.
- This architectures forces the encoded vector $\vec{c}$ to contain all the information required for generation.
- It doesn't work well for long sentences!
- It also requires the generator to be able to extract this information from the fixed-length vector.
- "You can't cram the meaning of a whole %&!$# sentence into a single $&!#* vector!" -Raymond Mooney
- This architecture can be can be substantially improved (in many cases it) by the addition of an attention mechanism.
- The attention mechanism attempts to solve this problem by allowing the decoder to "look back" at the encoder's hidden states based on its current state.

Conditioned Generation with Attention

- The input sentence (a length n input sequence $\vec{x}_{1:n}$) is encoded using a biRNN as a sequence of vectors $\vec{c}_{1:n}$.
- The decoder uses a soft attention mechanism in order to decide on which parts of the encoding input it should focus.
- At each stage j the decoder sees a weighted average of the vectors $\vec{c}_{1:n}$, where the attention weights ($\vec{\alpha}^j$) are chosen by the attention mechanism.

$$\vec{c}^j = \sum_{i=1}^{n} \vec{\alpha}^j_{[i]} \cdot \vec{c}_i$$

- The elemements of $\vec{\alpha}^j$ are all positive and sum to one.

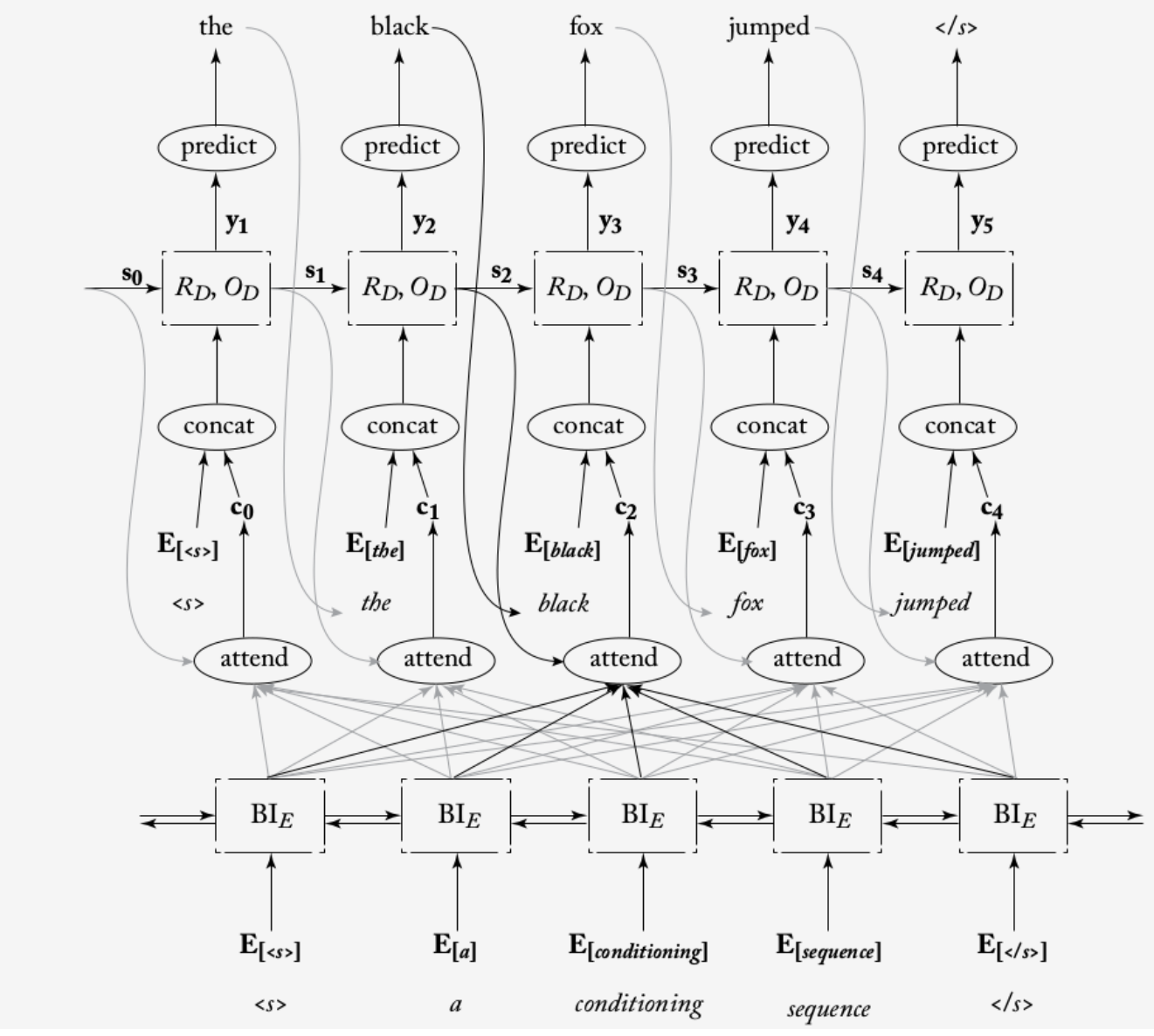
Conditioned Generation with Attention

- Unnormalized attention weights are produced using a feed-forward network MLP taking into account the decoder state at time j ($\vec{s}_j$) and each of the vectors $\vec{c}_i$.
- These unnormalized weights are then normalized into a probability distribution using the softmax function.

$$\begin{aligned}
\text{attend}(\boldsymbol{c_{1:n}}, \hat{t}_{1:j}) &= \boldsymbol{c^j} \\
\boldsymbol{c^j} &= \sum_{i=1}^{n} \boldsymbol{\alpha^j_{[i]}} \cdot \boldsymbol{c_i} \\
\boldsymbol{\alpha^j} &= \text{softmax}(\bar{\boldsymbol{\alpha}}^j_{[1]}, \ldots, \bar{\boldsymbol{\alpha}}^j_{[n]}) \\
\bar{\boldsymbol{\alpha}}^j_{[i]} &= \text{MLP}^{\text{att}}([\boldsymbol{s_j}; \boldsymbol{c_i}]),
\end{aligned}$$

- The encoder, decoder, and attention mechanism are all trained jointly in order to play well with each other.

Attention


the
black
fox
jumped
</s>
predict
y1
y2
y3
y4
y5
s0
s1
s2
s3
s4
RD, OD
concat
c0
c1
c2
c3
c4
E[<s>]
E[the]
E[black]
E[fox]
E[jumped]
<s>
the
black
fox
jumped
attend
BIE
E[<s>]
E[a]
E[conditioning]
E[sequence]
E[</s>]
<s>
a
conditioning
sequence
</s>

Attention and Word Alignments

- In the context of machine translation, one can think of MLP att as computing a soft alignment between the current decoder state $\vec{s}_j$ (capturing the recently produced foreign words) and each of the source sentence components $\vec{c}_i$.

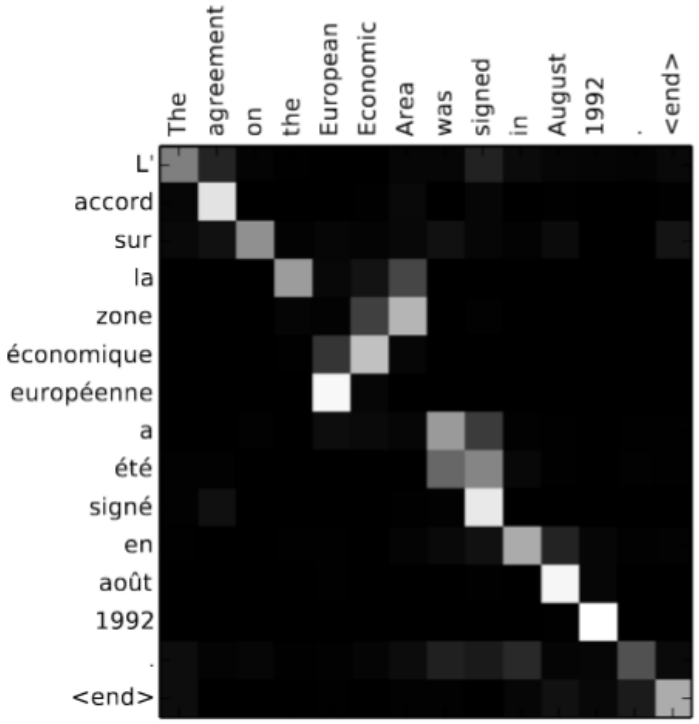


Fig. 2. Visualization of the attention weights α_j^t of the attention-based neural machine translation model [32]. Each row corresponds to the output symbol, and each column the input symbol. Brighter the higher α_j^t.

Figure: Source: [Cho et al., 2015]

Other types of Attention

Summary

Below is a summary table of several popular attention mechanisms (or broader categories of attention mechanisms).

Name	Alignment score function	Citation
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment max to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017
Self-Attention(&)	Relating different positions of the same input sequence. Theoretically the self-attention can adopt any score functions above, but just replace the target sequence with the same input sequence.	Cheng2016
Global/Soft	Attending to the entire input state space.	Xu2015
Local/Hard	Attending to the part of input state space; i.e. a patch of the input image.	Xu2015; Luong2015

(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.
(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.
(&) Also, referred to as "intra-attention" in Cheng et al., 2016 and some other papers.

Figure: Source: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Recursive Neural Networks over Sentiment Treebank

- A recursive neural tensor network for learning the sentiment of pieces of texts of different granularities, such as words, phrases, and sentences, was proposed in [Socher et al., 2013].
- The network was trained on a sentiment annotated treebank `http://nlp.stanford.edu/sentiment/treebank.html` of parsed sentences for learning compositional vectors of words and phrases.
- Every node in the parse tree receives a vector, and there is a matrix capturing how the meaning of adjacent nodes changes.
- The network is trained using a variation of backpropagation called Backprop through Structure.
- The main drawback of this model is that it relies on parsing.

Recursive Neural Networks over Sentiment Treebank

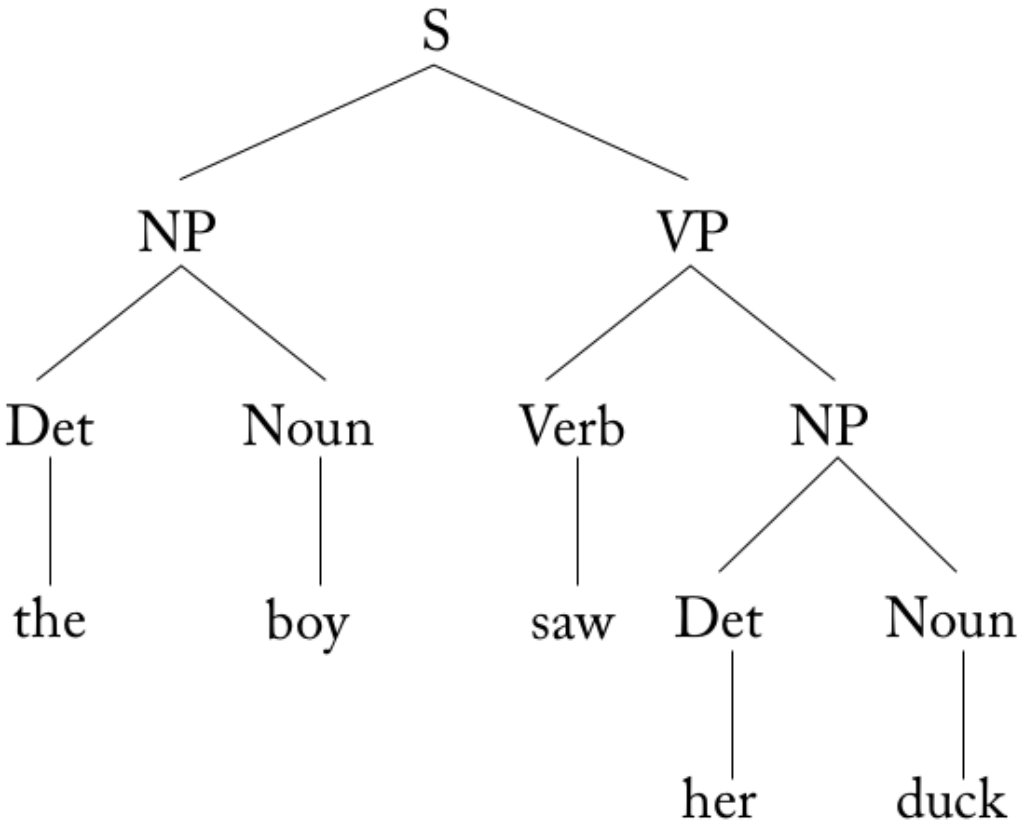


Figure: A parse tree

Recursive Neural Networks over Sentiment Treebank

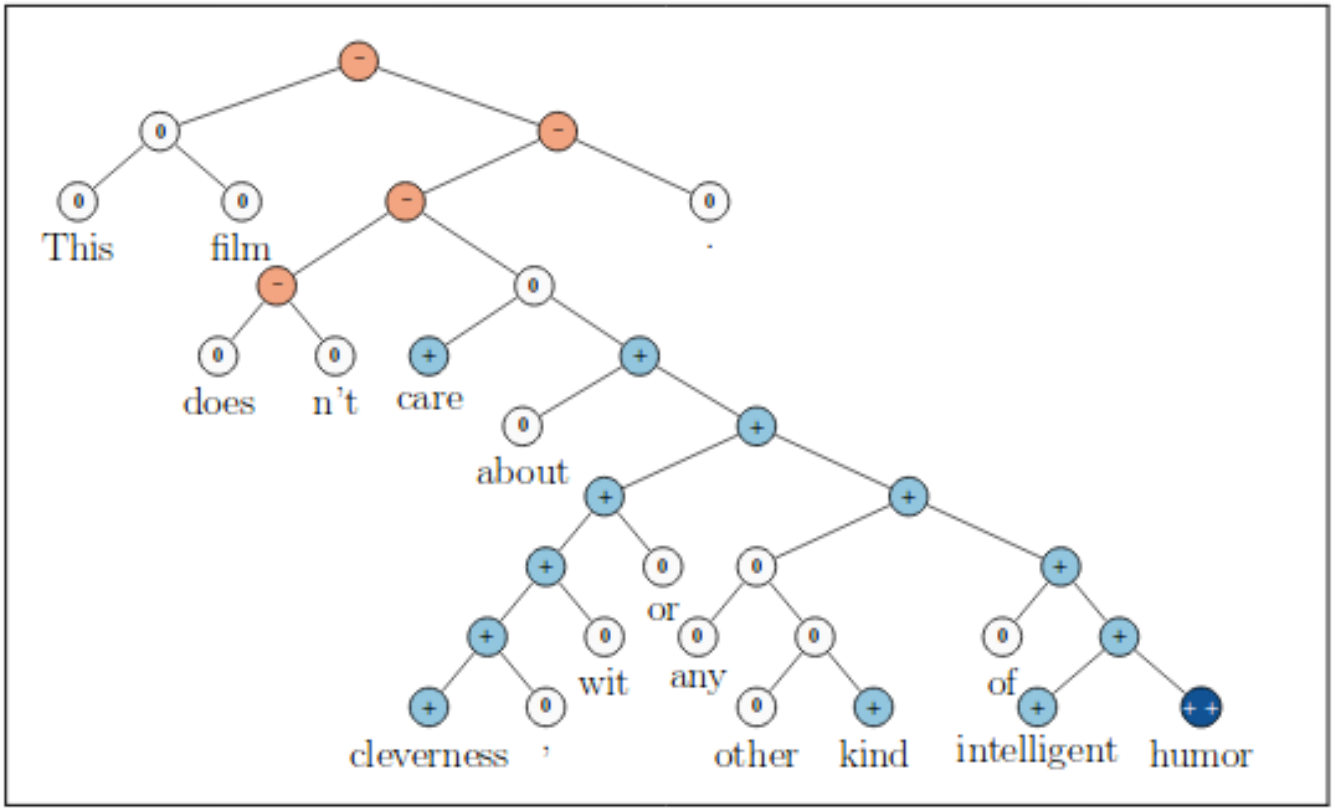


Figure 1: Example of the Recursive Neural Tensor Network accurately predicting 5 sentiment classes, very negative to very positive (– –, –, 0, +, + +), at every node of a parse tree and capturing the negation and its scope in this sentence.

Recursive Neural Networks over Sentiment Treebank

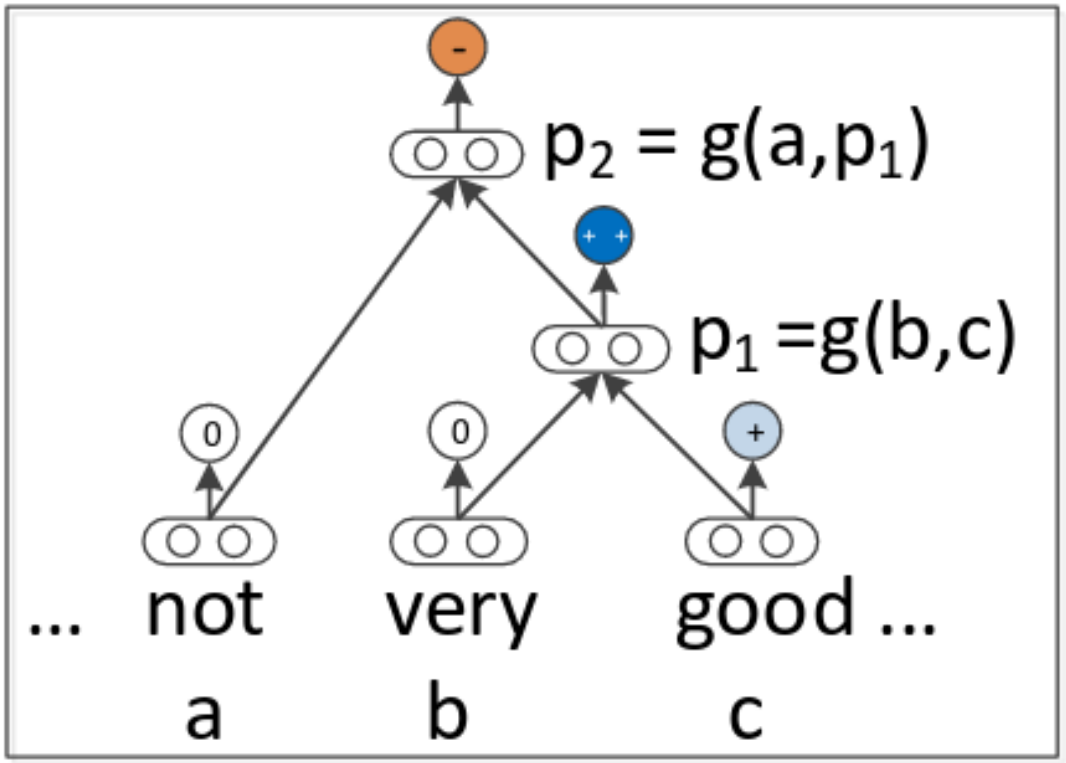


Figure 4: Approach of Recursive Neural Network models for sentiment: Compute parent vectors in a bottom up fashion using a compositionality function g and use node vectors as features for a classifier at that node. This function varies for the different models.

Paragraph vector

- A paragraph vector-embedding model that learns vectors for sequences of words of arbitrary length (e.g, sentences, paragraphs, or documents) without relying on parsing was proposed in [Le and Mikolov, 2014].
- The paragraph vectors are obtained by training a similar network as the one used for training the CBOW embeddings.
- The words surrounding a centre word in a window are used as input together with a paragraph-level vector for predict the centre word.
- The paragraph-vector acts as a memory token that is used for all the centre words in the paragraph during the training the phase.
- The recursive neural tensor network and the paragraph-vector embedding were evaluated on the same movie review dataset used in [Pang et al., 2002], obtaining an accuracy of 85.4% and 87.8%, respectively.
- Both models outperformed the results obtained by classifiers trained on representations based on bag-of-words features.
- Many researchers have have struggled to reproduce these paragraph vectors [Lau and Baldwin, 2016].

Paragraph vector

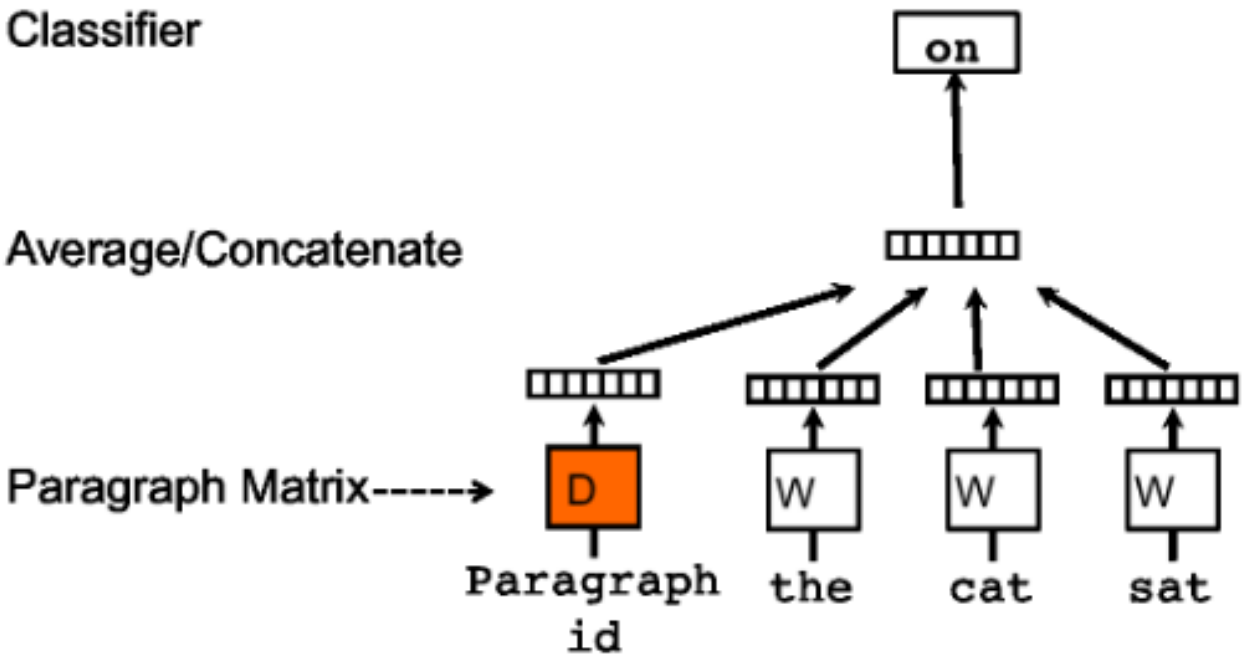


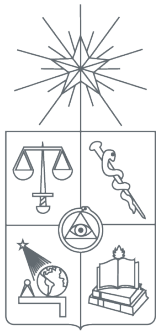
Figure 2. A framework for learning paragraph vector. This framework is similar to the framework presented in Figure 1; the only change is the additional paragraph token that is mapped to a vector via matrix D. In this model, the concatenation or average of this vector with a context of three words is used to predict the fourth word. The paragraph vector represents the missing information from the current context and can act as a memory of the topic of the paragraph.

Summary

- Neural networks are making improvements across many NLP tasks (e.g., sentiment analysis, machine translation).
- Deep Learning ! $=$ Feature Engineering.
- Word embeddings provide a practical framework for semi-supervised learning (i.e., leveraging unlabeled data).
- Character-level embeddings are worth paying attention to!
- Convolutional neural networks can capture useful features (e.g., n-grams) regardless of the position.
- Recurrent Neural Networks are very useful for learning temporal patterns, especially for long dependencies.
- We just scratched the surface!!

Questions?

Thanks for your Attention!



dcc CIENCIAS DE LA COMPUTAC
UNIVERSIDAD DE CHILE

References I


Amir, S., Ling, W., Astudillo, R., Martins, B., Silva, M. J., and Trancoso, I. (2015).
Inesc-id: A regression model for large scale twitter sentiment lexicon induction.
In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, pages 613–618, Denver, Colorado. Association for Computational Linguistics.


Baroni, M., Dinu, G., and Kruszewski, G. (2014).
Don't count, predict! a systematic comparison of context-counting vs. context-predicting semantic vectors.
In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pages 238–247. Association for Computational Linguistics.


Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. (2016).
Enriching word vectors with subword information.
arXiv preprint arXiv:1607.04606.


Brown, P. F., Desouza, P. V., Mercer, R. L., Pietra, V. J. D., and Lai, J. C. (1992).
Class-based n-gram models of natural language.
Computational linguistics, 18(4):467–479.


Cho, K. (2015).
Natural language understanding with distributed representation.
arXiv preprint arXiv:1511.07916.

References II


Cho, K., Courville, A., and Bengio, Y. (2015).
Describing multimedia content using attention-based encoder-decoder networks.
IEEE Transactions on Multimedia, 17(11):1875–1886.


Conneau, A., Schwenk, H., Barrault, L., and Lecun, Y. (2017).
Very deep convolutional networks for text classification.
In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 1107–1116.


Felbo, B., Mislove, A., Søgaard, A., Rahwan, I., and Lehmann, S. (2017).
Using millions of emoji occurrences to learn any-domain representations for detecting sentiment, emotion and sarcasm.
In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing, EMNLP 2017, Copenhagen, Denmark, September 9-11, 2017*, pages 1615–1625.


Goldberg, Y. (2016).
A primer on neural network models for natural language processing.
J. Artif. Intell. Res.(JAIR), 57:345–420.

References III


Goldberg, Y. and Levy, O. (2014).
word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method.
arXiv preprint arXiv:1402.3722.


Harris, Z. (1954).
Distributional structure.
Word, 10(23):146–162.


Jurafsky, D. and Martin, J. H. (2008).
Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition.
Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.


Lau, J. H. and Baldwin, T. (2016).
An empirical evaluation of doc2vec with practical insights into document embedding generation.
arXiv preprint arXiv:1607.05368.


Le, Q. V. and Mikolov, T. (2014).
Distributed representations of sentences and documents.
In *Proceedings of the 31th International Conference on Machine Learning*, pages 1188–1196.

References IV


LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998).
Gradient-based learning applied to document recognition.
Proceedings of the IEEE, 86(11):2278–2324.


Levy, O. and Goldberg, Y. (2014).
Neural word embedding as implicit matrix factorization.
In *Advances in neural information processing systems*, pages 2177–2185.


Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013).
Distributed representations of words and phrases and their compositionality.
In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K., editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. Curran Associates, Inc.


Mohammad, S. M., Kiritchenko, S., and Zhu, X. (2013).
Nrc-canada: Building the state-of-the-art in sentiment analysis of tweets.
Proceedings of the seventh international workshop on Semantic Evaluation Exercises (SemEval-2013).

References V


Nakov, P., Rosenthal, S., Kozareva, Z., Stoyanov, V., Ritter, A., and Wilson, T. (2013).
Semeval-2013 task 2: Sentiment analysis in twitter.
In *Proceedings of the seventh international workshop on Semantic Evaluation Exercises*, pages 312–320, Atlanta, Georgia, USA. Association for Computational Linguistics.


Pang, B., Lee, L., and Vaithyanathan, S. (2002).
Thumbs up? Sentiment classification using machine learning techniques.
In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 79–86. Association for Computational Linguistics.


Pennington, J., Socher, R., and Manning, C. D. (2014).
Glove: Global vectors for word representation.
In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1532–1543.


Read, J. (2005).
Using emoticons to reduce dependency in machine learning techniques for sentiment classification.
In *Proceedings of the ACL Student Research Workshop*, ACLstudent '05, pages 43–48, Stroudsburg, PA, USA. Association for Computational Linguistics.

References VI


Severyn, A. and Moschitti, A. (2015).
Twitter sentiment analysis with deep convolutional neural networks.
In *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 959–962, New York, NY, USA. ACM.


Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., and Potts, C. (2013).
Recursive deep models for semantic compositionality over a sentiment treebank.
In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1631–1642. Association for Computational Linguistics.


Tang, D., Wei, F., Qin, B., Zhou, M., and Liu, T. (2014).
Building large-scale twitter-specific sentiment lexicon : A representation learning approach.
In *COLING 2014, 25th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, August 23-29, 2014, Dublin, Ireland*, pages 172–182.

References VII

Turian, J., Ratinov, L., and Bengio, Y. (2010).
Word representations: a simple and general method for semi-supervised learning.
In *Proceedings of the 48th annual meeting of the association for computational linguistics*, pages 384–394. Association for Computational Linguistics.