A Handbook of Natural Language Processing

M. Şafak Bilici

safak@inzva.com

a draft

Contents

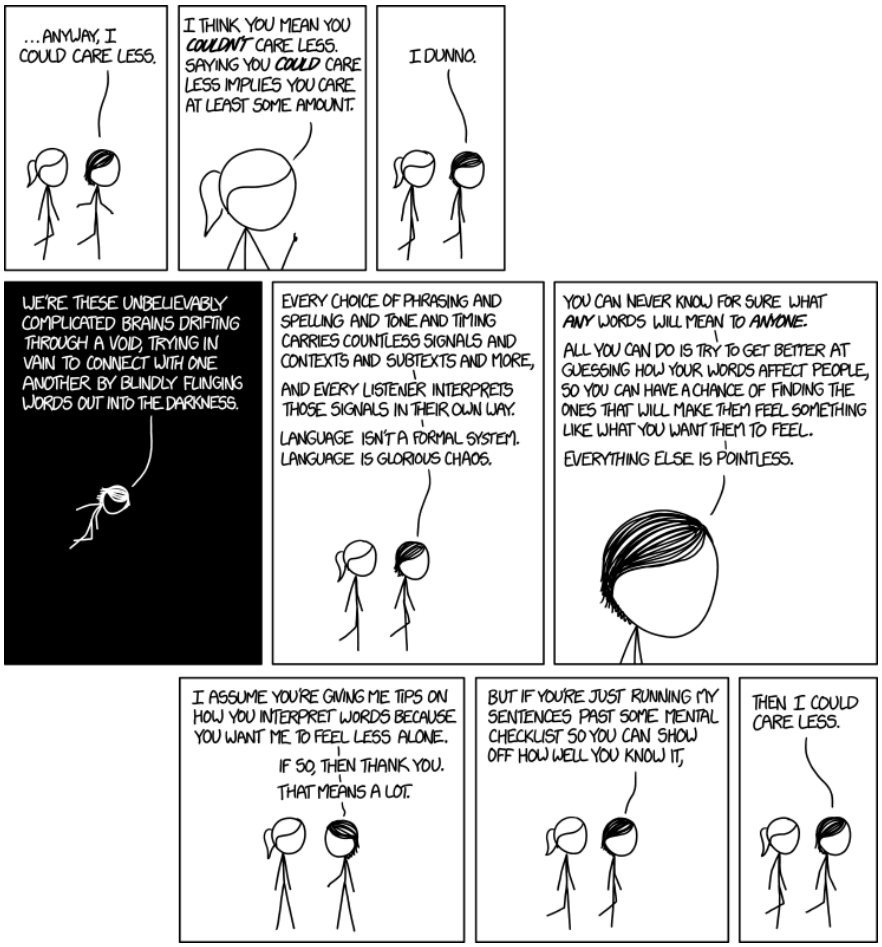


Figure 1: https://xkcd.com/1576/

1 Word Vectors

Words and meanings are ambigious. ∼85% of words are unambigious, however, accounting for only ∼15% of the vocabulary, are very common words, and hence ∼55% of word tokens in running text are ambigious [1]. Even though many words are easy to disambiguate linguistically, it is not always easy to represent this words computationally. Grammar is a mental system, a cognitive part of the brain/mind, which, if it is one's first native language, is acquired as a child without any specific instruction. Children develop language rapidly and efficiently, that is, with relatively few errors, because the basic form of language is given to them by human biology (the logical problem of language acquisition, Noam Chomsky, 1955). So, imagine that how hard it is to represent well while we have ambiguity in our brain's understanding. On account of this, a book which is about modern methods in NLP should be concerned about "Language And Representation" at first.

1.1 Word Representation With One-Hot Vectors

One-hot encoding is the most common, most basic way to turn a token into a vector. It consists in associating a unique integer index to every word, then turning this integer index i into a binary vector of size V, which is the size of our vocabulary, that would be all-zeros except for the i-th entry, which would be 1.
There are two major issues with this approach. First issue is the curse of dimensionality, which refers to all sorts of problems that arise with data in high dimensions. This requires exponentially large memory space. Most of the matrix is taken up by zeros, so useful data becomes sparse. Imagine we have a vocabulary of 50,000. (There are roughly a million words in English language.) Each word is represented with 49,999 zeros and a single one, and we need 50,000 squared = 2.5 billion units of memory space. Not computationally efficient.

$$\text{cos-sim}(\vec{u_1}, \vec{u_2}) = \frac{\langle \vec{u_1}, \vec{u_2} \rangle}{\|\vec{u_1}\|_2 \times \|\vec{u_2}\|_2} = \frac{\sum_{i=0}^{n} u_{1_i} \times u_{2_i}}{\sqrt{\sum_{i=0}^{n} u_{1_i}^2} \times \sqrt{\sum_{i=0}^{n} u_{2_i}^2}} = \frac{1 \times 0 + 0 \times 1 + 0 \times 0}{1 + 1} = 0 \qquad (1)$$

So, how to deal with those problems?

1.2 Lexical Semantics and Distributional Linguistics

Words that occur in **similar contexts** tend to have similar meanings. The link between similarity and words are distributed and similarity in what they mean is called **distributional hypothesis** or **distributional semantics** in the field of Computational Linguistics [1]. So what can be counted when we say similar contexts? For example if you surf on the Wikipedia page of linguistics, the words in this page somehow related with each other in the context of linguistics. This was formulated firsty by [2], [3], [4]. Some words have similar meanings, for example word cat and dog **similar**. Also, words can be **antonyms**, for example hot and cold. And words have **connotations** (TR: çağrışım), for example happy→positive connotation and sad→negative connotation. Can you feel the similarity of words, [study, exam, night, FF]?

Also each word can have multiple meanings. The word mouse can refeer to the rodent or the cursor control device. We call each of these aspects of the meaning of mouse a **word sense**. In other words words can be **polysemous** (have multiple senses), which can lead us to make word interpretations difficult!

- **Word Sense Disambiguation**: "Mouse info" (person who types this into a web search engine, looking for a pet info or a tool?) (determining which sense of a word is being used in a particular context)

The word **similarity** is very useful in larger semantics tasks. Knowing how similar two words are can help in computing how similar the meaning of two phrases or sentences are, a very important component of natural language understanding tasks like **question answering**, **summarization** etc.

Word1	**Word2**	**Similarity (0-10)**
Vanish	Disappear	9.8
Behave	Obey	7.3
Belief	Impression	5.95
Muscle	Bone	3.65
Modest	Flexible	0.98
Hole	Agreement	0.3

Table 1: [1]

We should look at **word relatedness** , the meaning of two words can be related in ways other than similarity. One such class of connections is called word **relatedness**, also traditionally called word **association** in pysch.

- The word *cup* and *coffee*.
- The word *inzva* and *deep learning*.

Also words can affective meanings. Osgood et al. 1957, proposed that words varied along three important dimensions of affective meaning: **valence, arousal, dominance** [1]:

1. **valence**: the pleasantness of the stimulus.
2. **arousal**: the intensity of emotion provoked by the stimulus.
3. **dominance**: the degree of control exerted by the stimulus.

Examples:

- happy(1) ↑, satisfied(1) ↑; annoyed(1) ↓, unhappy(1)↓
- excited(2) ↑, frenzied(2) ↑; relaxed(1) ↓, calm(2) ↓
- important(3)↑, controlling(3)↑; awed(3)↓, influenced(3)↓

Question: does word embeddings has these dimesions?

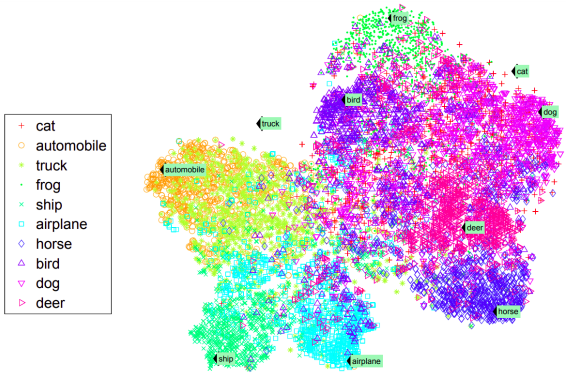
1.3 Word Embeddings

How can we build a computational model that successfully deals with the different aspects of word meaning we saw above (word senses, word similarity, word relatedness, connotation etc.)? Instead of representing words with one-hot vector, sparse; with word embeddings we represents words with dense vectors. The idea of vector semantics is thus to represent a word as a point in some multidimensional semantic space. Vectors for representing words are generally called **embeddings**. We learn this embeddings form an arbitrary context. Main two advantages of this word embeddings are:

- Now we represent words with dense vectors, which leads low memory requirements.
- We can now calculate similarity metrics on these vectors!

1.3.1 Visualizing Word Embeddings

Word embeddings can be visualized with various dimensionality reduction/matrix factorization algorithms.



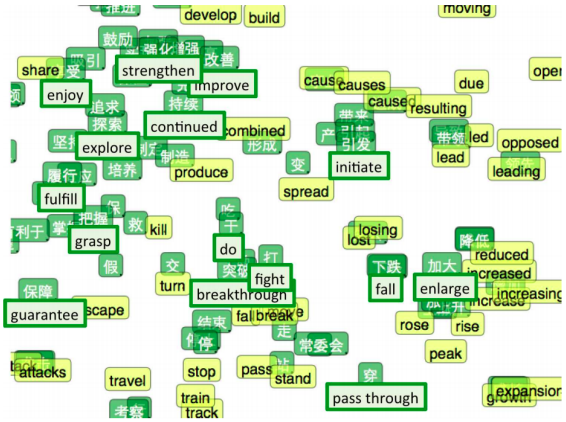
(a) image source: [5]

(b) image source: [6]

Figure 2: t-SNE

FRANCE	JESUS	XBOX	REDDISH	SCRATCHED	MEGABITS
454	1973	6909	11724	29869	87025
AUSTRIA	GOD	AMIGA	GREENISH	NAILED	OCTETS
BELGIUM	SATI	PLAYSTATION	BLUISH	SMASHED	MB/S
GERMANY	CHRIST	MSX	PINKISH	PUNCHED	BIT/S
ITALY	SATAN	IPOD	PURPLISH	POPPED	BAUD
GREECE	KALI	SEGA	BROWNISH	CRIMPED	CARATS
SWEDEN	INDRA	PSNUMBER	GREYISH	SCRAPED	KBIT/S
NORWAY	VISHNU	HD	GRAYISH	SCREWED	MEGAHERTZ
EUROPE	ANANDA	DREAMCAST	WHITISH	SECTIONED	MEGAPIXELS
HUNGARY	PARVATI	GEFORCE	SILVERY	SLASHED	GBIT/S
SWITZERLAND	GRACE	CAPCOM	YELLOWISH	RIPPED	AMPERES

(a) image source: [7]



(b) image source: [8]

Figure 3: t-SNE

1.4 Semantic & Syntactic Properties of Embeddings

Word embeddings have very important property: analogies. Analogy is another semantic property of embeddings that can capture relational meanings. Simply in words, analogy is to find **X**:

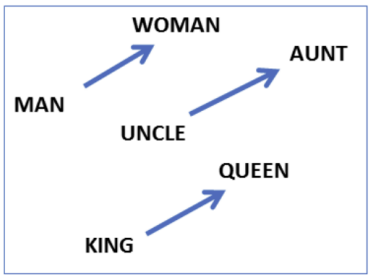
- **A is to B as C is to X**

For example **"woman is to queen as man is to X**. In this example **X** should be the word **king**.

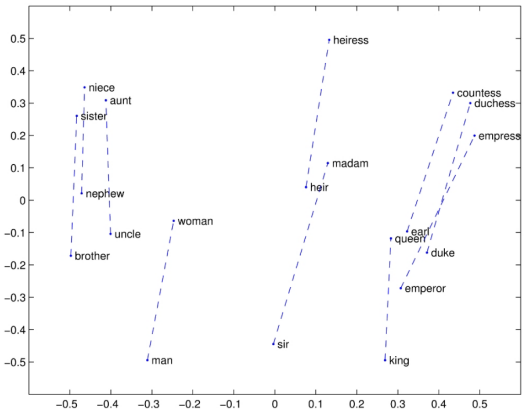
Interestingly, such embeddings exhibit seemingly linear behaviour in analogies. This linear behaviour can be formulated as

- w_a is to w'_a as w_b is to $w'_b \rightarrow\rightarrow\rightarrow w'_a - w_a + w_b \approx w'_b$
- vec('*queen*') - vec('*woman*') + vec('*man*') $\approx$ vec('*king*')

- or
- vec('*queen*') - vec('*woman*') ≈ vec('*king*') - vec('textitman')
- Another example:
- vec('*Paris*') - vec('*France*') ≈ vec('*Italy*') - vec('*Rome*')



(a) image source: [9]



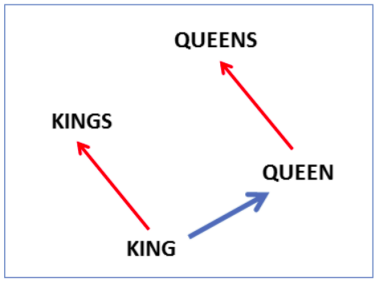
(b) image source: [1]

Relationship	Example 1	Example 2	Example 3
France - Paris	Italy: Rome	Japan: Tokyo	Florida: Tallahassee
big - bigger	small: larger	cold: colder	quick: quicker
Miami - Florida	Baltimore: Maryland	Dallas: Texas	Kona: Hawaii
Einstein - scientist	Messi: midfielder	Mozart: violinist	Picasso: painter
Sarkozy - France	Berlusconi: Italy	Merkel: Germany	Koizumi: Japan
copper - Cu	zinc: Zn	gold: Au	uranium: plutonium
Berlusconi - Silvio	Sarkozy: Nicolas	Putin: Medvedev	Obama: Barack
Microsoft - Windows	Google: Android	IBM: Linux	Apple: iPhone
Microsoft - Ballmer	Google: Yahoo	IBM: McNealy	Apple: Jobs
Japan - sushi	Germany: bratwurst	France: tapas	USA: pizza

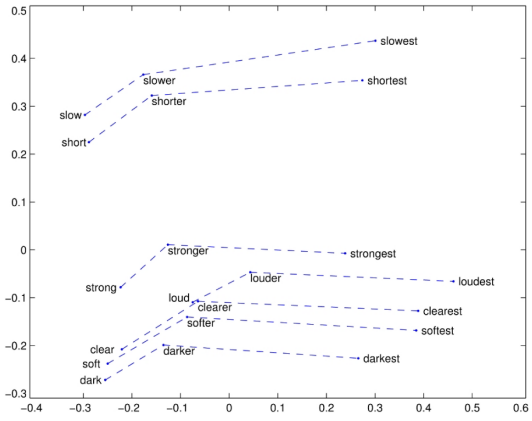
(c) image source: [10]

Figure 4: Semantic Properties Of Embeddings

Do vector embeddings capture syntactic relationships? Yes. Capture with linear behaviours? Yes.



(a) Stanford cs-230 cheatsheet



(b) image source: [1]

We refer the reader to [11] for a detailed study on the properties word vectors and analogy.

1.5 Evaluating Word Vectors

- Extrinsic Evaluation
  - This is the evaluation on a real task.
  - Can be slow to compute performance.
  - Unclear if the subsystem is the problem, or our system.
  - If replacing subsystem improves performance, the change is likely good.
  - NER, Question Answering etc.
- Intrinsic Evaluation
  - Fast to compute
  - Helps to understand subsystem
  - Needs positive correlation with real task to determine usefulness

SimLex-999 dataset [12] gives values on a scale from 0 to 10 by asking humans to judge how similar one word is to another. Other datasets for evaluating word vectors:

- WordSim 353
- TOEFL Dataset
- SCWS Dataset
- Word-in-Context (WiC)
- Miller & Charles Dataset
- Rubenstein & Goodenough Dataset
- Stanford Rare Word (RW)

1.6 Learning Word Embeddings

1.6.1 Embedding Matrix

Embedding matrix can be represented as a single matrix $E \in \mathbb{R}^{d \times |V|}$ (or $E \in \mathbb{R}^{|V| \times d}$, it does not even matter), where d is embedding size and $\mid V \mid$ is size of the vocabulary V.

$$E = \begin{bmatrix} \text{hello} & \text{i} & \text{love} & \text{inzva} & \cdots & \text{sanctuary} \\ 0.1 & 0.137 & -0.03 & -0.44 & \cdots & 0.36 \\ -0.78 & -0.25 & 2.09 & 0.19 & \cdots & 0.32 \\ -3.1 & 1.54 & -2.52 & 0.2 & \cdots & -1.51 \\ 1.13 & -0.78 & -0.56 & 0.95 & \cdots & 0.2112 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -0.6 & 0.13 & 3.89 & -0.071 & -0.27 & 0.27 \end{bmatrix} \in \mathbb{R}^{d \times |V|} \tag{2}$$

But how to learn them?

1.6.2 A Neural Probabilistic Language Model

In neural language models, the prior context is represented by embeddings of the previous words. Representing the prior context as embeddings, rather than by exact words as used in models, allows neural language models to generalize to unseen data much better than n-gram language models [1].

For example in our training set we see this sentence,

Today, after school, I am planning to go to cinema.

but we have never seen the word "concert" after the words "go to". In our test set we are trying to predict what comes after the prefix "Today, after school, I am planning to go to". An n-gram language model will predict "cinema" but not "concert". But a neural language model, which can make use of the fact that "cinema" and "concert" have similar embeddings, will be able to assign a reasonably high probability to "concert" as well as "cinema", merely because they have similar vectors.

[13] proposed a neural language model that can learns and uses embeddings to predict the next word in a sentence. Formally representation, for a sequence of words

$$x^{(1)}, x^{(2)}, ..., x^{(t)} \tag{3}$$

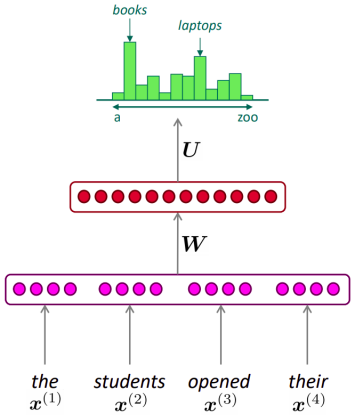
the probability distribution of the next word (output) is

$$p(x^{(t+1)} \mid x^{(1)}, x^{(2)}, ..., x^{(t)}) \tag{4}$$

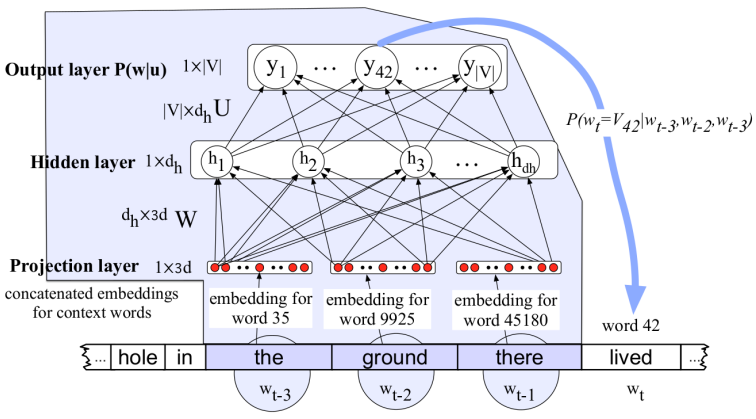
Bengio proposed a fixed-window neural Language Model which can be seen as the same approach of n-grams.

"~~as the proctor started the clock~~ the students opened their....."

We have a moving window at time t with an embedding vector representing each of the window size previous words. For window size 3, words $w_{t-1}, w_{t-2}, w_{t-3}$. These 3 vectors are concatenated together to produce input x. The task is to predict w_t.



(c) CS224n lecture 5



(d) Speech and Language Processing, Daniel Jurafsky, Third Edition

Figure 5: Neural Probabilistic Language Model

For this task, we are going to represent each of the N prevuios words as a one-hot-vector of length $\mid V \mid$. The forward equation for neural language model

- Input $x_i \in R^{1\times|V|}$
- Learning word embeddings: $e = concat(x_1E^T, x_2E^T, x_3E^T)$
  - $E \in \mathbb{R}^{d\times|V|}$
  - $e \in \mathbb{R}^{1\times 3d}$
- $h = \sigma(eW^T + b_1)$
  - $W \in \mathbb{R}^{d_h\times 3d}$
  - $h \in \mathbb{R}^{1\times d_h}$
- $z = hU^T + b_2$
  - $U \in \mathbb{R}^{|V|\times d_h}$
  - $z \in \mathbb{R}^{1\times|V|}$
- $\hat{y} = softmax(z) \in \mathbb{R}^{1\times|V|}$

Then the model is trained, at each word w_t, the negative log likelihood loss is:

$$L = -\log p(w_t \mid w_{(t-1)}, w_{(t-2)}, ..., w_{(t-n+1)}) = softmax(z) \tag{5}$$

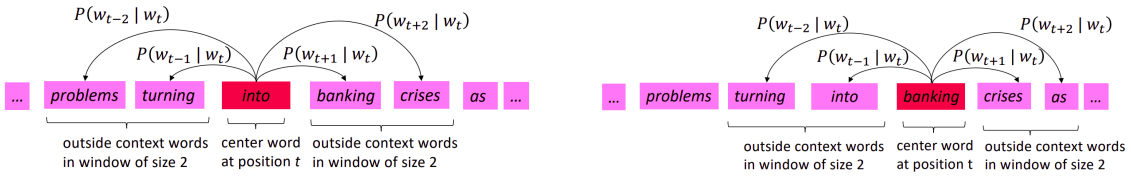
$$\theta_{t+1} = \theta_t - \eta\frac{\partial -\log p(w_t \mid w_{(t-1)}, w_{(t-2)}, ..., w_{(t-n+1)})}{\partial\theta} \tag{6}$$

1.6.3 word2vec

word2vec is proposed in [14]. It allows you to learn the high-quality distributed vector representations that capture a large number of precise syntactic and semantic word relationships. Word2vec algorithm uses Skip-gram model [10] to learn efficient vector representations. Those learned word vectors has interesting property, words with semantic and syntactic affinities give the necessary result in mathematical similarity operations.

Suppose that you have a sliding window of a fixed size moving along a sentence: the word in the middle is the "target" and those on its left and right within the sliding window are the context words.

The skip-gram model is trained to predict the probabilities of a word being a context word for the given target.



(a) Learning Word Embedding (b) Learning Word Embedding

Figure 6: Target - Context pairs.

For example consider this sentence,

"A change in Quantity also entails a change in Quality"

Our target and context pairs for window size of 5:

Sliding window (size = 5)	**Target word**	**Context**
[A change in]	a	change, in
[A change in Quantity]	change	a, in, quantitiy
[A change in Quantity also]	in	a, change, quantitiy,also
...	...	...
[entails a change in Quality]	change	entails, a, in, Quality
[a change in Quality]	in	a, change, Quality
[change in Quality]	quality	change, in

Each context-target pair is treated as a new observation in the data.

For each position $t = 1, .., T$ predict context words within a window of fixed size m, given center word w_j. In Skip-gram connections we have an objective to maximize, likelihood (or minimize log-likelihood):

$$\max_{\theta} \prod_{\text{center}} \prod_{\text{context}} p(\text{context}|\text{center}; \theta) \tag{7}$$

$$= \max_{\theta} \prod_{t=1}^{T} \prod_{-c \leq j \leq c, j \neq c} p(w_{t+j}|w_t; \theta) \tag{8}$$

$$= \min_{\theta} -\frac{1}{T} \prod_{t=1}^{T} \prod_{-c \leq j \leq c, j \neq c} p(w_{t+j}|w_t; \theta) \tag{9}$$

$$= \min_{\theta} -\frac{1}{T} \sum_{t=1}^{T} \sum_{-c \leq j \leq c, j \neq c} \log p(w_{t+j}|w_t; \theta) \tag{10}$$

So, how can we calculate those probabilities? Softmax gives the normalized probabilities.

1.6.3.1 Parameterization of Skip-Gram Model

Let's say w_t is our target word and w_c is current context word. The softmax is defined as

$$p(w_c \mid w_t) = \frac{\exp(v_{w_c}^T v_{w_t})}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \tag{11}$$

maximizing this log-likelihood function under v_{w_t} gives you the most likely value of the v_{w_t} given the data.

$$\frac{\partial}{\partial v_{w_t}} \cdot \log \frac{\exp(v_{w_c}^T v_{w_t})}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \tag{12}$$

$$= \frac{\partial}{\partial v_{w_t}} \cdot \log \underbrace{\exp(v_{w_c}^T v_{w_t})}_{\text{numerator}} - \frac{\partial}{\partial v_{w_t}} \cdot \log \underbrace{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})}_{\text{denominator}} \tag{13}$$

$$\frac{\partial}{\partial v_{w_t}} \cdot v_{w_c}^T v_{w_t} = v_{w_c} \quad \text{(numerator)} \tag{14}$$

Now, it is time to derive denominator.

$$\frac{\partial}{\partial v_{w_t}} \cdot \log \sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t}) = \frac{1}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \cdot \frac{\partial}{\partial v_{w_t}} \sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t}) \tag{15}$$

$$= \frac{1}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \cdot \sum_{i=0}^{|V|} \frac{\partial}{\partial v_{w_t}} \cdot \exp(v_{w_i}^T v_{w_t}) \tag{16}$$

$$= \frac{1}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \cdot \sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t}) \frac{\partial}{\partial v_{w_t}} v_{w_i}^T v_{w_t} \tag{17}$$

$$= \frac{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t}) \cdot v_{w_i}}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \text{ (denominator)} \tag{18}$$

To sum up,

$$\frac{\partial}{\partial w_t} \log p(w_c \mid w_t) = v_{w_c} - \frac{\sum_{j=0}^{|V|} \exp(v_{w_j}^T v_{w_t}) \cdot v_{w_j}}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \tag{19}$$

$$= v_{w_c} - \sum_{j=0}^{|V|} \frac{\exp(v_{w_j}^T v_{w_t})}{\sum_{i=0}^{|V|} \exp(v_{w_i}^T v_{w_t})} \cdot v_{w_j} \tag{20}$$

$$\underbrace{= v_{w_c} - \sum_{j=0}^{|V|} p(w_j \mid w_t) \cdot v_{w_j}}_{\nabla_{w_t} \log p(w_c|w_t)} \tag{21}$$

This is the observed representation subtract $\mathbb{E}[w_j \mid w_t]$.

1.6.3.2 Negative Sampling (Noise Contrastive Estimation (NCE))

The Noise Contrastive Estimation (NCE) metric intends to differentiate the target word from noise samples using a logistic regression classifier [15].

In softmax computation, look at the denominator. The summation over $\mid V \mid$ is computationally expensive. The training or evaluation takes asymptotically $O(\mid V \mid)$. In a very large corpora, the most frequent words can easily occur hundreds or millions of times ("in", "and", "the", "a" etc.). Such words provides less information value than the rare words. For example, while the skip-gram model benefits from observing co-occurences of "inzva" and "deep learning", it benefits much less from observing the frequent co-occurences of "inzva" and "the".In a very large corpora, the most frequent words can easily occur hundreds or millions of times ("in", "and", "the", "a" etc.). Such words provides less information value than the rare words. For example, while the skip-gram model benefits from observing co-occurences of "inzva" and "deep learning", it benefits much less from observing the frequent co-occurences of "inzva" and "the".

For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples! We "sample" from a noise distribution $P_n(w)$ whose probabilities match the ordering of the frequency of the vocabulary.

Consider a pair (w_t, w_c) of word and context. Did this pair come from the training data? Let's denote by $p(D = 1 \mid w_t, w_c)$ the probability that (w_t, w_c) came from the corpus data. Correspondingly $p(D = 0 \mid w_t, w_c)$ will be the probability that (w_t, w_c) didn't come from the corpus data.

First, let's model $p(D = 1 \mid w_t, w_c)$ with sigmoid:

$$p(D = 1 \mid w_t, w_c) = \sigma(v_{w_c}^T v_{w_t}) = \frac{1}{1 + \exp(-v_{w_c}^T v_{w_t})} \tag{22}$$

Now, we build a new objective function that tries to maximize the probability of a word and context being in the corpus data if it indeed is, and maximize the probability of a word and context not being in the corpus data if it indeed is not. Maximum likelihood says:

$$\max \prod_{(w_t,w_c)\in D} p(D = 1 \mid w_t, w_c) \times \prod_{(w_t,w_c)\in D'} p(D = 0 \mid w_t, w_c) \tag{23}$$

$$= \max \prod_{(w_t,w_c)\in D} p(D = 1 \mid w_t, w_c) \times \prod_{(w_t,w_c)\in D'} 1 - p(D = 1 \mid w_t, w_c) \tag{24}$$

$$= \max \sum_{(w_t,w_c)\in D} \log p(D = 1 \mid w_t, w_c) + \sum_{(w_t,w_c)\in D'} \log(1 - p(D = 1 \mid w_t, w_c)) \tag{25}$$

$$= \max \sum_{(w_t,w_c)\in D} \log \frac{1}{1 + \exp(-v_{w_c}^T v_{w_t})} + \sum_{(w_t,w_c)\in D'} \log \left(1 - \frac{1}{1 + \exp(-v_{w_c}^T v_{w_t})}\right) \tag{26}$$

$$\tag{27}$$

Note that $\frac{\exp(-x)}{(1+\exp(-x))} \times \frac{\exp(x)}{\exp(x)} = \frac{1}{(1+\exp(x))}$

$$= \max \sum_{(w_t,w_c)\in D} \log \frac{1}{1 + \exp(-v_{w_c}^T v_{w_t})} + \sum_{(w_t,w_c)\in D'} \log \frac{1}{1 + \exp(v_{w_c}^T v_{w_t})} \tag{28}$$

Maximizing the likelihood is the same as minimizing the negative log likelihood:

$$L = - \sum_{(w_t,w_c)\in D} \log \frac{1}{1 + \exp(-v_{w_c}^T v_{w_t})} - \sum_{(w_t,w_c)\in D'} \log \frac{1}{1 + \exp(v_{w_c}^T v_{w_t})} \tag{29}$$

The Negative Sampling (NEG) proposed in the original word2vec paper. NEG approximates the binary classifier's output with sigmoid functions as follows:

$$p(d = 1|v_{w_c}, v_{w_t}) = \sigma(v_{w_c}^T v_{w_t}) \tag{30}$$

$$p(d = 0|v_{w_c}, v_{w_t}) = 1 - \sigma(v_{w_c}^T v_{w_t}) = \sigma(-v_{w_c}^T v_{w_t}) \tag{31}$$

So the objective is

$$L = -[\log \sigma(v_{w_c}^T v_{w_t}) + \sum_{\substack{i=1 \\ \tilde{w}_i \sim Q}}^{K} \log \sigma(v_{\tilde{w}_i}^T v_{w_t})] \tag{32}$$

In the above formulation, $v_{\tilde{w}_i} \mid i = 1...K$ are sampled from $P_n(w)$. How to define $P_n(w)$? In the word2vec paper $P_n(w)$ defined as

$$P_n(w_i) = 1 - \sqrt{\frac{t}{freq(w_i)}} \quad t \approx 10^{-5} \tag{33}$$

This distribution assigns lower probability for lower frequency words, higher probability for higher frequency words.

Hence, this distribution is sampled form a unigram distribution $U(w)$ raised to the $\frac{3}{4}$rd power. The unigram distribuiton is defined as

$$P_n(w) = \left(\frac{U(w)}{Z} \right)^{\alpha} \tag{34}$$

Or just by Andrew NG's definition:

$$P_n(w_i) = \frac{freq(w_i)^{\frac{3}{4}}}{\sum_{j=0}^{M} freq(w_j)^{\frac{3}{4}}} \tag{35}$$

Raising the unigram distribution $U(w)$ to the power of α has an effect of smoothing out the distribution. It attempts to combat the imbalance between common words and rare words by decreasing the probability of drawing common words, and increasing the probability drawing rare words.

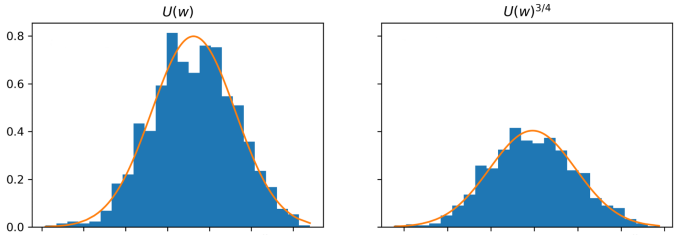


Figure 7

1.6.3.3 Hierarchical Softmax

Hierarchical softmax is proposed to make the sum calculation faster with the help of a binary tree structure. The model uses a binary tree to represent all words in the vocabulary. The $\mid V \mid$ words must be leaf units of the tree.

Hieararchical softmax is a technique for **aproximating** naive softmax. Hieararchical softmax uses binary tree, all nodes have 2 children, and is organized like Huffmann Tree. Rare words are down at deeper levels and frequent words are at shallower levels.

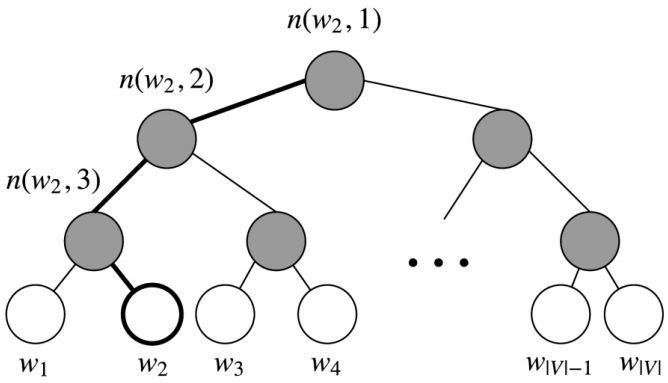


Figure 8: Hierarchical Softmax

Let's introduce some notation. Let $L(w)$ be the number of nodes in the path from the root to the leaf w. For instance, above figure, $L(w_2)$ is 3. Let's write $n(w, i)$ as the i-th node on this path

with associated vector $v_{n(w,i)}$. Now for each inner node n, we arbitrarily choose one of its children and call it $ch(n)$ (e.g. always the left node). Then, we can compute the probability as

$$P(w \mid w_i) = \prod_{j=0}^{L(w)-1} \sigma([n(w, j+1) == ch(n(w, j))] \cdot v_{n(w,j)}^T v_{w_i}) \tag{36}$$

First, we are computing a product of terms based on the shape of the path from the root $n(w, i)$ to leaf (w). If we assume $ch(n)$ is always the left node of n, then term $[n(w, j+1) == ch(n(w, j))]$ returns 1 when the path goes left, and -1 if right. Furthermore, the term $[n(w, j+1) == ch(n(w, j))]$ provides normalization. At a node n, if we sum the probabilities for going to the left and right node, you can check that for any value of $v_{n(w,j)}^T v_{w_i}$

$$\sigma(v_{n(w,j)}^T v_{w_i}) + \sigma(-v_{n(w,j)}^T v_{w_i}) = 1 \tag{37}$$

Finally, we compare the similarity of our input vector v_{w_i} to each inner node vector $v_{n(w,j)}^T$ using a dot product.

Taking w_2 in Figure we must take two left edges and then a right edge to reach w_2 from the root, so

$$P(w_2 \mid w_i) = p(n(w_2, 1), left) \cdot p(n(w_2, 2), left) \cdot p(n(w_2, 3), right) \tag{38}$$

$$= \sigma(v_{n(w,1)}^T v_{w_i}) \cdot \sigma(v_{n(w,2)}^T v_{w_i}) \cdot \sigma(-v_{n(w,3)}^T v_{w_i}) \tag{39}$$

To train the model, our goal is still to minimize the negative log likelihood $-\log P(w \mid w_i)$.

So, instead $O(\mid V \mid)$, our complexity is the depth of the tree, $O(\log \mid V \mid)$, in worst-case.

1.6.4 GloVe: Global Vectors for Word Representation

GloVe [16] combines the advantages of global matrix factorization (such as Latent Semantic Analysis) and local context window methods. Methods like skip-gram (word2vec) may do better on the analogy task, but they poorly utilize the statistics of the corpus since they train on separate local context windows insted of on global co-occurence counts. GloVe defines matrix $\mathbf{X} = [X_{ij}]$ which represents word-word co-occurence counts. Entries X_{ij} tabluate the number of times word j occurs in the context of word i. If we choose our context-target pairs as in word2vec algorithm, then $\mathbf{X}$ would be symmetric. If we choose our context-target pairs in one direction (left or right), then $X_{ij} \neq X_{ji}$. The objective function of GloVe comes from a relation between symmetry and homomorphism between $(\mathbb{R}, +)$ and $(\mathbb{R}_{>0}, \times)$. Since this is not scope of the course, we will evaluate the objective function intuitively. The objective function of GloVe is defined as

$$J = \sum_{i,j=0}^{|V|} f(X_{ij})(w_i^T \tilde{w_j} + b_i + \tilde{b_j} - \log X_{ij})^2$$

i and j are playing the role of target and context. The term $w_i^T \tilde{w_j} - \log X_{ij}$ says that "how related are those two words?" as measured by how often they occur with each other. But what if words i and j never accour together? Then $X_{ij} = 0$, and $\log 0$ is not defined. So we need to define a weigthing term $f(X_{ij})$:

1. $f(0) = 0$. We are going to use a convention that "$0 \log 0 = 0$".
2. $f(x)$ should be non-decreasing so that rare word co-occurences are not overweighted.

3. $f(x)$ should be relatively small for large values of x, so that frequent co-occurences are not overweighted.

Yes there are a lot of functions that have characteristic like f. GloVe choose f as:

$$f(x) = \begin{cases} \left(\frac{x}{x_{\max}}\right)^{\alpha}, & \text{if } x < x_{\max} \\ 1, & \text{otherwise} \end{cases} \tag{40}$$

The most beautiful part of this algoirthm is that w_i and $\tilde{w}_j$ are symmetric. This the role of derivation of objective is same for w_i and $\tilde{w}_j$ when $i = j$. $\rightarrow$ The model generates two set of vectors, W and $\tilde{W}$. When X is symmetrix, W and $\tilde{W}$ are equivalent and differ only as a result of their random initializations. At final level, we are able to choose

$$\text{word}_k^{\text{final}} = \frac{w_k + \tilde{w_k}}{2}$$

1.6.5 fasttext

The methods that we have talked about are very effective to find good and dense word representations. However, handling not-seen words is hard. We train those models on a specific corpus, and language is infinite. Suppose that we have words "fast", "faster", "fastest" and "long", but "longer" and "longest" not seen in this corpus. After learning word representations, how can we represent "longer" and "longest" when they are not in the training set? The theory of Derivational Morphology helps us here. Morphology is basically internal structure of words and forms. Words are meaningful linguistics units that can be combined to form phrases and sentences. For example, look at the sentence "Bach composed the piece". The word "Bach" is a free morpheme (lexical morpheme), a free morpheme is a morpheme that can stand alone as a word. The word composed can be decomposed to "compose" and "-d". The suffix "-d" is a past marker here. In linguistics, we call "-d" grammatical morpheme or bound. The word "the" in the sentence is still a grammatical morpheme but it is not bound, it is called independent words. So, Derivational Morphology says that, new words enter language in two main ways - through the addition of words unrelated to any existing words and derivational morphology, the creation of new open-class words (open-class word or lexical content word is basically nouns, lexical verbs, adjectives, and adverbs) by the addition of morphemes to existing roots. Derivational morphemes increase the vocabulary and may allow speakers to convey their thoughts in a more interesting manner, but their occurence is not related to sentence structure [17].

FastText [18] has no difference between word2vec. It is word2vec plus Derivational Morphology. The method obtains that by something like parameter sharing. The main idea is representing words as a character n-grams. This type of formulation is important for morphologically rich languages like Turkish.

FastText uses char n-grams. For example we have word "fastest". FastText encodes this word as "<fas", "fast", "aste", "stes", "test", "est>" when $n = 4$. From now on, the word vector is representation of sum of its subwords: vec("fastest") = vec("<fas") + vec("fast") + vec("aste") + vec("stes") + vec("test") + vec("est>"). Each subword has $N = 300$ dimensions and thus so does words. Remember that, we have a scoring function in word2vec $\exp(v_{w_c}^T v_{w_t})$. In FastText we have scoring function that is defined as

$$s(w_1, w_2) = \sum_{g \in G_{w_1}} \mathbf{z}_g^T \mathbf{v}_{w_2} \tag{41}$$

Suppose that you are given a dictionary of n-grams of size G. Given a word w_1, let us denote by $G_{w_1} \subset \{1, ..., G\}$ the set of n-grams appearing in w_1. We associate a vector representation $\mathbf{z}_g$ to each n-gram g. We derive the objection that has this scoring with respect to each $\mathbf{z}_g^T$. While we updating our subword vector representations, we update words by sum of its updated subword vector representations. This morphological information significally improves syntactic tasks, also it gives good representations on small datasets.

1.7 Final Notes On Word Vectors

GloVe outperforms word2vec, but word2vec is more popular than Glove.

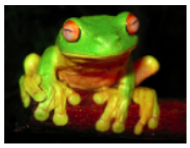
nearest neighbors of *frog*	Litoria	Leptodactylidae	Rana	Eleutherodactylus
Pictures				

Figure 9: Source: Official GloVe implementation repository

1.7.1 Roles Of Symmetry, Dimension & Window Size

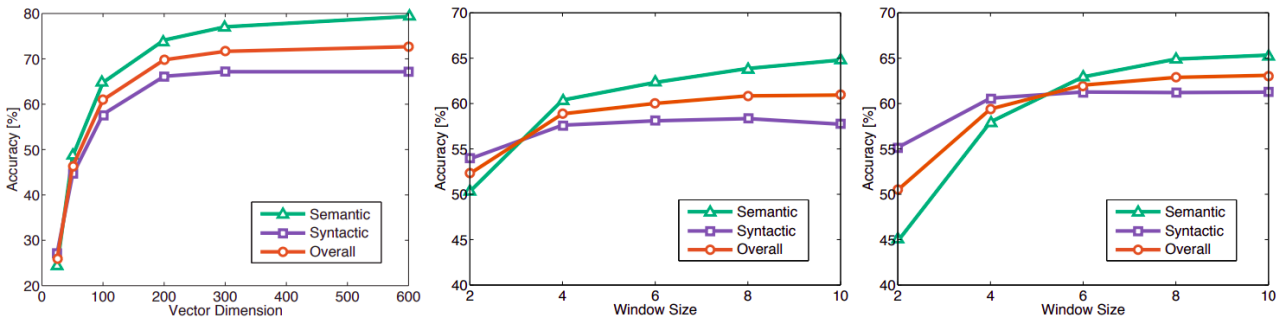


Figure 10: [16]

They observed diminishing returns for vectors larger than about 200 dimensions. Performance is better on the syntactic subtask for small and asymmetric context windows, which aligns with the intuition that syntactic information is mostly drawn from the immediate context and can depend strongly on word order. Semantic information, on the other hand, is more frequently non-local, and more of it is captured with larger window sizes.

1.7.2 Roles Of Corpus

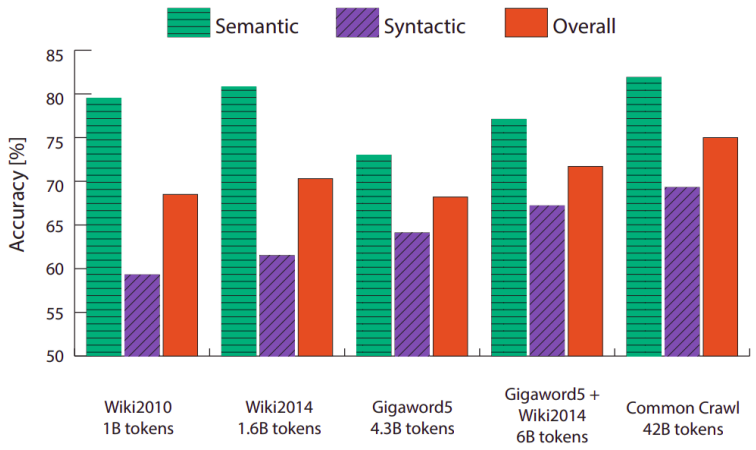


Figure 11: [16]

On the syntactic subtask, there is a monotonic increase in performance as the corpus size increases. This is to be expected since larger corpora typically produce better statistics. Interestingly, the same trend is not true for the semantic subtask, where the models trained on the smaller Wikipedia corpora do better than those trained on the larger Gigaword corpus. Due to, Wikipedia has fairly more comprehensive articles than news.

1.7.3 Are Word Vectors Only Learned With Neural Networks?

No:

- Term-Document Matrix in information retrieval.
- Word-Word Co-occurence Matrix.
- TF-IDF:

$$\mathrm{tf}_{t,d} = \mathrm{count}(t, d)$$

$$\mathrm{idf}_t = \log_{10}\left(\frac{N}{\mathrm{df}_t}\right)$$

N : total number of documents, df_t : total number of documents includes word t

- Positive Pointwise Mutual Information (PPMI):

$$p_{ij} = \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, \quad p_{i*} = \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}, \quad p_{*j} = \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}}$$

$$\mathrm{PPMI}_{ij} = \max\left(\log_2 \frac{p_{ij}}{p_{i*}p_{*j}}\right)$$

- Singular Value Decomposition (SVD):

$$X = U\Sigma V^T$$

- Hellinger-PCA
- etc.

1.8 Word Embeddings in Practice

So far, we have seen there are different methodologies to learn rich and diverse word embeddings. In practice, these word embeddings are trained on a generalized corpus or domain specific corpus to infer on a task. Before diving into the usage of word emebddings, we should learn "what tasks are in Natural Language Processing in general?".

1.8.1 NLP in Practice

Sequence Classification is the core task of overall. The main aim is to assign a label to a single sentence. For example, spam mail classification, you are given a sentence to predict whether this mail is spam. Or, you are given a sentence to determine whether this sentence is grammatically is correct.

Sequence Labeling is a syntactic task that the main goal is to predict the tags of the tokens by considering intra-relationships and inter-relationships among tokens. To give a more precise example from sequence labeling, Part-of-Speech (POS) Tagging is a sub-task of Sequence Labeling, where the main task is to identify the grammatical tag (i.e. ADJ: adjective, ADV: adverb, DET: determiner) for each token in sequence. Another popular task fir Sequence Classification is Named Entity Recognition (NER). In NER, the aim is to extract meaningful informations on a sentence or document. The common tags in NER are can be classifiede as LOC (location), PER (person), ORG (organization) etc.

Parsing can be evaluated in two categories: Dependency Parasing and Constituency Parsing. In Dependency Parsing, the aim is to examine the dependencies between the phrases in a sentence. Consider the sentence "I prefer the morning flight through Denver". In dependency between "fligh" and "Denver" (flight $->$ Denver) "flight" is the pinnacle and "Denver" is the kid or dependent. This relationship is classified as "nmod" which stands for nominal modifier. It functionally corresponds to an adverbial when it attaches to a verb, adjective or other adverb (explanation from Universal Dependencies). On the other hand, Constituency Parsing is basically parsing the sentence to chunks, in other words, extracting the representation of the syntactic structure of a string according to some context-free grammar (i.e., NP: Noun Phrase, VB: Verb Phrase).

Question Answering can be evaluated in four categories: classification based QA, Extractive QA, Open Generative QA and Closed Generative QA. In classification based QA, you are given a large number of classes and task is to choose the label from answer space as classification task. In Extractive QA, you are given a context and a question. The task is to extract a span from the given context. In Open Generative QA, you are given a context and a question. However, the task becomes answer generation from the context instead of extracting context. In Closed Generative QA, you are not given a context but only a question. The task becomes generating answer basedn on only the question.

Machine Translation is the task of translating a source sentence from source language to target sentence from target language. It generally requires a parallel corpus. Machine Translation is divided to Statistical Machine Translation (SMT) and Neural Machine Translation (NMT). In SMT, the common approach is extracting phrases and syntactic features and using them on a statistical language model (such as Moses Decoder). In NMT, the task becomes a sequence-to-sequence problem. The neural network tries to decode the target sentence regarding the features from source sentence which are obtained from neural network.

Summarization is a sequence-to-sequence problem. An encoder network takes a passage of text

and decoder tries to decode this text to relatively short passage. It requires a parallel corpus.

1.8.2 Common Benchmarks in NLP

SQuAD is **S**tanford **Q**uestion **A**nswering **D**ataset. Dataset contais context and question pairs

(Super) GLUE GLUE is a collection of different datasets and it is the most popular benchmark for evaluating models.

- **Microsoft Research Paraphrase Corpus (MRPC)**: Goal is to predict whether two sentences are semantically equivalent.
- **Semantic Textual Similarity Benchmark (STSB)**: Goal is to score two sentences by their semantic closeness.
- **Multi-Genre Natural Language Inference (MultiNLI)**: Based on a premise sentence and a hypothesis sentence, the aim is to predict whether the premise entails the hypothesis, contradicts the hypothesis, or neither of the two.
- **Recognizing Textual Entailment (RTE)**: Two class split version of MNLI.
- **Winograd Schema Challenge (WSC)**: Pairs of sentences are constructed by replacing the ambiguous pronoun with any possible referent. The task is to predict if the sentence with the pronoun substituted is entailed by the original sentence.
- **Words in Context (WiC)**: Goal is to decide whether the word is used with the same meaning in both sentences.

Stanford Natural Language Inference (SNLI) is a collection of sentence pairs that are labeled either as entailment, contradiction or semantic independence. **WMT** is a benchmark from Workshop on Statistical Machine Translation. Every year, it is released for different language pairs.

Penn Treebank (PTB) is one of the most known and used corpus for the evaluation of models for sequence labelling. The task is to predict each token's POS tag.

Universal Dependencies (UD) is a framework for consistent annotation of grammar (parts of speech, morphological features, and syntactic dependencies) across different human languages. UD is an open community effort with over 300 contributors producing nearly 200 treebanks in over 100 languages (from original website).

The Universal Morphology (UniMorph) project is a collaborative effort to improve how NLP handles complex morphology in the world's languages. The goal of UniMorph is to annotate morphological data in a universal schema that allows an inflected word from any language to be defined by its lexical meaning, typically carried by the lemma, and by a rendering of its inflectional form in terms of a bundle of morphological features from our schema (from original website).

Since we investigated the main tasks and benchmarks in NLP, now we should learn what can be done with word embedding frameworks. Normally a word embedding framework is a support model for a different neural network. For example, RNNs or Transformer have embedding layers to construct rich word vectors during training. However, word embeddings can be used for search, recommendation and similarity tasks in general, since it is a perfect suit for similarity metrics.
An example for usage of word embeddings is search engines. In search engines, you have collection of n documents and each document has a document identifier or a index D which is named as

docid. A word embedding model that is trained on this collection of documents would be helpful to rank and retrieve documents from a query at inference time. After training, we encode each document to a vector. This vector could be the mean of each word vector from tokenized tokens in a document. If we get the embedding of query from the trained model, we can compute the cosine similarity score between query vector and document vectors to rank documents. After ranking, we can recommend m documents to user by their ranking scores.

1.9 Multimodality and Natural Language Grounding

Multimodal learning is a wide research area in Deep Learning for last decades. There are so many studies are proposed and explored which have different objectives, sub-tasks and more. As a result of high-speed developments in both Natural Language Processing and Computer Vision, vision-and-language interaction has gained more attention. Taking into account the uncountable number of developments in this area, we would like to divide past and current research as **generative** models and **non-generative** models.

Generative models mainly include a sequence-to-sequence architecture, to generate image from text or vice-versa. One of the beginnings of sequence-to-sequence architectures, [19] proposed a multimodal recurrent architecture. They use a Recurrent Neural Network (RNN) to extract language features, and a Convolutional Neural Network (CNN) to generate images from text embedding. They adapt this model for both generation and ranking for retrieval. The same approach is used in [20], however they replaced Recurrent Neural Network with Long Short-Term Memory Network (LSTM), which gives embedding interaction spaces with a better precision. Following the developements on Machine Translation with alignment of source and target languages with cross-attention ([21], [22], [23]), and visual attention ([24], [25]), the first cross-attention model between two modalities is proposed by [26]. They use a CNN layer to extract image embeddings and RNN layer to generate captions. The main contribution is the adaptation of hard and soft attentions, which are proposed in [23], to learn visual-language embeddings and alignments. One of the first joint embedding space model is proposed by [27]. They use a R-CNN like model to generate captions not just for whole image but also for arbitrary image regions. With the latest developments on text transformers [28] and image transformers [29], multimodal generation can be accomplished by a full transformer network, as shown in [30].

To relate the multimodality with word embeddings, in this section we are going to talk about joint spaces for bi-modalities and image/text retrieval.

1.9.1 Image-Text Matching Objective

Image-Text matching objective is firstly shown in [31] and built on only Transformer based model: two distinct *intra* feature extractor (for both image and text) and a *inter* feature extractor, which is called "multimodal interactor".
For *intra* feature extractor for text modality, an encoder based pre-trained Transformer model can be used. For *intra* feature extractor for image modality, Vision Transformer (ViT) can be used. Passing images to ViT model, each image is pre-processed to square image with dimension of $3 \times 224 \times 224$. Then, each square image is separated to patches with $C \times 16 \times 16$. This means, we have 197 (+1 for special token `[CLS]`) total number of patches. Each patch is flattened and then passed to the Vision Transformer.
Output of text encoder and image encoder represents each modality with contextual information, however, interactions among modalities have not been extracted yet. To address this problem, it is helpful to design a multimodal interactor layer. This layer is a stack of self-attention layers with non-casual masking (Transformer's Encoder).

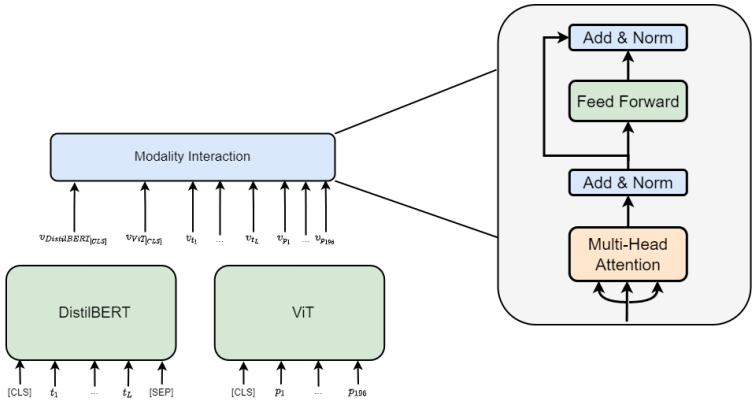


Figure 12: Multimodal Transformer Architecture for Image-Text Matching Objective

First of all, the output of ViT and text encoder are concatenated to a single tensor, which have dimension of $B \times (L+197) \times 768$. However, the position of `[CLS]` vector of ViT relocated to right of the `[CLS]` vector of DistilBERT:

$$X' = [v_{DistilBERT_{CLS}}; v_{ViT_{CLS}}; v_{DistilBERT_0}; ...; v_{DistilBERT_L}; v_{DistilBERT_{SEP}}; v_{ViT_0}; ...; v_{ViT_{197}}] \tag{42}$$

this formulation give same performance with averaging `CLS` vectors, however we do not want to reduce the alignment of two modalities. Then, the linear projection of each vector is obtained, namely Query (Q), Key (K), Value (V) vectors

$$\mathbf{X}' \cdot \mathbf{W}_Q + POS_{X'} = \mathbf{Q} \tag{43}$$
$$\mathbf{X}' \cdot \mathbf{W}_K + POS_{X'} = \mathbf{K} \tag{44}$$
$$\mathbf{X}' \cdot \mathbf{W}_V + POS_{X'} = \mathbf{V} \tag{45}$$

where POS is the positional encoding for input vectors. Then, alignment of each modality is calculated with self-attention

$$\mathbf{A} = \left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_{\mathbf{Q}}}} \right) \cdot V \tag{46}$$

At training, with probability $p = 0.5$, image of a single caption is changed. The objective is to classify whether it is changed or not. This states that, retrieval task can be done with a binary classification. Thus, it is needed that a single classifier layer top of the output of $[v_{DistilBERT_{CLS}}; v_{ViT_{CLS}}]$ embedding. This embedding has information of both multimodality and alignment of each modality, as explained in Chapter 5.

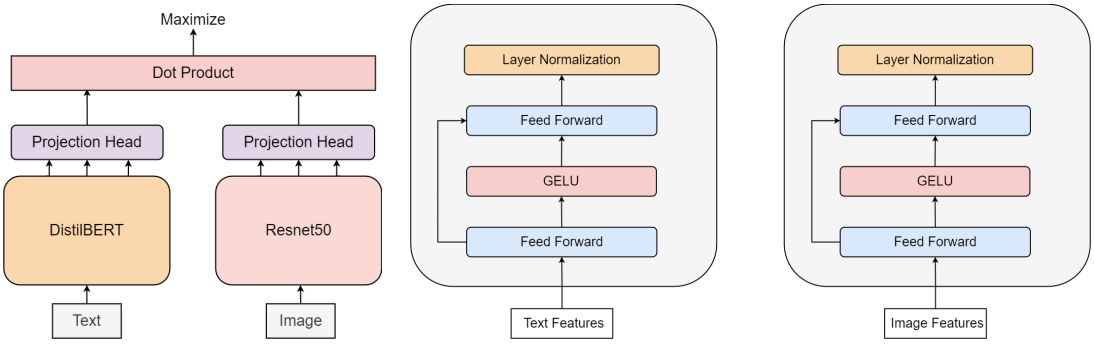
$$\mathbf{Z} = LN(\mathbf{A}) \tag{47}$$
$$p = softmax(\mathbf{Z}_0^N \cdot \mathbf{W}_{pool}) \tag{48}$$

where LN is Layer Normalization. The multimodal interactor can be designed as stack of transformer's encoder.

To understand the architecture of Transformer model, we refer reader to Section 7.

1.9.2 Deep Metric Learning for Retrieval

The latter objective choice is named as "Deep Metric Learning for Retrieval", which is nearly same as CLIP model [32]. This objective is built a Transformer based model for extracting *intra* features



(a) Model (b) Projection Heads

Figure 13: Modeling Deep Metric Learning for Retrieval

of texts and a ResNet model for extracting *intra* features of images. Rather than a multimodal interaction layer, the objective is to represent these output vectors in a joint space and maximize their similarity $d(image, text)$. As in image-text matching objective, two feature extractor for both modality is used.

To model a deep metric learning architecture, a projection head is designed to project text and image features vectors to same dimensionality. This projection head includes feed forward layers and a layer normalization layer with residual connection between input embedding.

$$\mathbf{Z}_1 = \text{MLP}(\mathbf{V}_{\text{modality}}) \tag{49}$$
$$\mathbf{Z}_2 = \text{GELU}(\mathbf{Z}_1) \tag{50}$$
$$\mathbf{Z}_3 = \text{MLP}(\mathbf{Z}_2) + \mathbf{Z}_1 \tag{51}$$
$$\mathbf{E} = \text{LN}(\mathbf{Z}_3) \tag{52}$$

Now, we call output of image projection head as "image embeddings" and output of image projection head as "text embeddings".
After output of the projection layer for both image and text, the dot product of both embeddings is calculated as

$$\mathbf{s} = \mathbf{E}_{\text{text}} \cdot \mathbf{E}_{\text{image}}^T \tag{53}$$

The dimension of output in dot product layer is $B \times 1$, which contains similarity scalars between image embeddings and text embeddings. Deep Metric Learning plays a role when our aim is to maximize this similarity vector $\mathbf{s}$. When the loss is calculated for dot product, gradient signal is flowing through to both text and image projection head by backpropagating. Hence, the embeddings for both modality is learned in a joint space. This allows us to compute similarities between query and images when we are trying to retrieve relevant images.

1.10 Word Embeddings Beyond English

2 Recurrent Neural Networks

2.1 Recurrent Neural Networks Motivation

Why Recurrent Neural Networks (RNNs)? If you have studied theoretical and practical aspects of Fully Connected Neural Networks, answer of this question may be intuitive for you. RNNs are a family of neural networks for processing and modeling "sequential" data. So, what is sequential?

- **Biological Data**: A biological data may have sequential structure, for example a single, continuous molecule of nucleic acid or protein.
- **Time Series**: A Time Series application can have specific practical case, for example Financial Forecasting. The ratio between income and expense can be modeled by time. The ratio at 3th month will affect the ratio at 5th month.
- **Language**: Language is position dependent. The word at position i may related with word j. Besides, its recursive and unbounded structure pushes us to use a sequential model like RNNs.

However; we will make our experiments, evaluations and examples in language perspective, mostly. Before introducing, let's explaing why not Fully Connected Networks. Take an input sentence, a traditional fully connected Fully Connected network would have separate parameters for each input feature, so it would need to learn all of the rules of the language separately at each position in the sentence [33].

2.2 Introducing Recurrent Neural Networks

Recurrent Neural Network is a function (f), which can be written recursively. Take and input $\mathbf{x}$, which can be decomposed as $\mathbf{x} : \{x_i, i \in 1, 2, ..., t\}$. Each x_i represents input element at time i. Introducing our function parameters as $\boldsymbol{\theta} \in \Theta$, our function can be written as

$$\mathbf{h}^{(t)} = f(\mathbf{h}^{(t-1)}; \boldsymbol{\theta}) \tag{54}$$

where $\mathbf{h}^{(t)}$ is the hidden state of our function. If the input set is $\mathbf{x} : \{x_i, i \in 1, 2, 4\}$, the introduced function can be written as

$$\begin{aligned} \mathbf{h}^{(4)} =& f(\mathbf{h}^{(3)}; \boldsymbol{\theta}) && (55)\\ =& f(f(\mathbf{h}^{(2)}; \boldsymbol{\theta}); \boldsymbol{\theta})) && (56)\\ =& f(f(f(\mathbf{h}^{(1)}; \boldsymbol{\theta}); \boldsymbol{\theta})); \boldsymbol{\theta}) && (57) \end{aligned}$$

Same formulation is applied for all finite length inputs. The formulation tells us that, each hidden state can be written as previous hidden states with same parameters. This allows us to model sequential data. Let's illustrate this process. As seen in Figure 1, at time step 1, $x^{(1)}$ is passed to

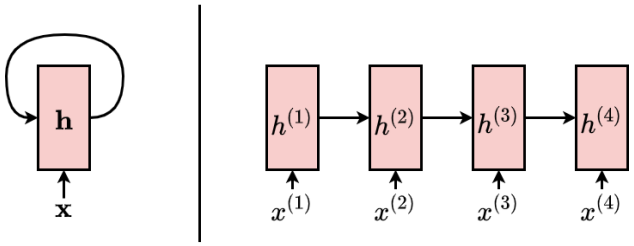


Figure 14: Left: Folded representation of RNN — Right: Unfolded representation of RNN.

a black block, it is processed and passed to time step 2. This time $x^{(2)}$ is processed with $h^{(2)}$, and with same parameters. This "prior" information from previous step allows us to model "sequential" structure of $\mathbf{x}$. At last, we process $x^{(4)}$ with given informations $x^{(1)} \rightarrow h^{(1)}, x^{(2)} \rightarrow h^{(2)}, x^{(3)} \rightarrow h^{(3)}$. Besides, this "shared parameter" prodecure is practical for processing variable length input with same architecture: for example $t = 5, 6, 7$ for $\mathbf{x} : \{x_i, i \in 1, 2, ..., t\}$. We will talk about this in later chapters.

2.3 Recap: Representing Text

Before more on RNNs, it is significant to learn "how to represent text in computer" in simplest way. Normally, a text is a string. We have to represent it numerically to use in neural networks.

2.3.1 One-Hot Encoding

One-hot encoding is the most common, most basic way to turn a token into a vector. It consists in associating a unique integer index to every word, then turning this integer index i into a binary vector of size $\mid V \mid$, which is the size of our vocabulary, that would be all-zeros except for the i-th entry, which would be 1. For example, if we have an vocabulary V: {"i": 0, "we": 1, "go": 2, "school": 3}, the word "we" becomes

$$\begin{bmatrix} 0 & 1 & 0 & 0 \end{bmatrix} \in \mathbb{N}^4 \tag{58}$$

and the sentence "i go school" becomes

$$\left\{ \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 3 \end{bmatrix} \right\} \in \mathbb{N}^{(3\times 4)} \tag{59}$$

2.4 Formulating Recurrent Neural Networks

We studied Recurrent Neural Networks as a mathematical function and representing text numerically. Now, we are going to write down the forward equation of the RNNs.

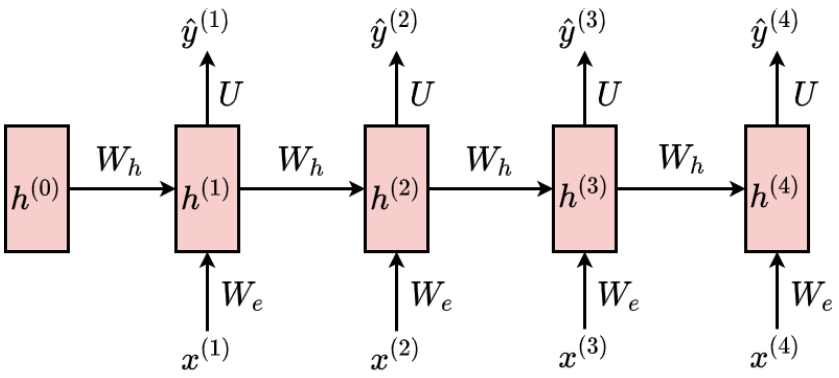


Figure 15: Representing forward equation of RNNs.

Figure 2 shows the forward phase of the RNNs. Let's write the algorithm with the dimensions of inputs, outputs and parameters.

Algorithm 1 Forward Algorithm for Recurrent Neural Network

Require: $\mathbf{x}, h^{(0)}, \{W_e, W_h, U\}, t$ ▷ $h^{(0)}$ can be learned or initialized with all-zeros.

1: **for** $i \in \{1, 2, 3, ..., t\}$ **do**

2: $\quad h^{(t)} \leftarrow \sigma(W_h \cdot h^{(t-1)} + W_e \cdot x^{(t)} + b_1)$ ▷ $x^{(t)} \in \mathbb{R}^{1\times|V|}, W_e \in \mathbb{R}^{d_h\times 1}, W_h \in \mathbb{R}^{d_h\times d_h}$

3: $\quad \hat{y}^{(t)} \leftarrow softmax(U \cdot h^{(t)} + b_2)$ ▷ $U \in \mathbb{R}^{1\times d_h}, \hat{y}^{(t)} \in \mathbb{R}^{1\times|V|}$

4: **end for**

2.4.1 Learning Embeddings Simultaneously

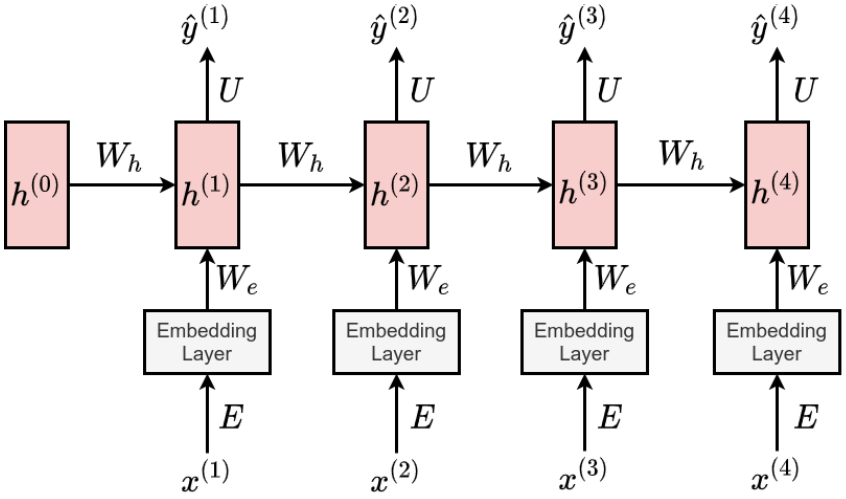


Figure 16: Dense embeddings with learning.

Not scope of the chapter, but word representations can be learned by Embedding Layer, which is a trainable matrix. It contains better representations than a single one-hot. This embeddings can be used as pretrained, or can be learned simultaneously. Let's write our equations again.

Algorithm 2 Forward Algorithm for Recurrent Neural Network (Embedding Layer)

Require: $\mathbf{x}, h^{(0)}, \{W_e, W_h, U, E\}, t$ ▷ $h^{(0)}$ can be learned or initialized with all-zeros.

1: **for** $i \in \{1, 2, 3, ..., t\}$ **do**

2: $\quad e^{(t)} \leftarrow E \cdot x^{(t)}$ ▷ $x^{(t)} \in \mathbb{R}^{1\times|V|}, E \in \mathbb{R}^{D\times 1}, e^{(t)} \in \mathbb{R}^{D\times|V|}$

3: $\quad h^{(t)} \leftarrow \sigma(W_h \cdot h^{(t-1)} + W_e \cdot e^{(t)} + b_1)$ ▷ $W_e \in \mathbb{R}^{d_h\times D}, W_h \in \mathbb{R}^{d_h\times d_h}$

4: $\quad \hat{y}^{(t)} \leftarrow softmax(U \cdot h^{(t)} + b_2)$ ▷ $U \in \mathbb{R}^{1\times d_h}, \hat{y}^{(t)} \in \mathbb{R}^{1\times|V|}$

5: **end for**

2.5 Backpropagation Through Time

The backward propagation for RNNs differs from Fully Connected or Convolutional Neural Networks. Since it is time dependent and parameter sharing, we have to formulate very careful.

2.5.1 Recap: Computational Graphs and Automatic Differentation

A computational graph is useful in visualizing dependency relations between intermediate variables; in other words, automatic differentation. In AD, all numerical computations are ultimately

compositions of a finite set of elementary operations for which derivatives are known, and combining the derivatives of the constituent operations through the chain rule gives the derivative of the overall composition. In the first phase of AD, the original function code is run forward, populating intermediate variables v_i and recording the dependencies in the computational graph through a book-keeping procedure. In the second phase, derivatives are calculated by propagating adjoints v_i' in reverse, from the outputs to the inputs [34]. Let's define a neural network:

$$\mathbf{z}_1 = \mathbf{X} \cdot w_1^T + b_1 \tag{60}$$
$$\mathbf{h}_1 = \sigma(\mathbf{z}_1) \tag{61}$$
$$\mathbf{z}_2 = \mathbf{h}_1 \cdot w_2^T + b_2 \tag{62}$$
$$\mathbf{h}_2 = \sigma(\mathbf{z}_2) \tag{63}$$
$$\mathbf{z}_3 = \mathbf{h}_2 \cdot w_3^T + b_3 \tag{64}$$
$$\hat{\mathbf{y}} = \mathbf{h}_3 = \sigma(\mathbf{z}_3) \tag{65}$$
$$L(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{M} \sum (y - \hat{y})^2 \tag{66}$$

The computational graph of this forward equation would be

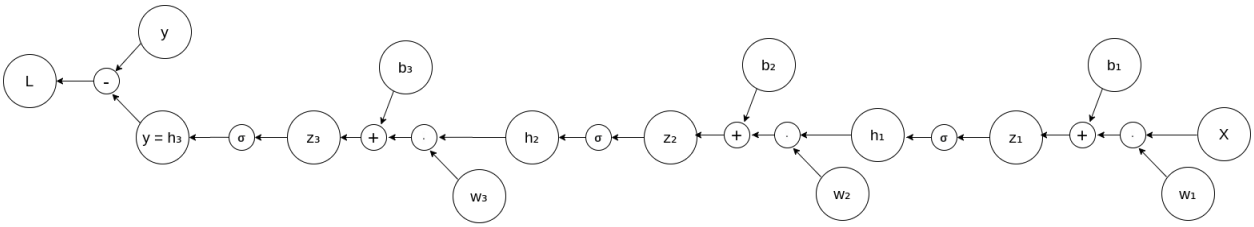


Figure 17: Computational Graph (Forward)

If we reverse the graph, the backward will be calculated by propagating adjoints in reverse, from the outputs to the inputs.

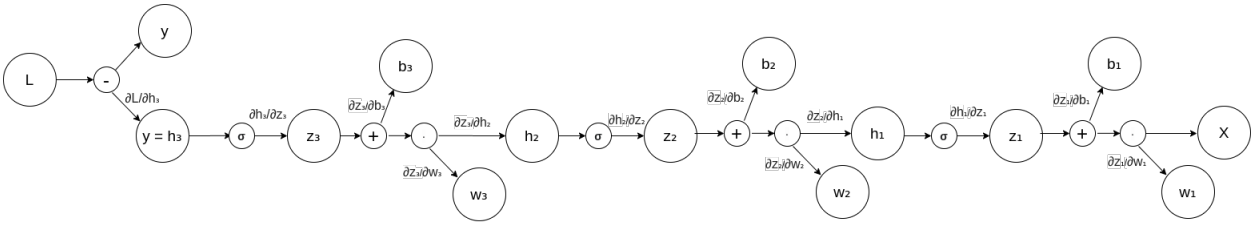


Figure 18: Computational Graph (Backward)

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial w_3} \tag{67}$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial w_2} \tag{68}$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} \tag{69}$$

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial b_3} \tag{70}$$

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial b_2} \tag{71}$$

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial h_3} \cdot \frac{\partial h_3}{\partial z_3} \cdot \frac{\partial z_3}{\partial h_2} \cdot \frac{\partial h_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} \tag{72}$$

We will talk about how to construct a computational graph for RNNs in later chapters.

2.5.2 Formulating Backpropagation Through Time

Suppose that we are calculating the loss for each output $y^{(t)}$. So loss is to be the cross-entropy loss:

$$\mathbb{E}_t[y_i^t, \hat{y}_i^t] = -\, y_i^t \cdot \log \hat{y}_i^t \tag{73}$$

$$\mathbb{E}_t[y^t, \hat{y}^t] = -\, y^t \cdot \log \hat{y}^t \tag{74}$$

Then the loss is average loss for each output,

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{N} \cdot \sum_t^T y^t \cdot \log \hat{y}^t \tag{75}$$

2.5.3 Gradient for U

It is more clear for derivatives with denoting $q^t = U \cdot h^{(t)}$

$$\frac{\partial L^t}{\partial U_{ij}} = \frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial V_{ij}} \tag{76}$$

You can clearly see that for $L^t = -y_k^t \log \hat{y}_k^t$, the derivative is

$$\frac{\partial L^t}{\partial \hat{y}_k^t} = -\frac{y_k^t}{\hat{y}_k^t} \tag{77}$$

Now, we will derive $\hat{y}_k^t = softmax(q_l^t)$ which is $\frac{\partial \hat{y}_k^t}{\partial q_l^t}$. First of all, we have to evaluate the derivation of softmax function.

$$\hat{y}_k^t = \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} \tag{78}$$

Here, softmax is a $\mathbb{R}^n \to \mathbb{R}^n$ mapping function:

$$\hat{y}_k^t : \begin{bmatrix} q_1^t \\ q_2^t \\ q_3^t \\ \vdots \\ q_N^t \end{bmatrix} \to \begin{bmatrix} \hat{y}_1^t \\ \hat{y}_2^t \\ \hat{y}_3^t \\ \vdots \\ \hat{y}_N^t \end{bmatrix} \tag{79}$$

Therefore, the Jacobian of the softmax will be:

$$\frac{\partial \hat{y}^t}{\partial q^t} = \begin{bmatrix} \frac{\partial \hat{y}_1^t}{\partial q_1^t} & \frac{\partial \hat{y}_1^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_1^t}{\partial q_N^t} \\ \frac{\partial \hat{y}_2^t}{\partial q_1^t} & \frac{\partial \hat{y}_2^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_2^t}{\partial q_N^t} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \hat{y}_N^t}{\partial q_1^t} & \frac{\partial \hat{y}_N^t}{\partial q_2^t} & \cdots & \frac{\partial \hat{y}_N^t}{\partial q_N^t} \end{bmatrix} \tag{80}$$

Let's compute the $\frac{\partial \hat{y}_k^t}{\partial q_l^t}$

$$\frac{\partial \hat{y}_k^t}{\partial q_l^t} = \frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} \tag{81}$$

You all remember the quotient rule for derivatives from high school; for $f(x) = \frac{g(x)}{h(x)}$, the derivative of $f(x)$ is given by:

$$f'(x) = \frac{g'(x) \cdot h(x) - h'(x) \cdot g(x)}{h^2(x)} \tag{82}$$

Hence, in our case, $g_k = \exp(q_{t_k})$ and $h_k = \sum_n^N \exp(q_{t_n})$. The derivative, $\frac{\partial}{\partial q_l^t} \cdot h_k = \frac{\partial}{\partial q_l^t} \cdot \sum_n^N \exp(q_n^t)$ is equal to $\exp(q_l^t)$ because $\frac{\partial}{\partial q_l^t} \cdot \exp(q_n^t) = 0$ for $l \neq n$.
Derivative of g_k respect to q_l^t is $\exp(q_l^t)$ only if $k = l$, otherwise it is a constant 0. Therefore, **if we derive the gradient of the off-diagonal entries** of the Jacobian will yield:

$$\begin{aligned} \frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} =& \frac{0 \cdot \sum_n^N \exp(q_n^t) - \exp(q_l^t) \cdot \exp(q_k^t)}{\left[\sum_n^N \exp(q_n^t)\right]^2} && (83) \\ =& - \frac{\exp(q_l^t)}{\sum_n^N \exp(q_n^t)} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} && (84) \\ =& - \hat{y}_k^t \cdot \hat{y}_l^t && (85) \end{aligned}$$

Similiarly, **if we derive the gradient of the diagonal entries**, $diag(\partial \hat{y}_k / \partial q_l)$ for $k = l$, we have that:

$$\begin{aligned} \frac{\partial}{\partial q_l^t} \cdot \frac{\exp(q_k^t)}{\sum_n^N \exp(q_n^t)} =& \frac{\exp(q_k^t) \cdot \sum_n^N \exp(q_n^t) - \exp(q_l^t + q_k^t)}{\left[\sum_n^N \exp(q_n^t)\right]^2} && (86) \\ =& \frac{\exp(q_k^t) \cdot \left(\sum_n^N \exp(q_n^t) - \exp(q_l^t)\right)}{\left[\sum_n^N \exp(q_n^t)\right]^2} && (87) \\ =& \hat{y}_k^t (1 - \hat{y}_l^t) && (88) \end{aligned}$$

Thus, we have

$$\frac{\partial \hat{y}_k^t}{\partial q_l^t} = \begin{cases} -\hat{y}_k^t \hat{y}_l^t, & \text{if } k \neq l \\ \hat{y}_k^t (1 - \hat{y}_l^t), & \text{if } k = l \end{cases} \tag{89}$$

If we put Equation 24 and 36 together, gives us a sum over all values of k to obtain $\frac{\partial L^t}{\partial q_l^t}$:

$$\frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t}\bigg|_{k=l} = \frac{y_k^t}{\hat{y}_k^t} \cdot \hat{y}_k^t(1 - \hat{y}_k^t) = -y_k^t + y_k^t \cdot \hat{y}_l^t = -y_l^t + y_l^t \cdot \hat{y}_l^t \tag{90}$$

$$\frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t}\bigg|_{k\neq l} = \sum_{k\neq l} \frac{y_k^t}{\hat{y}_k^t} \cdot \hat{y}_k^t \cdot \hat{y}_l^t = \sum_{k\neq l} y_k^t \cdot \hat{y}_l^t \tag{91}$$

$$(37) + (38) = -y_l^t + y_l^t \cdot \hat{y}_l^t + \sum_{k\neq l} y_k^t \cdot \hat{y}_l^t = -y_l^t + \sum_k y_k^t \cdot \hat{y}_l^t = -y_l^t + \hat{y}_l^t \sum_k y_k^t \tag{92}$$

If you recall that y^t are all one-hot vectors, then that sum is just equal to 1. So we have

$$\frac{\partial L^t}{\partial q_l^t} = \hat{y}_l^t - y_l^t \tag{93}$$

Recall that we defined $q^t = U \cdot h^{(t)}$, so we can say that

$$q_l^t = U_{lm} \cdot h_m^t \tag{94}$$

Then we have,

$$\frac{\partial q_l^t}{\partial U_{ij}} = \frac{\partial}{\partial U_{ij}} \cdot (U_{lm} h_m^t) \tag{95}$$

$$= \delta_{il} \cdot \delta_{jm} \cdot h_m^t \tag{96}$$

$$= \delta_{il} \cdot h_j^t \tag{97}$$

If we put Equation 40 and 44, we have

$$\frac{\partial L^t}{\partial U} = (\hat{y}^t - y^t) \cdot h_t \tag{98}$$

2.5.4 Gradient for W_h

It is clear that y^t depends on W_h both directly and indirectly. We can directly see the partial derivatives like:

$$\frac{\partial L^t}{\partial W_{h_{ij}}} = \frac{\partial L_t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial h_m^t} \cdot \frac{\partial h_m^t}{\partial W_{h_{ij}}} \tag{99}$$

Yes this is the partial derivatives respect to $W_{h_{ij}}$ but note that at the last term, there is an implicit dependency of h^t on $W_{h_{ij}}$ through $h^{(t-1)}$. Hence, we have

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial W_{h_{ij}}} \tag{100}$$

We can apply this again and again

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial W_{h_{ij}}} + \frac{\partial h_m^t}{\partial h_n^{(t-1)}} \cdot \frac{\partial h_n^{(t-1)}}{\partial h_p^{(t-2)}} \cdot \frac{\partial h_p^{(t-2)}}{\partial W_{h_{ij}}} \tag{101}$$

This equation continues until h^0 is reached. h^0 is a vector of zeros.
Note that of these four terms we have already calculated first two derivatives. The third one is:

$$\frac{\partial q_l^t}{\partial h_m^t} = \frac{\partial}{\partial h_m^t} \cdot (U_{lb} \cdot h_b^t) \tag{102}$$

$$= U_{lb} \cdot \delta_{bm} \tag{103}$$

$$= U_{lm} \tag{104}$$

The last term at Equation 48 collapses to

$$\frac{\partial h_m^t}{\partial h_n^{(t-2)}} \cdot \frac{h_n^{(t-2)}}{\partial W_{h_{ij}}} \tag{105}$$

and we can turn the first item into

$$\frac{\partial h_m^t}{\partial h_n^t} \cdot \frac{\partial h_n^t}{\partial W_{h_{ij}}} \tag{106}$$

Then we have the compact form that is:

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{107}$$

And we rewrite this as sum over all values of p less than t, we have

$$\frac{\partial h_m^t}{\partial W_{h_{ij}}} = \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{108}$$

If we combine all, we have

$$\frac{\partial L^t}{\partial W_{h_{ij}}} = (\hat{y}_l^t - y_l^t) \cdot U_{lm} \cdot \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{h_{ij}}} \tag{109}$$

2.5.5 Gradient for W_e

It is easy to calculate the gradients W_e now.

$$\frac{\partial L^t}{\partial W_{e_{ij}}} = \frac{\partial L^t}{\partial \hat{y}_k^t} \cdot \frac{\partial \hat{y}_k^t}{\partial q_l^t} \cdot \frac{\partial q_l^t}{\partial h_m^t} \cdot \frac{\partial h_m^t}{\partial W_{e_{ij}}} \tag{110}$$

$$\frac{\partial h_m^t}{\partial W_{e_{ij}}} = \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{e_{ij}}} \tag{111}$$

$$\frac{\partial L^t}{\partial W_{e_{ij}}} = (\hat{y}_l^t - y_l^t) \cdot U_{lm} \cdot \sum_{p=0}^{t} \frac{\partial h_m^t}{\partial h_n^p} \cdot \frac{\partial h_n^p}{\partial W_{e_{ij}}} \tag{112}$$

2.6 Multilayer Recurrent Neural Networks

A recurrent neural network can be made deep in many way. Multilayer Recurrent Neural Networks are stack of RNNs, which is designed to be more powerfull. However, if the number of stacks increased carelessly, the model tends to overfit the current dataset and perform poor on dev dataset.
Forward algorithm and backward algorithm is nearly the same as Vanilla Recurrent Neural Network.

However, with deep architecture of it, it may require more computational power and memory for both activations, parameters and gradients. In practice, N layer RNN can be implemented with N single RNN networks but modern frameworks like PyTorch have `n_layers` parameter to build N layer RNN easily.

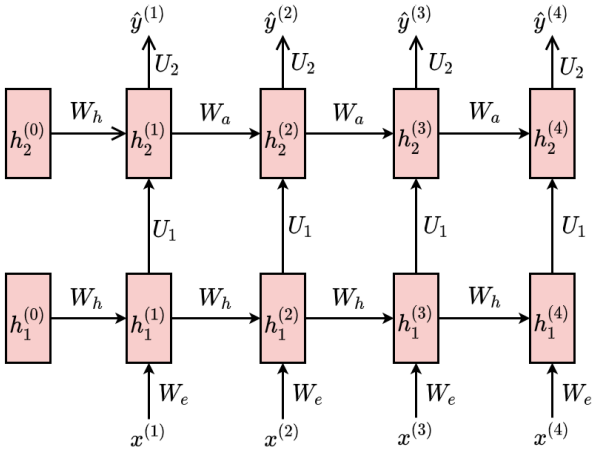


Figure 19: Multilayer Recurrent Neural Networks

Figure 6 shows 2 layer Multi RNN. In recent years power of Multi RNNs are shown: 2 layer RNNs are used to get contextual representation of words [35]. However, this is not the scope of this chapter.

2.7 Bidirectional Recurrent Neural Networks

Throughout the vanishing and exploding gradient problems (we will evaluate this situation on later chapters) and long-term dependencies of traditional Recurrent Neural Networks, yet there is an another problem for traditional Recurrent Networks. The characteristics of a sequential data could be unidirectional or bidirectional. Lack of traditional Recurrent Neural Networks is that the state at a particular time unit only has knowledge about the past inputs up to a certain point in a sequential data, but it has no knowledge about future states. What do I mean when I said unidirectional or bidirectional features of a sequential data? In certain applications like language modelling, the results are vastly improved with knowledge about both past and future states.
Let's give a very simple example based on a very simple sentence: "The rat ate cheese". The word "rat" and the word "cheese" are related in someway. There is a 'passive' advatage in using knowledge about both the past and future words. Cheese is generally eaten by rats and rats generally eat cheese. As I said this is a very simple example.

Given a sequence of N tokens $(t_1, t_2, ..., t_N)$, forward model computes the probability of the sequence by modeling the probability of token t_k, given the history $(t_1, t_2, ...t_{k-1})$:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, ..., t_{k-1}) \tag{113}$$

At each position k, each RNN layer outputs a context-dependent representation $\vec{\mathbf{h}}_{k,j}$ where $j = 1, 2, ..., L$ (number of layers). Top layer RNN output, $\vec{\mathbf{h}}_{k,L}$ is used to predict next token t_{k+1} with softmax. The equations that we formulated above are for forward RNN. The backward RNN is

similar to forward LM:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N) \tag{114}$$

with each backward RNN layer j in L layer deep model producing representations $\overleftarrow{\mathbf{h}}_{k,j}$ of t_k, given $(t_{k+1}, t_{k+2}, ..., t_N)$. And the formulation jointly maximizes the log-likelihood of the forward and backward directions:

$$\sum_{k=1}^{N} \log p(t_k \mid t_1, t_2, ..., t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \tag{115}$$

2.8 Architecture of bi-RNN

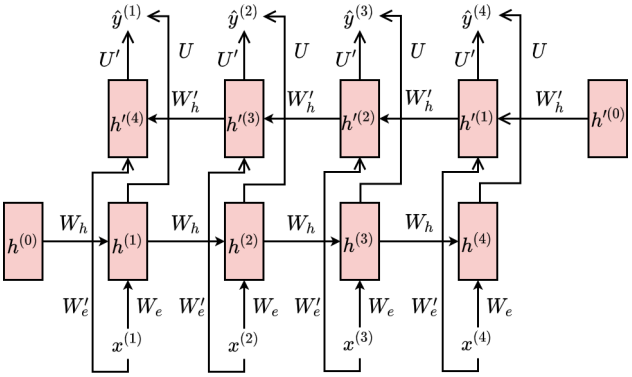


Figure 20: Bidirectional Recurrent Neural Network

In the bidirectional Recurrent Neural Networks, we have seperate hidden states/layers $h^{(t)}$ and $h'^{(t)}$ for the forward and backward directions. The forward hidden states interact with each other and the same is true for the backward hidden states. There is no multilayered connections between them. However, both $h^{(t)}$ and $h'^{(t)}$ receive input from the same vector $\hat{y}^{(t)}$. In general, any property of the current word can be predicted more effectively using this approach, because it uses the context on both sides. For example, the ordering of words in several languages is somewhat different depending on grammatical structure. Bidirectional Recurrent Neural Networks work well in tasks where the predictions are based on bidirectional context like handwrittings.
The outputs of both forward and backward directions can be averaged or concatenated to produce a single output. PyTorch framework applies this pooling strategy by concatenating.

2.9 Turing Completeness

Recurrent neural networks are known to be Turing complete. Turing completeness means that a recurrent neural network can simulate any algorithm, given enough data and computational resources [36]. Define a formal language Σ^* for some finite alphabet Σ. Intuitively, a "grammar" is the finite set of rules that describes the structure of an infinite language. In formal language theory, a grammar can be seen as a computational system (automaton) describing a language. For example, a Deterministic Finite Automaton can recognize ab^*. However it cannot recognize $a^n b^n$, because we must "count" the n to recognize b^n after a^n. Finite-state automata represent the limits of computation with only finite memory available. With additional memory, a recognizer is capable of specifying more complex languages. One way to view more complicated formal grammars,

then, is as finite-state automata augmented with various kinds of data structures, which introduce different kinds of unbounded memory [37].
With a stack automaton, we have a different way to create a recognizer for $a^n b^n$. We push each a onto the stack. For a b, we pop if the stack contains a, and raise an error if it contains a b or is empty. Finally, at the end of the string, we accept if the stack is empty. Also, a DFA cannot recognize parentheses languages. The set of languages recognizable by a stack automaton is called the deterministic context-free languages (DCFLs), and forms a strict subset of the context-free languages (CFLs) proper [37].

A big question in mathematical linguistics is the place of natural language within this hierarchy. Context-free formalisms describe much of the hierarchical structure of natural languages (or mildly context-sensitive).

A weighted finite-state automata uses a finite-state machine to encode a string into a number, rather than producing a boolean decision for language membership. A path score is seen like a basic RNN:

$$W(\pi) = \lambda(q_0) \otimes \left(\bigotimes_{i=1}^{t} \tau(q_{i-1,x_i,q_i}) \right) \otimes p(q_t) \tag{116}$$

If we introduce a Hankel Matrix H_f to a WFS, we can approximate any non-binary string with learning optimal $\hat{H}_f$ with SGD [37]. This definition is related to computational power of RNNs.

2.10 Practical Recurrent Neural Networks

In this section, we will evaluate what we can build with RNNs as a practical application. RNNs are designed to model sequential tasks like Sequence Classification, Sequence Labeling, Language Modeling. Besides, it can be used for Speech Recognition, Machine Translation, Image Generation etc. However, we do not include this topics in this chapter.

2.10.1 Special Tokens

In general, it is convenient to use "special tokens" in our tasks. For example, introducing `[START]` and `[END]` tokens might be helpful. To give an example

- **S1:** `[START]` fall fires burning neath black twisted boughs `[END]`
- **S2:** `[START]` a blaze so high it lights the night `[END]`

Assume that, we are tokenizing those sentences to get token ids. Then, pass it to a RNN model. This means batch size of 2. The token ids are

- **S1:** [0, 167, 137, 5141, 78, 77, 4561, 13, 1]
- **S1:** [0, 55, 744, 21, 44, 324, 879, 13596, 7, 1]

Generally in Deep Learning, we represent our data with tensors and the batch size would be a dimension of our tensor. However, as you can see, length of the introduced sentences are not equal. We are not able to represent it as a tensor, due to number of entries in a row must be equal.

2.10.2 Batching Variable Length Inputs

As we mentioned in previous section, length of sentences (sequences) might be different and we are not always able to generate batch tensors. So, what should we do?

If we use batch size of 1, this will not a real problem. Because, some of deep learning frameworks uses dynamic computational graphs. A dynamic computation graph is rebuilt at each training iteration. You can easily creat or remove a differentiable parameter at training, and this will not ruin your computational graph. Using variable sequences with batch size of 1 requires dynamic computational graph.

On the other side, static computational graphs, is rebuilt only once. So, you should pass your inputs as constant length.

Frameworks like PyTorch or DyNet [38] have excellent and efficient implementations for dynamic computational graphs, variable length batching etc. However, frameworks like Keras or Tensorflow still lack about it (even with introducing eager execution).

What if we use batch size of >1? We have to pad our sequences with padding token `[PAD]` to construct an appropriate tensor for batching.

- **S1:** `[START]` fall fires burning neath black twisted boughs `[PAD] [END]`
- **S2:** `[START]` a blaze so high it lights the night `[END]`

And,

- **S1:** [0, 167, 137, 5141, 78, 77, 4561, 13, 2, 1]
- **S1:** [0, 55, 744, 21, 44, 324, 879, 13596, 7, 1]

2.10.3 Sequence Classification

Sequence Classification is a task of classifying sentences by their sentiment or something else. Since a sentence is compisition of positional words, we can achieve this task with RNNs. So, how we can build a classifier with a RNN?

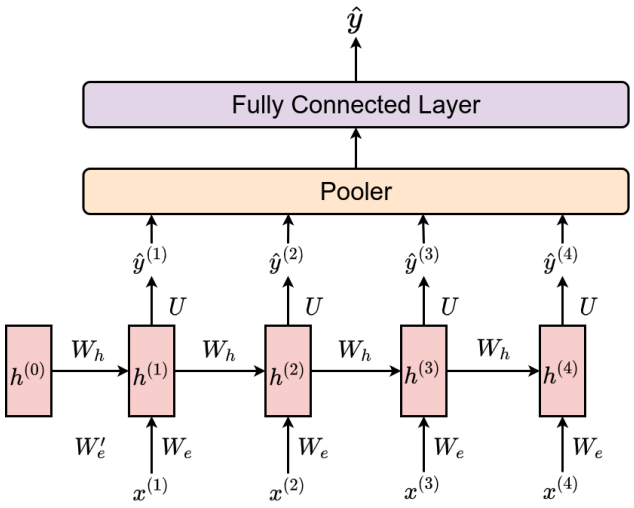


Figure 21: A sentence classifier with RNN.

Comparing to other tasks, this problem has very intuitive answer: adding a classifier layer (can be a Fully Connected Layer) to last output of the RNN. Since last output of RNN contains full information of the current sentence (because last position contains pervious information), it is proper to design this model.

2.10.4 Sequence Labeling

Sequence labeling has been one of the most discussed topics in Linguistics and Computational Linguistics history. Challenges like Dependency Parsing, Word Sense Disambiguation and Sequence Labeling, etc., arose with the formal definition of the syntax. Sequence labeling is a Natural Language Processing task. It aims to classify each token (word) in a class space **C**. This classification approach can be independent (each word is treated independently) or dependent (each word is dependent on other words).

Words are sequential building blocks of sentences. Each word contributes syntactic and semantic properties to a sentence. For example, a word can be an adjective, which can give positive semantics (delicate). By doing that, this Adjective can describe a noun (tender boy). This sequential relationship can go recursively and unbounded. This shows us words are related to each other. But how can we determine the role of the word? For example, how do we decide which word or phrase is a noun or which one is an adjective?

2.10.4.1 Part-Of-Speech-Tagging

Part-Of-Speech Tagging (POS Tagging) is a sequence labeling task. It is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech (syntactic tag), based on both its definition and its context. POS Tagging is a helper task for many tasks about NLP: Word Sense Disambiguation, Dependency Parsing, etc.

A sequence is a series of tokens where tokens are not independent of each other. Series in mathematics and sentences in linguistics are both sequences. Because; in both of them, the next token depends on the previous ones or vice versa.

Determining POS Tags is much more complicated than simply mapping words to their tags. Consider word **back**:

- Earnings growth took a **back/JJ** seat.
- A small building in the **back/NN**.
- A clear majority of senators **back/VBP** at the bill.

Each word back has a different role. The first back is Adjective; the second black is Noun, the third back is non-3rd person singular present Verb [1].

2.10.4.2 Named Entity Recognition

The first step in information extraction is to detect the entities in the text. A named entity is, roughly speaking, anything that can be referred to with a proper name: a person, a location, an organization. NER helps us to identify and extract critical elements from the text.

NER can be used for customer services, medical purposes, document categorization. For example, extracting aspects from customer reviews with NER helps identification of semantics. Or extracting medical diseases and drugs from a text helps identification of treatment.

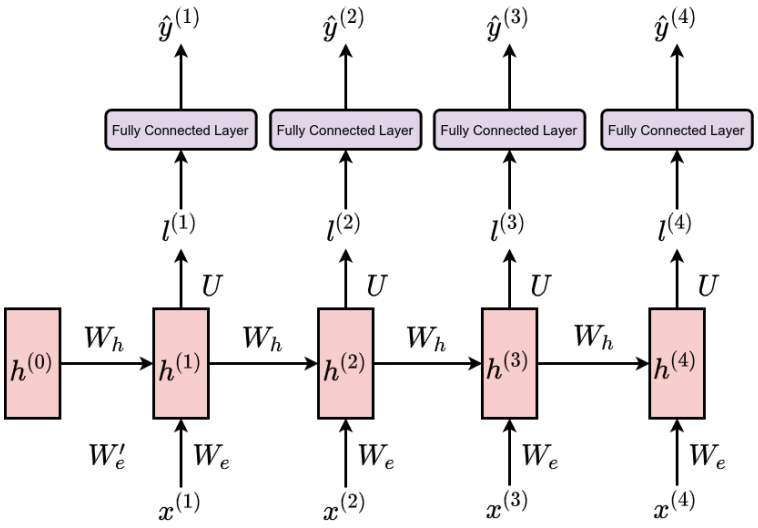


Figure 22: A token classifier with RNN.

If we put a classifier layer (for example fully connected layer), we are able to classify each token. A POS task or NER task can be done with RNNs like that.

2.10.5 Language Modeling

A (generative) language model is "autoregressive". A language model is called autoregressive if it predicts future tokens based on past tokens.
Training is autoregressive language model is similiar to POS tagging. We are calculating loss between input and shifted input:

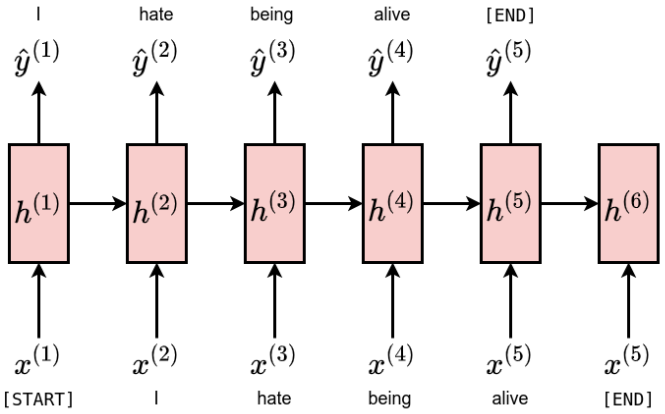


Figure 23: Training a autoregressive language model.

However, inference time differs from training procedure. We actually, generate tokens/words autoregressively.

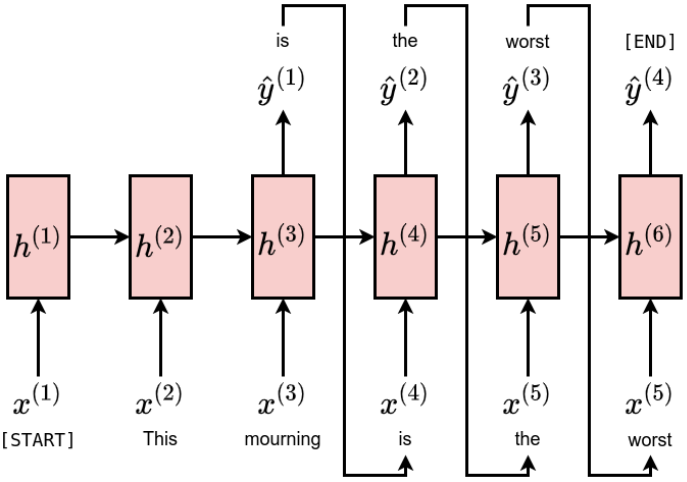


Figure 24: Generating text with RNN LM.

2.11 Vanishing Gradients

Vanishing gradients is a crucial problem in RNNs. It is mainly caused by long dependencies (high sequence lengths). If our sentence length is around ∼10, then it is easy to capture full information from text. However, applications like text summarization, the length of the text may be long. If the length of our sentence converges to ∼10000, it is not always easy to capture full information: i.e., last hidden layer may not generalize the first words of our sentence. To make a formal definition of vanishing gradients, recall that from RNNs,

$$h^{(t)} = \sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1) \tag{117}$$

Therefore,

$$\frac{\partial h^{(t)}}{\partial h^{(t-1)}} = diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1))W_h \tag{118}$$

Consider the gradient of loss $J^{(i)}(\theta)$ on step i, with respect to the hidden state $h^{(j)}$ on the same previous step j:

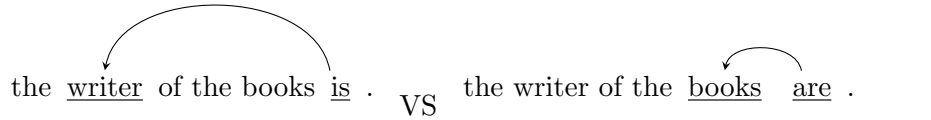
$$\frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} = \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot \prod_{j<t\leq i} \frac{\partial h^{(t)}}{\partial h^{(t-1)}} \tag{119}$$

$$= \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \cdot W_h^{(i-j)} \prod_{j<t\leq i} diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)) \tag{120}$$

So, if W_h is small, then this term gets vanishingly small as$i$ and j get further apart. Consider L2 matrix norms,

$$\left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(j)}} \right\| \leq \left\| \frac{\partial J^{(i)}(\theta)}{\partial h^{(i)}} \right\| \cdot \|W_h\|^{(i-j)} \cdot \prod_{j<t\leq i} \left\| diag(\sigma'(W_h \cdot h^{(t-1)} + W_x \cdot x^{(t)} + b_1)) \right\| \tag{121}$$

If the $\max_{\lambda_i} \lambda_i \leq 1$ then the gradient will shrink exponentially. The problem of vanishing gradients cause what we call "syntactic recency" vs "sequential recency". As in the Vanishing Gradient, the problem of recency is getting important while sentence length is increasing:



2.11.1 Gradient Clipping

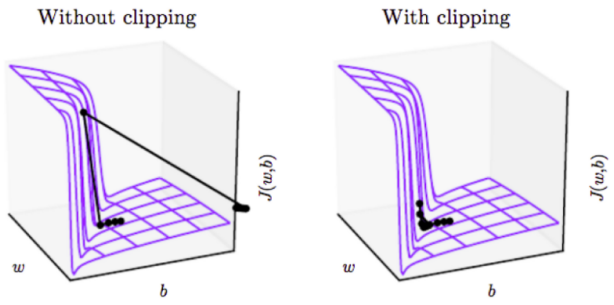


Figure 25: without clipping vs with clipping [39]

Algorithm 3 Gradient Clipping Algorithm [39]

Require: *threshold*

1: $\hat{\mathbf{g}} \leftarrow \frac{\partial E}{\theta}$
2: **if** $\|\hat{\mathbf{g}}\| \geq threshold$ **then**
3: $\quad \hat{\mathbf{g}} \leftarrow \frac{threshold}{\|\hat{\mathbf{g}}\|}\hat{\mathbf{g}}$
4: **end if**

From the Deep Learning textbook [33]: the basic idea is to recall that the gradient does not specify the optimal step size, but only the optimal direction within an infinitesimal region. When the traditional gradient descent algorithm proposes to make a very large step, the gradient clipping heuristic intervenes to reduce the step size to be small enough that it is less likely to go outside the region where the gradient indicates the direction of approximately steepest descent [33].

2.12 Long Short-Term Memory

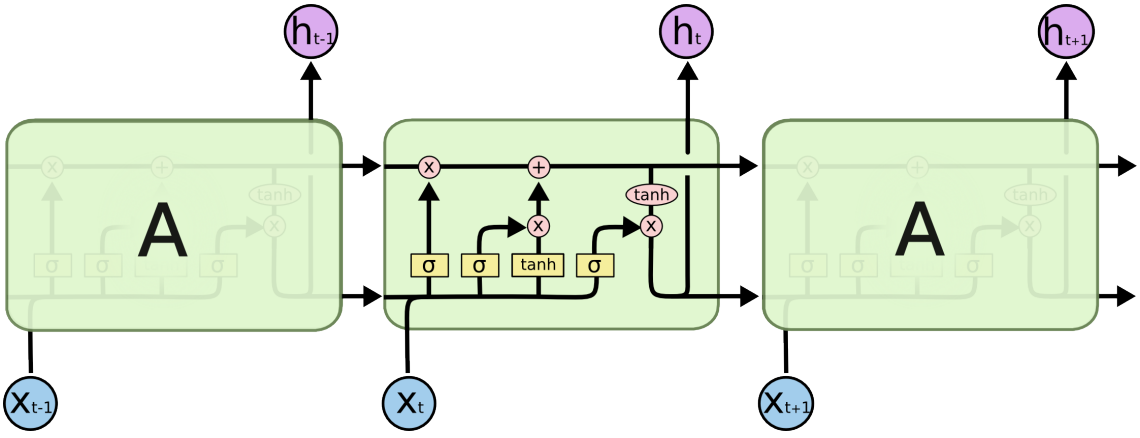


Figure 26: Long short-term memory [40].

We are lucky that there is an amazing explanation and definition for Long Short-Term Memory from Christopher Olah's LSTM Blog Post [40]. We are going to study his detailed work:

Long Short-Term Memory (LSTMs) are a special kind of RNNs, capable of learning long-term

dependencies. They were proposed in [41]. LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!

LSTMs are fancy version of RNNs. Instead of a unit that simply applies an elementwise nonlinearity to the affine transformation of inputs and recurrent units, LSTM recurrent networks have "LSTM cells" that have an internal recurrence (a self-loop), in addition to the outer recurrence of the RNN. Each cell has the same inputs and outputs as an ordinary recurrent network, but has more parameters and a system of gating units that controls the flow of information [33].

2.12.1 Cell State

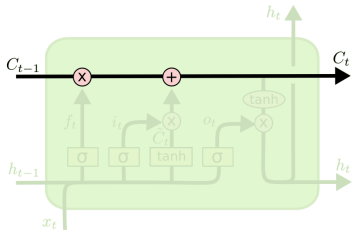


Figure 27: Cell state [40].

The key to LSTMs is the cell state, the horizontal line running through the top of the diagram. The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged. The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates. Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.

2.12.2 Forget Gate

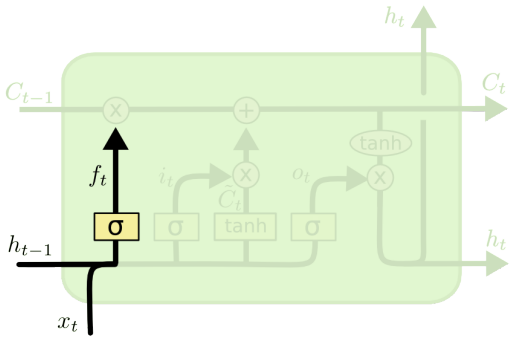


Figure 28: Forget gate [40].

The first step in our LSTM is to decide what information we're going to throw away from the cell state. This decision is made by a sigmoid layer called the forget gate. Let's consider a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject [40].
The formal definition of forget gate can be done:

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right) \tag{122}$$

where $\mathbf{x}^{(t)}$ is current input vector and $\mathbf{h}^{(t)}$ is the current hidden layer vector. $\mathbf{b}^f$ is biases, $\mathbf{U}^f$, $\mathbf{W}^f$ are trainable input and recurrent weights for forget gate.

2.12.3 External Input Gate

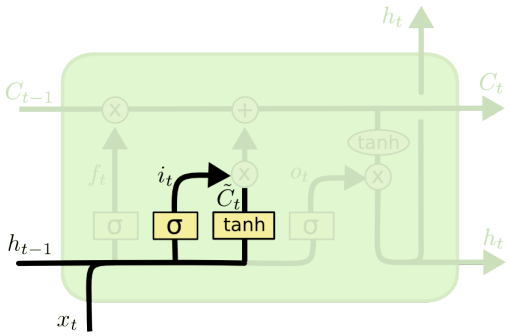


Figure 29: External input gate [40].

The next step is to decide what new information we're going to store in the cell state. This has two parts. First, a sigmoid layer called the external input gate decides which values we'll update.

Next, a *tanh* layer creates a vector of new candidate values, $\tilde{\mathbf{C}}^{(t)}$, that could be added to the state. In the next step, we'll combine these two to create an update to the state.

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting [40]. The formal definition of external input gate can be done:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right) \tag{123}$$

$$\tilde{C}_i^{(t)} = \tanh\left(b_i^C + \sum_j U_{i,j}^C x_j^{(t)} + \sum_j W_{i,j}^C h_j^{(t-1)}\right) \tag{124}$$

2.12.4 Updating Cell State

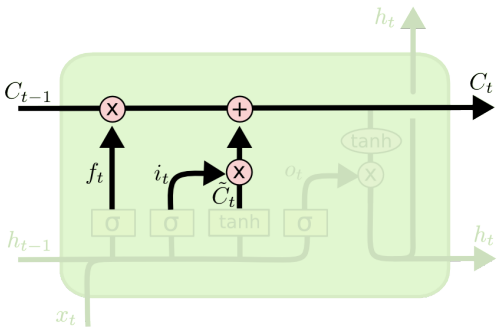


Figure 30: Cell state updating [40].

It's now time to update the old cell state: $\mathbf{C}^{(t-1)} \rightarrow \mathbf{C}^{(t)}$. We multiply the old state by $\mathbf{f}^{(t)}$ forgetting the things we decided to forget earlier. Then we add

$$\mathbf{i}^{(t)} \odot \tilde{\mathbf{C}}^{(t)} \tag{125}$$

$$C_i^{(t)} = f_i^{(t)} C_i^{(t-1)} + g_i^{(t)} \tilde{C}_i^{(t)} \tag{126}$$

This is the new candidate values, scaled by how much we decided to update each state value. In the case of the language model, this is where we'd actually drop the information about the old subject's gender and add the new information, as we decided in the previous steps [40].

2.12.5 Output Gate

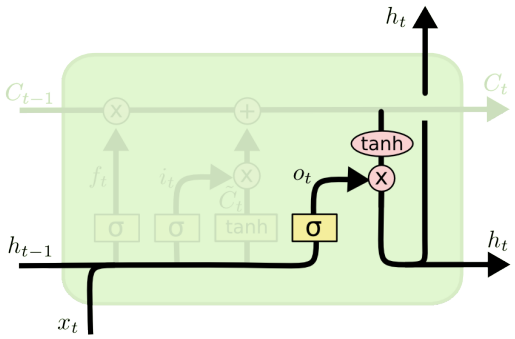


Figure 31: Output gate [40].

Finally, we need to decide what we're going to output. This output will be based on our cell state, but will be a filtered version. First, we run a sigmoid layer which decides what parts of the cell state we're going to output. Then, we put the cell state through tanh (to push the values to be between -1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to [40].

$$\mathbf{C}^{(t)} \odot \mathbf{o}^{(t)} \tag{127}$$

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next [40]. The formal definition of output gate can be done:

$$o_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t)}\right) \tag{128}$$

$$h_i^t = \tanh(C_i^{(t)}) o_i^{(t)} \tag{129}$$

2.13 Gated Recurrent Unit

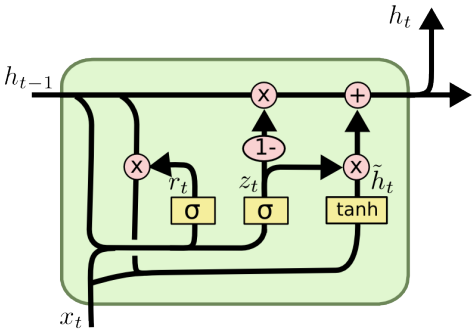


Figure 32: Gated Recurrent Unit.

A different modification of LSTM is Gated Recurrent Unit (GRU) [42]. It combines the forget and input gates into a single "update gate." It also merges the cell state and hidden state, and makes some other changes. The resulting model is simpler than standard LSTM models, and has

been growing increasingly popular [40].

$$z_i^{(t)} = \sigma\left(b_i^z + \sum_j W_{i,j}^z x_j^{(t)} + \sum_j W_{i,j}^z h_j^{(t-1)}\right) \tag{130}$$

$$r_i^{(t)} = \left(b_i^r + \sum_j W_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t-1)}\right) \tag{131}$$

$$\tilde{h}_i^{(t)} = \tanh\left(b_{hh}^r + r^{(t)} \sum_j W_{i,j}^{hh} h_j^{(t-1)} + \sum_j W_{i,j}^{hh} x_j^{(t)}\right) \tag{132}$$

$$h_i^{(t)} = (1 - z^{(t)}) * h^{(t-1)} + z^{(t)} \tilde{h}^{(t)} \tag{133}$$

2.14 Recursive Neural Networks

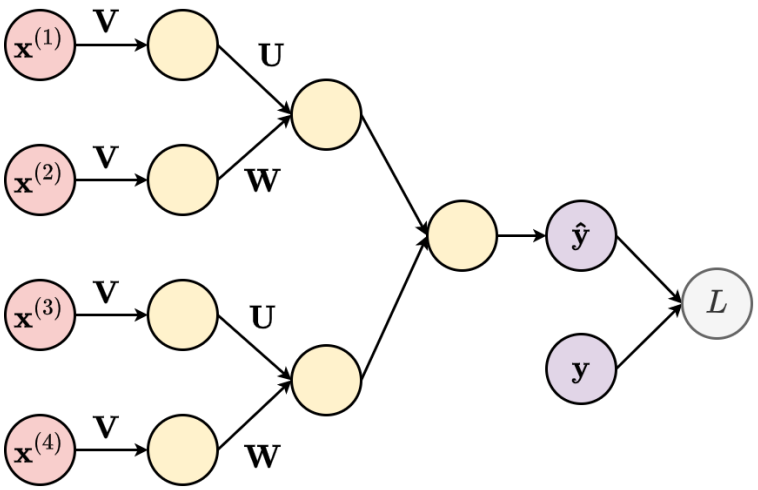


Figure 33: Recursive Neural Network.

Recursive Neural Networks are special design of neural networks to represent and process a sentence in the tree form rather than the chain-like structure of RNNs. Normally, almost every linguistics approach includes tree data structure. In some applications in Natural Language Processing, choosing a Recursive Neural Network might be best to represent the task, i.e., Recursive Neural Networks can easily handle with parse trees, morphological trees etc.

2.15 LSTMs with PyTorch

```python
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 1

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 1])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 1,
    bidirectional = False,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 128])
print(hidden.shape)
# torch.Size([1, 16, 128])
print(cell.shape)
# torch.Size([1, 16, 128])
```

(a) One Layer Unidirectional LSTM

```python
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 4,
    bidirectional = False,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 128])
print(hidden.shape)
# torch.Size([4, 16, 128])
print(cell.shape)
# torch.Size([4, 16, 128])
```

(b) Multilayer Unidirectional LSTM

Figure 34

```python
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 1,
    bidirectional = True,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 256])
print(hidden.shape)
# torch.Size([2, 16, 128])
print(cell.shape)
# torch.Size([2, 16, 128])
```

(a) One Layer Bidirectional LSTM

```python
import torch
import torch.nn as nn

batch_size = 16
seq_len = 512
input_size = 64

input = torch.randn(batch_size, seq_len, input_size)
print(input.shape)
# torch.Size([16, 512, 64])

lstm = nn.LSTM(
    input_size = input_size,
    hidden_size = 128,
    num_layers = 4,
    bidirectional = True,
    batch_first = True)

output, (hidden, cell) = lstm.forward(input)

print(output.shape)
# torch.Size([16, 512, 256])
print(hidden.shape)
# torch.Size([8, 16, 128])
print(cell.shape)
# torch.Size([8, 16, 128])
```

(b) Multilayer Bidirectional LSTM

Figure 35

3 Neural Machine Translation

One of the main usage of RNNs is (was...) Neural Machine Translation (NMT). In NMT, our task is translating source language to target language, by using encoder-decoder RNNs. Encoder-decoder RNNs generally noted as sequence-to-sequence models, which is just a combination of 2 RNN model. The first one is used for encoding the source language to a hidden vector, the latter one is used for decoding it to the target language. The encoder of the seq2seq NMT model might be bidirectional or unidirectional, however, the decoder must be used as a autoregressive model. Hence, it must be a unidirectional RNN. At training time, our model acts like a POS tagging scheme. We calculate each token's loss and sum them, then backpropagate. For example $J^{(1)}(\theta)$ is negative log-probability of "hayatın" (end-to-end training). Or, $J^{(1)}(\theta)$ is negative log-probability of special token "<END>".
NMT directly calculates $p(y \mid x)$:

$$p(y \mid x) = p(y^{(1)} \mid x) \cdot p(y^{(2)} \mid y^{(1)}, x) \cdot p(y^{(3)} \mid y^{(2)}, y^{(1)}, x) \cdot ... \cdot p(y^{(T)} \mid y^{(T-1)}, y^{(T-2)}, ..., y^{(1)}, x) \tag{134}$$

Each component is probability of next target word given target words so far and source sentence x. The objective is, as can be seen,

$$\arg\max_y p(y \mid x)$$

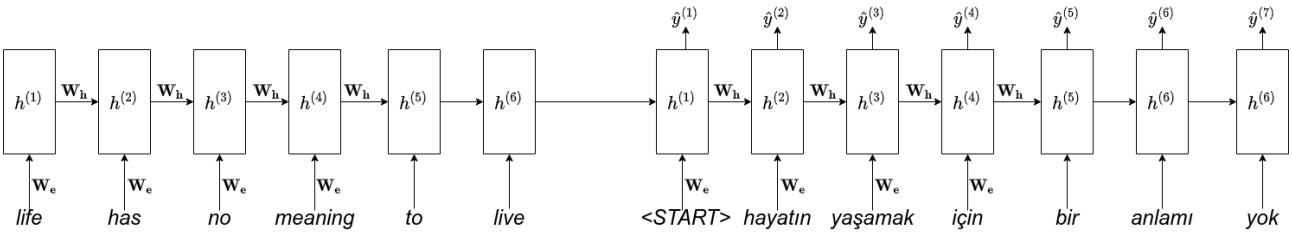


Figure 36: NMT

However, at inference time, we pass the source sentence to translate. Then, we generate the words in autoregressive fashion. Each generated word is passed to $t+1$th step as an input, then produce the $t+1$th generated sentence.

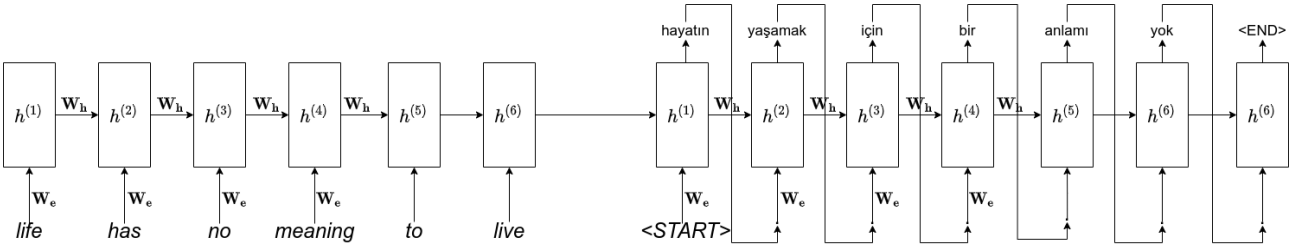


Figure 37: NMT

We call it, this is **greedy decoding**. Taking the most probable word on each step. There are several decoding strategies in natural language processing. So, what is wrong with greedy decoding?

3.1 Decoding Strategies

3.1.1 Greedy Decoding

Greedy decoding has no way to undo decisions. Also we are not guaranteed that, there will be <END> token at decoding state. On the other hand, if we get a <END> token, there is still a possibility that it may not be the most optimal solution.

input: life has no meaning to live.
→ hayatın .....
→ hayatın yaşamak .....
→ hayatın yaşamak için .....
→ hayatın yaşamak için turtası ..... (???? no going back now)

What we are doing in greedy decoding actually is taking arg max of each output, so we have one output with highest probability, but this does not mean that it will converge to highest overall probability.

3.1.2 Exhaustive Search

Could we try computing all possible sentences y? Well yes but actually no. This has $O(\mid V \mid^T)$ complexity.

3.1.3 Beam Search

On each step of decoder, keep track of the k most probable partial translations (which we call hypotheses).

- k is the beam size (in practice: 5-10).
- A hypothesis $y_1, ..., y_t$ has a score which is log probability.
  - We search for high-scoring hypotheses, tracking top k on each step.
- Beam search is not guaranteed to find optimal solution.

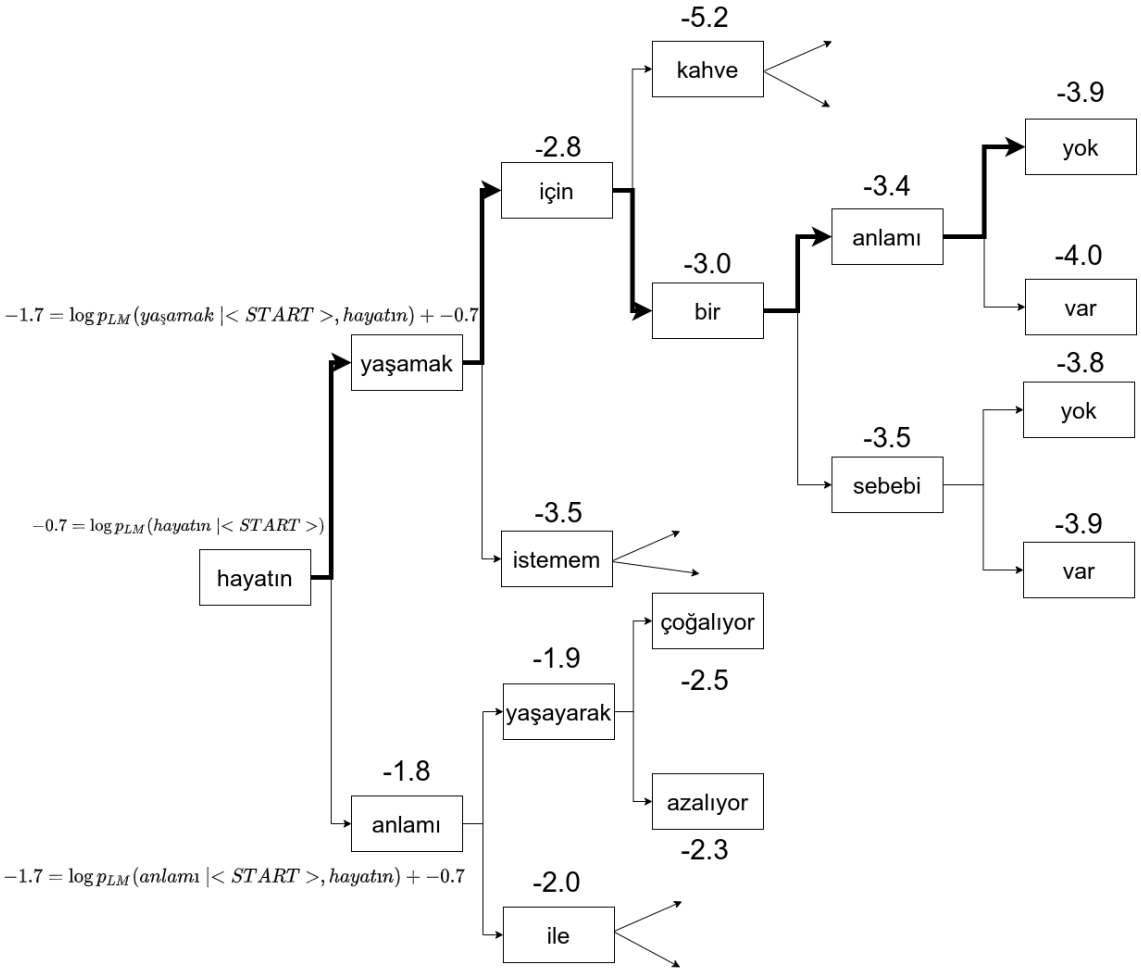


Figure 38: Beam Search

As you can see, Greedy Decoding chooses [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”anlamı”, ”yok”] with score of -3.9. But it is not the most probable sentence. With Beam Search, with beam size = 2, we can select the most probable sentence (in the space of beam size = 2), [”hayatın”, ”yaşamak”, ”için”, ”bir”, ”sebebi”, ”yok”].

Beam Search: Stopping Criterion

- In greedy decoding, we usually decode until the model produces <END> token.
- In Beam Search, different hypotheses may produce <END> tokens on different timesteps.
  - When a hypothesis produces <END>, that hypothesis is complete.
  - Place it aside and continue exploring other hypotheses via Beam Search.
- Usually we continue Beam Search until:
  - We reach timestep T (pre-defined) or,
  - We reach at least n completed hypotheses (pre-defined).

Beam Search: Finishing Up

- We have our list of completed hypotheses.

- How to select top one with highest score?
- **Choosing shortest one is a problem**: longer hypotheses have lower scores.
- So, normalize by length!

$$\frac{1}{t}\sum_{i=1}^{t}\log p_{\text{LM}}(y_i \mid y_{i-1}, ..., y_1, x)$$

3.2 BLEU Score

Turkish: kedi paspasın üzerinde
Reference 1: the cat is on the mat
Reference 2: there is a cat on the mat
Candidate: the cat the cat on the mat

3.2.1 Unigram Precision

Unigram	Shown In References?
the	1
cat	1
the	1
cat	1
on	1
the	1
mat	1

Then, merge the unigram counts:

Unique Unigram	Count
the	3
cat	2
on	1
mat	1

The total number of counts for the unique unigrams in the candidate sentence is 7, and the total number of unigrams in the candidate sentence is 7. The unigram precision is $7/7 = 1.0$

3.2.2 Bigram Precision

Bigram	Shown In References?
the cat	1
cat the	0
the cat	1
cat on	1
on the	1
the mat	1

Then, merge the bigram counts:

Unique Bigram	Count
the cat	2
cat the	0
cat on	1
on the	1
the mat	1

The total number of counts for the unique bigrams in the candidate sentence is 5, and the total number of bigrams in the candidate sentence is 6. The bigram precision is $5/6 = 0.833$.

3.2.3 Calculating BLEU

$$\text{BLEU} = \text{BP} \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right)$$

and

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ \exp\left(1 - \frac{r}{c}\right) & \text{if } c \leq r \end{cases}$$

Where p_n is the precision of n-gram, with natural logarithm, w_n is the weight between 0 and 1, with constraint of $\sum_{i=1}^{n} w_i = 1$, and BP is the brevity penalty to penalize short machine translations. Where c is the number of unigrams (length) in all the candidate sentences, and r is the best match lengths for each candidate sentence in the corpus. BLEU generally uses $N = 4$ and $w_n = \frac{1}{n}$.

3.2.4 Proof of BLEU $\in [0, 1]$

$$\begin{aligned} \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) &= \prod_{n=1}^{N} \exp\left(w_n \log p_n\right) \\ &= \prod_{n=1}^{N} \left[\exp\left(\log p_n\right)\right]^{w_n} = \prod_{n=1}^{N} {p_n}^{w_n} \\ &\in [0, 1] \end{aligned}$$

4 Attention for Recurrent Layers

Attention is invented for mainly alignment of source-target language. In Statistical Machine Translation (SMT), the problem of alignment is done by hand-crafted methods or phrase-to-phrase methods. With neural attention, models can learn the alignment by gradient based methods.

4.1 The Problem of Alignment

Since Statistical Machine Translation, alignment between source sentence's words and target sentence's words is a main problem in research. There are many alingment types and some of them are complex.

4.1.1 one-to-one Alignment

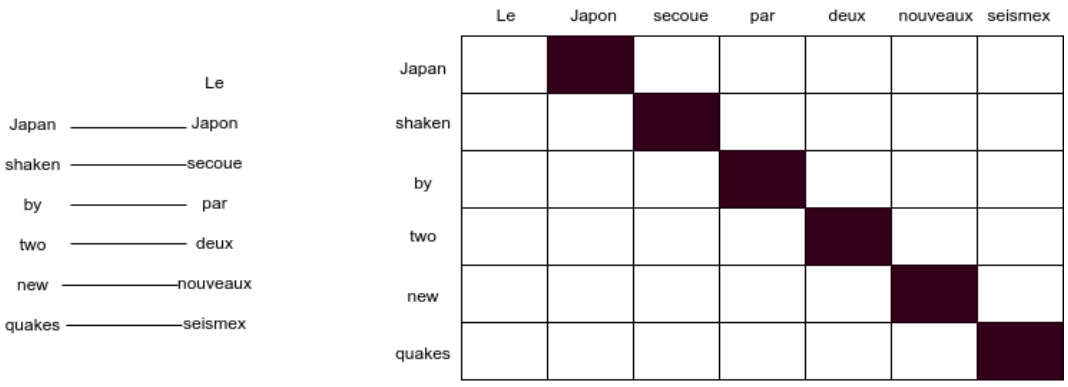


Figure 39: one-to-one

We call "Le" as "spurios" word.

4.1.2 many-to-one Alignment

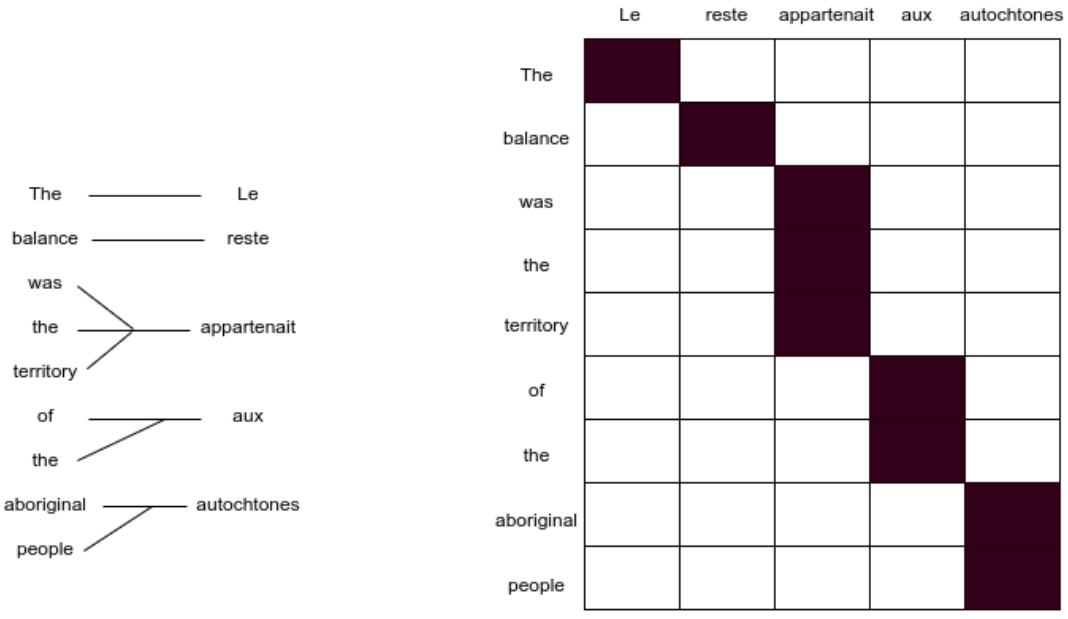


Figure 40: many-to-one

4.1.3 one-to-many Alignment

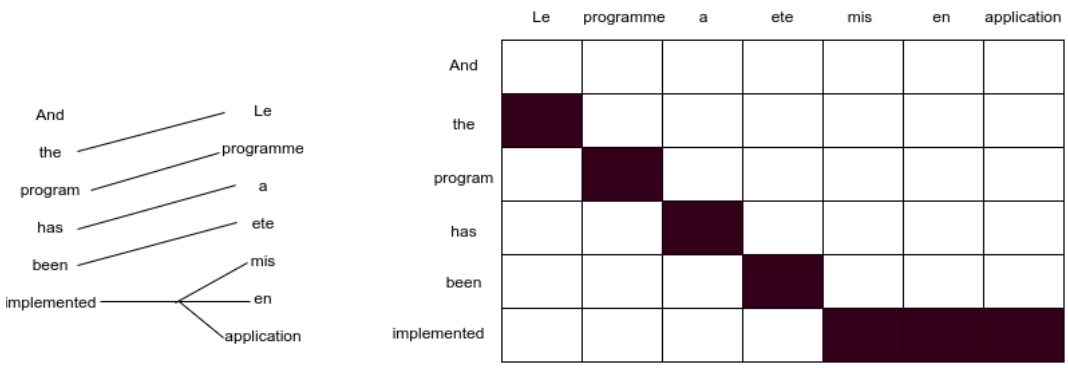


Figure 41: one-to-many

We call "implemented" as "fertile" word.

4.1.4 many-to-many Alignment

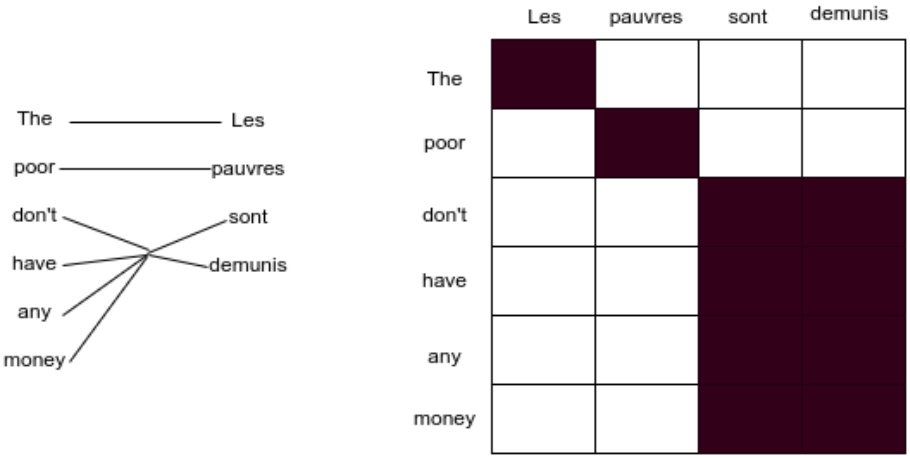


Figure 42: many-to-many

4.2 Fixing Problems with Attention Mechanism

The attention mechanism was born to help memorize long source sentences in neural machine translation (NMT). Rather than building a single context vector out of the encoder's last hidden state, the secret sauce invented by attention is to create shortcuts between the context vector and the entire source input. The weights of these shortcut connections are customizable for each output element. While the context vector has access to the entire input sequence, we don't need to worry about forgetting. The alignment between the source and target is learned and controlled by the context vector. Essentially the context vector consumes three pieces of information; encoder hidden states, decoder hidden states, alignment between source and target. [43]

4.3 Neural Machine Translation by Jointly Learning to Align and Translate

Neural Machine Translation with attention mechanism allows us to learn alignments insted of defining phrase based alignments as in Statistical Machine Translation. Attention mechanism is not only for NMT, it can be applied into Language Modeling, Question Answering and other tasks

in NLP. Also, since we showed alignments in Statistical Machine Translation, the attentions are binary, not weighted. In linguistics, words are related and dependent, and dependencies may have importance. Learning attention in NMT also allows us to learn those weighted dependencies. In Neural Machine Translation by Jointly Learning to Align and Translate [43], the additive attention is proposed. The illustration of this mechanism can be shown,

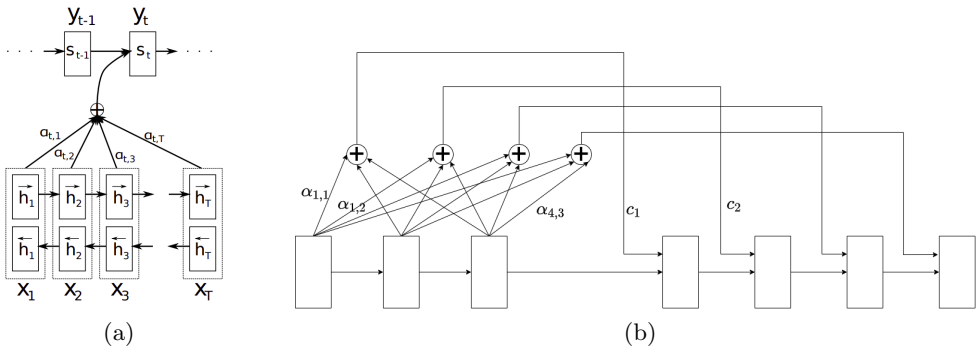


Figure 43: Additive Attention

To make a formal definition, we have input sequence $\mathbf{x}$ with length of n and output sequence $\mathbf{y}$ with length of m,

$$\mathbf{x} = [x_1, x_2, ..., x_n] \tag{135}$$

$$\mathbf{y} = [y_1, y_2, ..., x_m] \tag{136}$$

Bi-directional encoder state is

$$\boldsymbol{h}_i = [\overrightarrow{\boldsymbol{h}}_i^\top; \overleftarrow{\boldsymbol{h}}_i^\top]^\top, i = 1, \dots, n \tag{137}$$

Let's denote decoder hidden states as s_t which is a function of

$$\boldsymbol{s}_t = f(\boldsymbol{s}_{t-1}, y_{t-1}, \mathbf{c}_t)$$

The context vector c_t is defined as

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \mathbf{h_i} \tag{138}$$

The term $\alpha_{t,i}$ represents "How well two words y_t and x_i are aligned", and defined as

$$\alpha_{t,i} = \text{align}(y_t, x_i) = \frac{\exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_i))}{\sum_{j=1}^{n} \exp(\text{score}(\mathbf{s}_{t-1}, \mathbf{h}_j))} \tag{139}$$

The alignment model assigns a score $\alpha_{t,i}$ to the pair of input at position i and output at position t, (y_t, x_i), based on how well they match. The set of $\alpha_{t,i}$ are weights defining how much of each source hidden state should be considered for each output. Since this alignment is built with softmax, it can be seen as a classification between pairs. Score function is a scoring function between (y_t, x_i) pairs.

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i]) \tag{140}$$

where both $\mathbf{v}$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model. The matrix of alignment scores is a nice byproduct to explicitly show the correlation between source and target words.

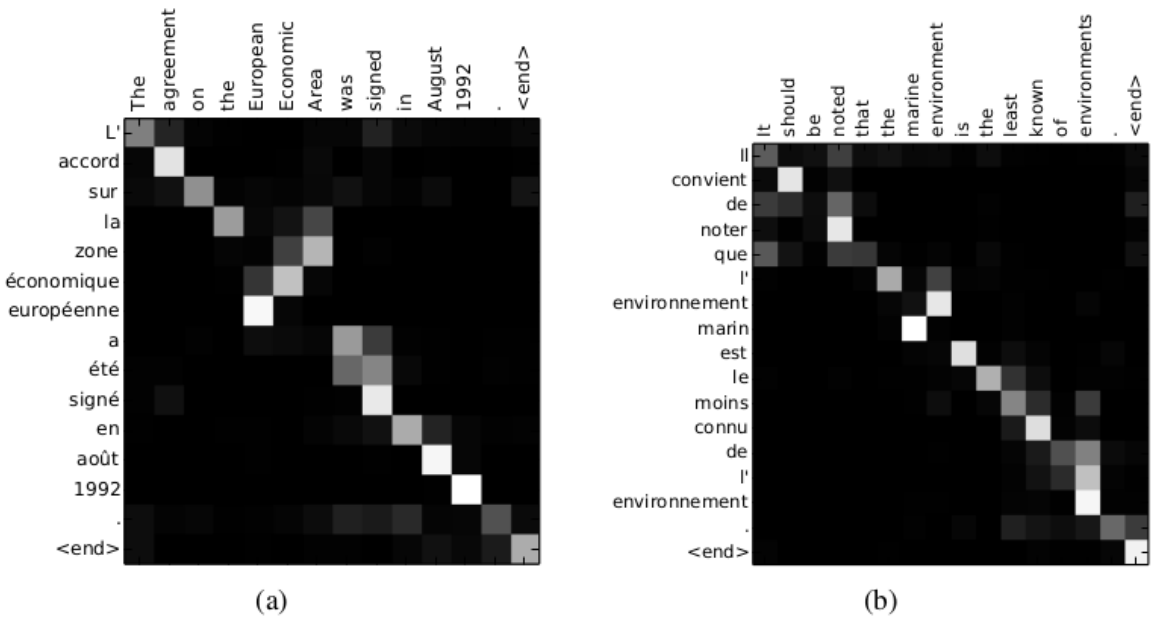


Figure 44: illustrated attention [43]

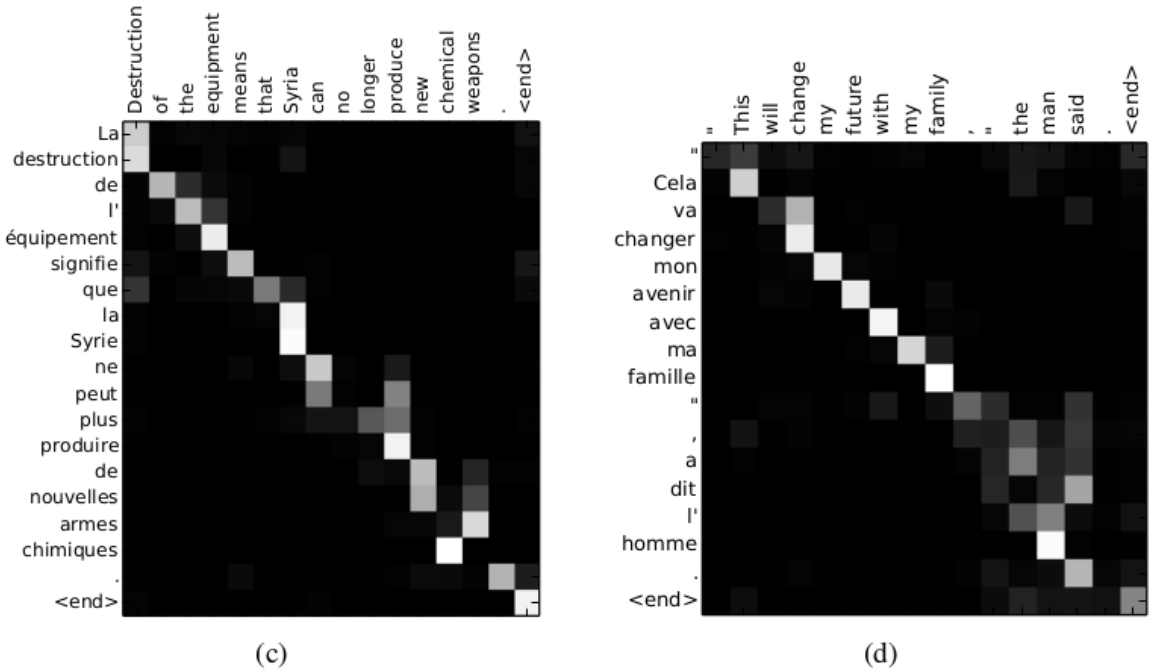


Figure 45: illustrated attention [43]

4.4 Types Of Attention Mechanism (source)

- **Content-base attention**

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$$

- **Location-Base**:

$$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$$

- **General**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$$

- **Dot-Product**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$$

- **Scaled Dot-Product**:

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$$

5 Contextualized Representations

Meaning of a word can differ from context to context. Or, words can have different meanings that derive from a common origin (polysemous word). For example, in English the word "pike" has nearly 9 meanings. Suppose that, our training corpus have word "pike", do word2vec or other word represetation methods capture this 9 meanings? In [44], authors proposed a linear superposition in standard word embeddings like word2vec. It is a powerful method but requires labeling for polysemous words. For example,

$$v_{\text{pike}} = \alpha_1 \times v_{\text{pike}_1} + ... + \alpha_9 \times v_{\text{pike}_9} \tag{141}$$

where

$$\alpha_n = \frac{\text{freq}(\text{pike}_n)}{\sum_{i=0}^{9} \text{freq}(\text{pike}_i)} \tag{142}$$

Although it improves performance, tagging all polysemous words is not effective. And yet, context of word can change whether it is polysemous or not. Word representation methods like word2vec or gloVe are global word representations. Contextual word representation don't treat words as global. Meaning of a word can be changed by its context: the sentence in which the word occurs. We have only one vector for "composer". In contextual representations, we don't have fixed word vectors.

5.1 Deep Contextualized Word Representations

ELMo (Deep contextualized word representations) [35] was proposed to use contextualized representations of words (in addition, ELMo was not the first one that used contextualized representations, see [45] and [46]). ELMo uses vectors derived from a bidirectional LSTM that is trained with a coupled language model (LM) objective on large text corpus. ELMo word representations are functions of the ENTIRE input sentence. They are computed on top of two-layer biLM with character convolutions.

Given a sequence of N tokens $(t_1, t_2, ..., t_N)$, a forward language model computes the probability of the sequence by modeling the probability of token t_k, given the history $(t_1, t_2, ...t_{k-1})$:

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_1, t_2, ..., t_{k-1}) \tag{143}$$

At each position k, ach LSTM layer outputs a context-dependent representation $\vec{\mathbf{h}}_{k,j}^{LM}$ where $j = 1, 2, ..., L$ (number of layers). Top layer LSTM output, $\vec{\mathbf{h}}_{k,L}^{LM}$ is used to predict next token t_{k+1} with softmax. The equations that we formulated above are for forward LM. The backward LM is similar to forward LM.

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N) \tag{144}$$

with each backward LSTM layer j in a L layer deep model producing representations $\overleftarrow{\mathbf{h}}_{k,j}^{LM}$ of t_k given $(t_{k+1}, t_{k+2}, ..., t_N)$. And the formulation jointly maximizes the log-likelihood of the forward and backward directions:

$$\sum_{k=1}^{N} \log p(t_k \mid t_1, t_2, ..., t_{k-1}; \Theta_x, \vec{\Theta}_{LSTM}, \Theta_s) + \log p(t_k \mid t_{k+1}, t_{k+2}, ..., t_N; \Theta_x, \overleftarrow{\Theta}_{LSTM}, \Theta_s) \tag{145}$$

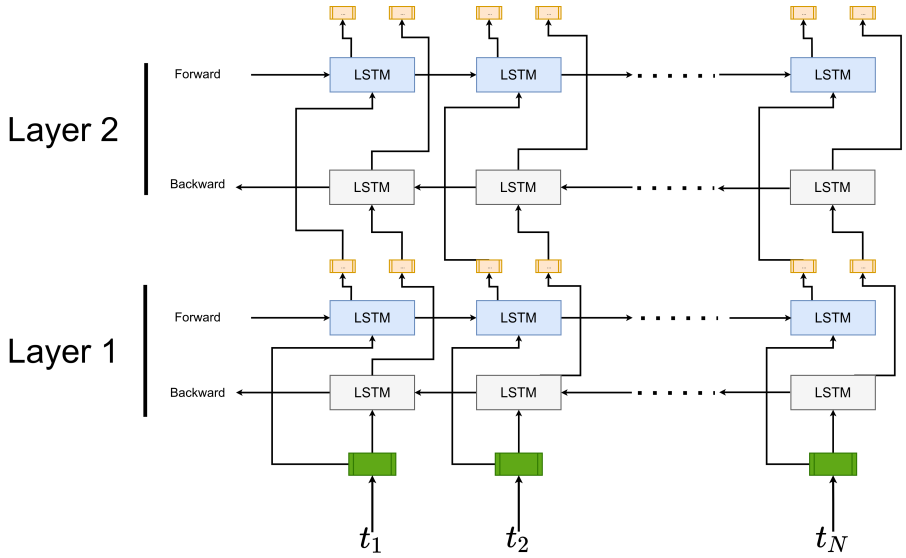


Figure 46: ELMo Architecture.

ELMo is a task specific combination of the intermediate layer representations in the biLM. Higher-level LSTM states capture context-dependent aspects of word meaning (word sense disambiguation tasks) while lower level states model aspects of syntax (POS tagging). For each token t_k, a L-layer biLM computes a set of $2L + 1$ representations (forward + backward + x_k)

$$R_k = \{\mathbf{x}_k^{LM}, \vec{\mathbf{h}}_{k,j}^{LM}, \overleftarrow{\mathbf{h}}_{k,j}^{LM} \mid j = 1, 2, ..., L\} \tag{146}$$

$$= \{\mathbf{h}_{k,j}^{LM} \mid j = 0, 1, ..., L\} \tag{147}$$

where $\mathbf{h}_{k,0}^{LM}$ is the token layer and $\mathbf{h}_{k,j}^{LM} = [\vec{\mathbf{h}}_{k,j}^{LM}; \overleftarrow{\mathbf{h}}_{k,j}^{LM}]$, for each biLSTM layer. For inclusion in a downstream model, ELMo collapses all layers in R into single vector, $\text{ELMo}_{\mathbf{k}} = E(R_k; \Theta_e)$. In the simplest case, ELMo just selects the top layer $E(R_k) = \mathbf{h}_{k,l}^{LM}$.

In later chapters, we are going to examine Transformer model, also plays a role as contextual word representation model. However, we are not going to examine the details of this model in this chapter.

6 Subword Models

The methods that we have talked about are very effective to find good and dense word representations. But handling not-seen words is hard. We train those models on a specific corpus, and language is infinite. Suppose that we have words "fast", "faster", "fastest" and "long", but "longer", and "longest" not seen in this corpus. After learning word representations, how can we represent "longer" and "longest" when they are not in the training set? The theory of Derivational Morphology helps us here. Morphology is basically internal structure of words and forms. Words are meaningful linguistics units that can be combined to form phrases and sentences.

For example, look at the sentence "Bach composed the piece". The word "Bach" is a free morpheme (lexical morpheme), a free morpheme is a morpheme that can stand alone as a word. The word composed can be decomposed to "compose" and "-d". The suffix "-d" is a past marker here. In linguistics, we call "-d" grammatical morpheme or bound. The word "the" in the sentence is still a grammatical morpheme but it is not bound, it is called independent words.

So, Derivational Morphology says that, new words enter language in two main ways - through the addition of words unrelated to any existing words and derivational morphology, the creation of new open-class words (open-class word or lexical content word is basically nouns, lexical verbs, adjectives, and adverbs) by the addition of morphemes to existing roots. Derivational morphemes increase the vocabulary and may allow speakers to convey their thoughts in a more interesting manner, but their occurence is not related to sentence structure [47].

From this point, we should find a method to overcome out-of-vocabulary (OOV) problem in NLP.

6.1 Character Level Models

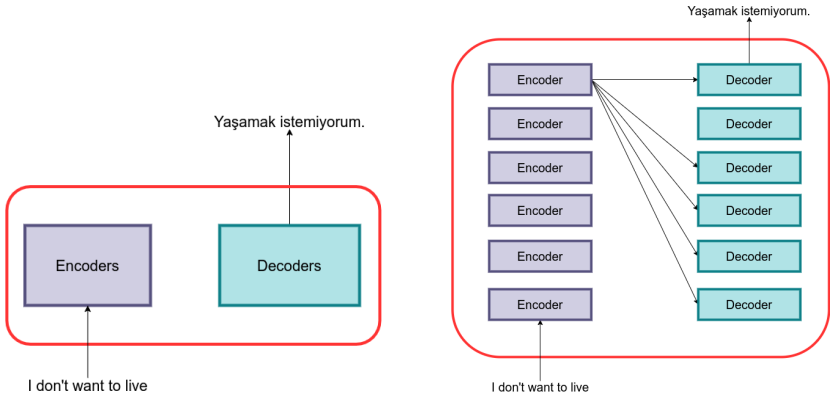
6.2 Hybrid Models

6.3 Byte Pair Encoding

7 Transformers

Transformer [48] is a architecture for sequential processing of the sequential data. However, it can be used for vision tasks to. Not to deviate from the subject, Transformer is built on a attention mechanism called "self attention".
To comparing with NMT-RNN, as introduced in NMT chapter, Transformer is a sequence-to-sequence model as well. However, it is purely built on attention mechanism. Besides, all calculations happen at once. RNNs have a time dependence at computing level: first we calculate $\hat{y}_1$, then $\hat{y}_2$. Transformer calculates $\hat{y}_1$ and $\hat{y}_2$ at once. Hence, it is more parallelizable and requiring significantly less time to train.

Transformer has encoder and decoder. The encoder part of the Transformer is bidirectional, the decoder part of the Transformer is unidirectional. To compare with NMT-RNN, we pass the source sentence to Transformer's encoder, and Transformer's decoder decodes encoder's output to target sentence. The encoding component is a stack of encoders. The decoding component is a stack of decoders of the same number.



(a) Folded Transformer (b) Unfolded Transformer

Figure 47: The Transformer.

But, what are the components of this encoder/decoder stacks? Encoder is used for encoding bidirectional context of the source sentence; by looking previous, current and future words. It picks "interesting parts" of the source sentence and outputs those interesting parts' contextualized representation. We call it self-attention. Then, the decoder uses those "interesting parts".

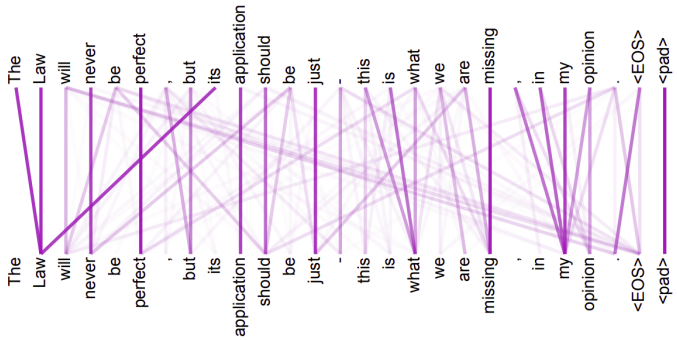


Figure 48: Transformers

Self-attention discovers a some kind of alignment inside the source sentence. For example in Figure 15, self-attention discovers the word "Law" is related with word "its" in some way.

7.1 Self-Attention with Intuition

Before diving into the components of the self-attention, we should examine this procedure intuitively.

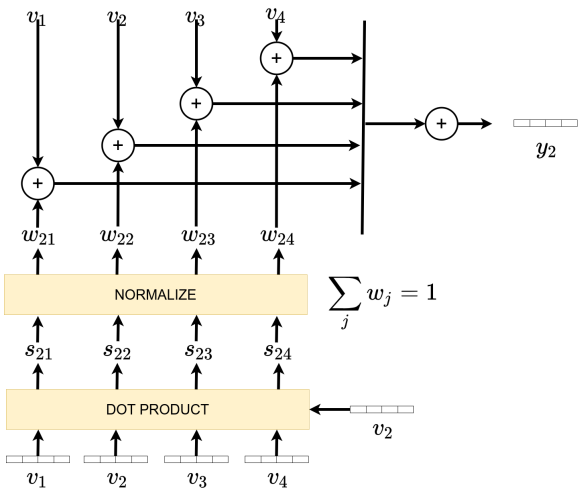


Figure 49: Reweighing

Self-attention actually is a reweighing all vectors (vectors of tokens in our case), towards current vector. Consider the vector v_2. First we calculate the dot product between v_2 and other vectors. By doing this, we obtain non-scaled similarities between v_2 and other vectors (remember from cosine similarity function). However, those similarity values are independent from each other. So, we should do some kind of normalizing (can be a single softmax). With normalized values, we have some kind of "contextualized" similarity values.

To obtain contextualized representation of v_n, we product those normalized values with related vector, then sum up. This procedure tells us "vector v_i is related with v_n with weight of w_{ni}, so I should multiply v_i with w_{ni} to obtain importance of v_i by looking v_n. If I do this for all v_i and sum them, I will obtain the contextualized version of v_n by looking all v_is". If we formulate this for v_1:

$$v_1 \cdot v_1 = \tilde{w}_{11} \rightarrow normalize(w_{11}) \tag{148}$$
$$v_1 \cdot v_2 = \tilde{w}_{12} \rightarrow normalize(w_{12}) \tag{149}$$
$$v_1 \cdot v_3 = \tilde{w}_{13} \rightarrow normalize(w_{13}) \tag{150}$$
$$v_1 \cdot v_4 = \tilde{w}_{14} \rightarrow normalize(w_{14}) \tag{151}$$
$$y_1 = w_{11} \cdot v_1 + w_{12} \cdot v_2 + w_{13} \cdot v_3 + w_{14} \cdot v_4 \tag{152}$$

This procedure applies to all v_is.

$$y_2 = w_{21} \cdot v_1 + w_{22} \cdot v_2 + w_{23} \cdot v_3 + w_{24} \cdot v_4 \tag{153}$$
$$y_3 = w_{31} \cdot v_1 + w_{32} \cdot v_2 + w_{33} \cdot v_3 + w_{34} \cdot v_4 \tag{154}$$
$$y_4 = w_{41} \cdot v_1 + w_{42} \cdot v_2 + w_{43} \cdot v_3 + w_{44} \cdot v_4 \tag{155}$$
$$\tag{156}$$

At the end, we have now y_1, y_2, y_3, y_4 with better and more context.

However, we do not perform any "learning" here. We need to learn this reweighing operation with linguistics features. In Transformers, instead using same v_i vectors, we use "keys, queries and values" to represent embeddings in 3 different ways. Keys, queries and values are simply linear transformations with fully-connected layers. We select most related key with queries, then select values with this key. The first step in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word).

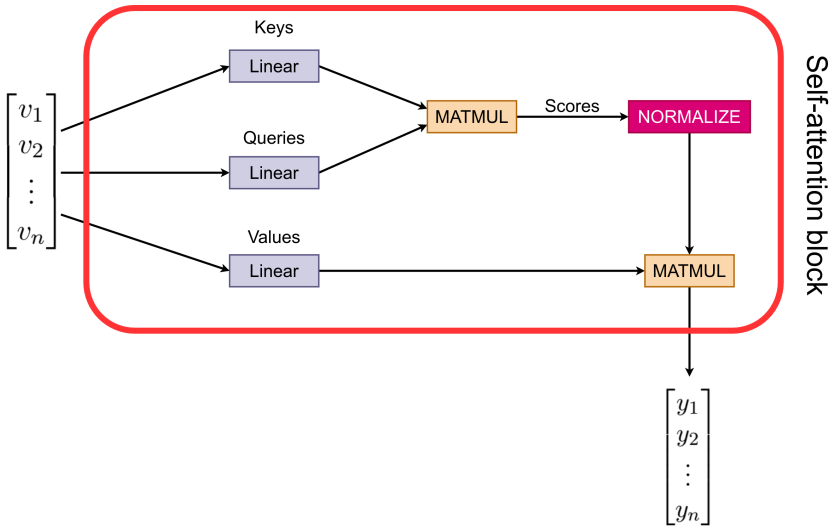


Figure 50: Self-attention block

So for each word, we create a query vector q, a key vector k, and a value vector v. These vectors are created by multiplying the embedding by three matrices that we trained during the training process. Notice that these new vectors are smaller in dimension than the embedding vector. For example, if their dimensionality is 64, the embedding and encoder input/output vectors have dimensionality of 512. They don't have to be smaller, this is an architectural choice to make the computation of multiheaded attention (mostly) constant.

By learning query, key and value vectors, we can learn and encode linguistics features of the sentence by performing reweighing scheme. The only difference between Figure 16 and 17 is learning the keys, queries and values.

7.2 Multihead Attention

Ensemble always wins. What about using multiple self-attention blocks? Authors of the Transformer paper proposed multihead self-attention. Instead using only one block, we have 8 different self-attention block. This allows us to see and learn different subspaces of linguistic representations.

For example first head of the multihead self-attention can learn noun-noun alignments, other one can learn noun-verb alignments.

First of all, the input is passed to 8 different key, query and value mapping; so we have 8 different keys, queries and values. After that, same self-attention procedure is applied for each key, query and value. At the output level, each head is concatenated with each other. After all of that, a linear layer is applied for reducing the dimension of concatenated matrix.

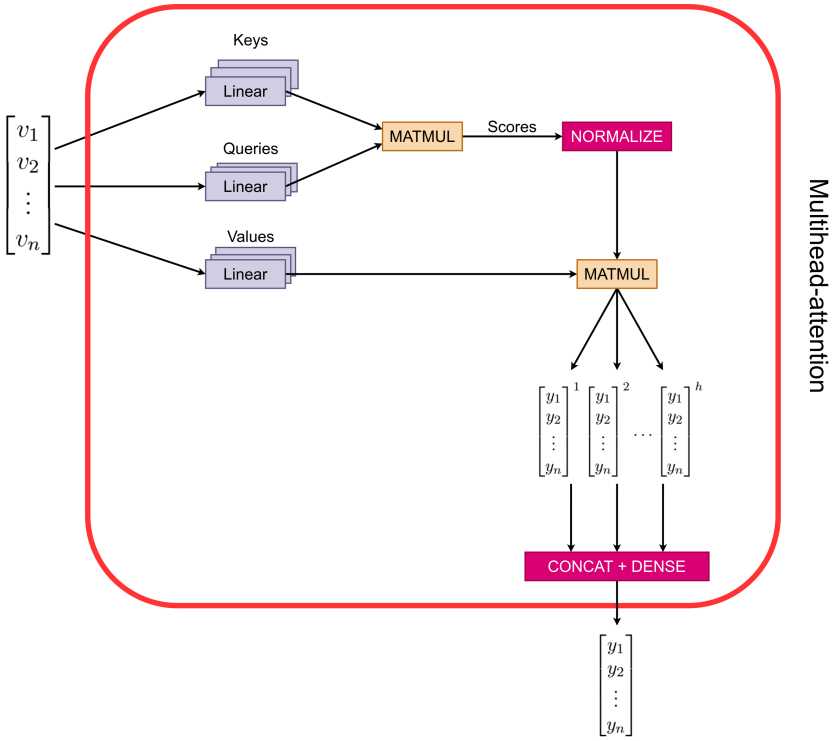


Figure 51: Self-attention block

7.3 Self-Attention Formulation In Nutshell

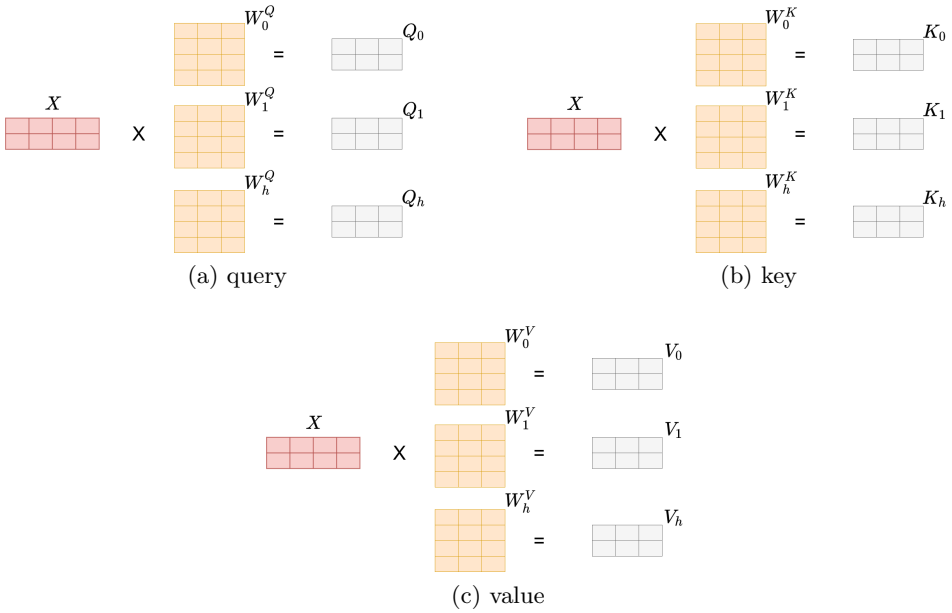
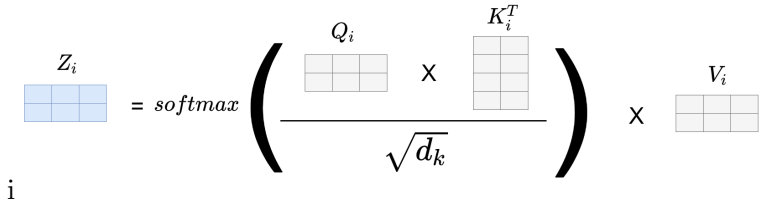


Figure 52: Linear mapping of Key, Queries & Values



i

Figure 53: scaled dot product

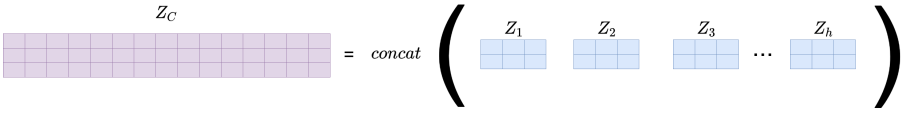


Figure 54: concatenating Z_i

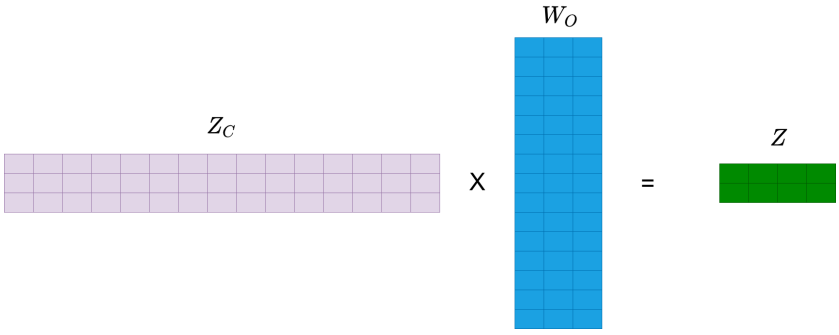


Figure 55: dense

7.4 Self-Attention Dimensions In Nutshell

- $X \in \mathbb{R}^{input\_length \times 512}$
- $W_i^K, W_i^Q, W_i^V \in \mathbf{R}^{512 \times 64}$
- $K_i, Q_i, V_i \in \mathbf{R}^{input\_length \times 64}$
- $Z_i \in \mathbf{R}^{input\_length \times 64}$
- $Z_c \in \mathbf{R}^{input\_length \times (64 \times h)}$
- $W_O \in \mathbf{R}^{(64 \times h) \times 512}$
- $Z \in \mathbf{R}^{input\_length \times 512}$

7.5 Attention Is All You Need

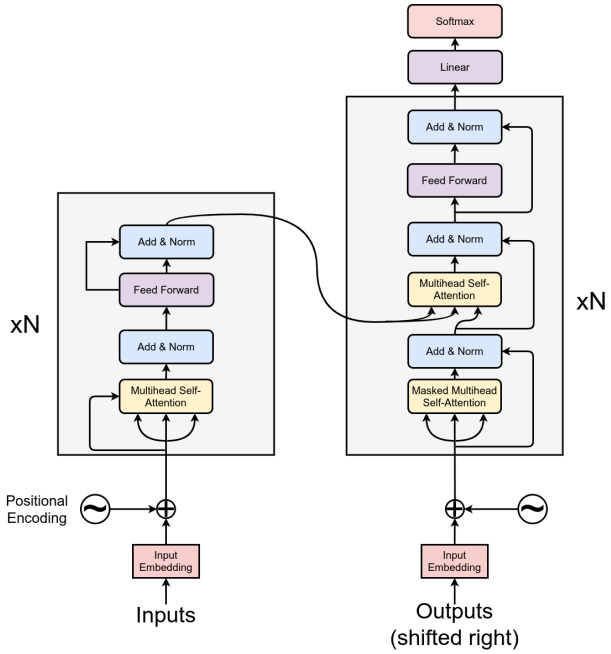


Figure 56: Transformers architecture in Attention Is All You Need (Vaswani et al., 2017)

The Transformer model is introduced by Vaswani et al. in the famous "Attention Is All You Need" paper[48] and is today the de-facto standard encoder-decoder architecture in natural language processing. Now let's define more formally and detailed the encoder and the decoder of Transformers.

7.5.1 Encoder

The encoder layer has multihead attention + residual connections + feedforward layer. It is nearly described in previous sections. The 3 arrows, inputs of multihead attention, are keys, queries & values which was explained.

Encoder has an input of word vectors. The output of the encoder is contextualized encoding sequence:

$$f_{\theta_{\text{encoder}}} : (x_1, x_2, ..., x_n) \rightarrow (z_1, z_2, ..., z_n)$$

Each encoder block consists of a bi-directional self-attention layer, followed by two feed-forward layers.

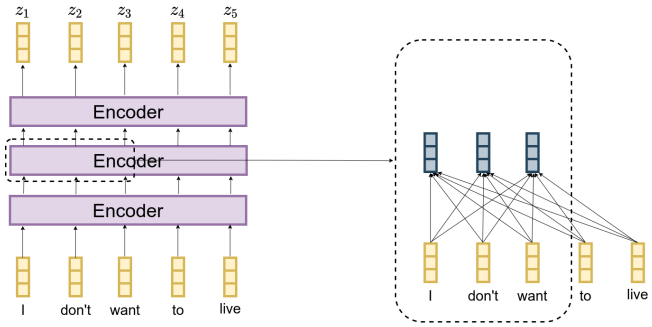


Figure 57: Encoder

As seen in the figure 25, the contextual representation of "don't" is the up-second block at right-most part of figure. The best part of this bidirectionality is now, the word "want" is depends directly on all input word "I", "don't", "want", "to", "live".

7.5.2 Decoder

The decoder structure is nearly same as in the encoder. But the decoder auto-regressively generate the outputs at inference time. At training, all happens at once, loss is calculated like a "part-of-speech tagging" scheme.

A generative language model must be autoregressive, if it looks to future positions, it will be cheating. In the decoder, the self-attention layer is only allowed to attend to earlier positions in the output sequence. This is done by masking future positions (setting them to -inf) before the softmax step in the self-attention calculation.

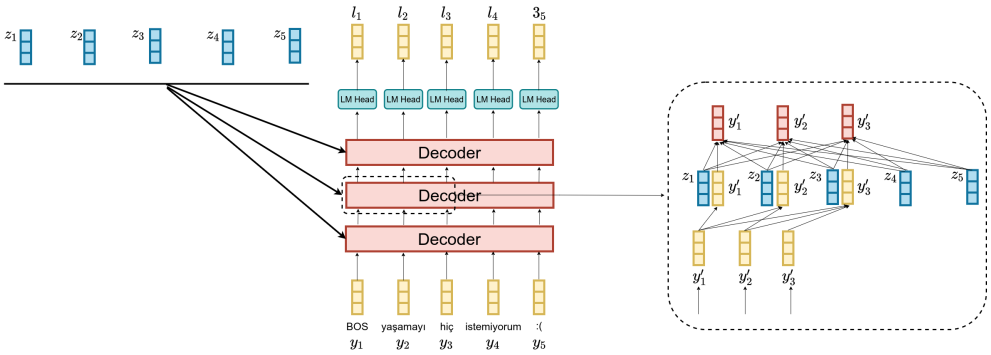


Figure 58: Decoder

The transformer-based decoder hereby maps the sequence of encoded hidden states $z_1, z_2, ..., z_n$ and all previous target vectors $y_1, y_2, ..., y_{i-1}$ to the logit vector l_i. he logit vector l_i is then processed by the softmax operation to define the conditional distribution $p_{\theta_{\text{decoder}}}(y_i \mid y_1, y_2, ..., y_{i-1}, z_1, z_2, ..., z_n)$ just as it is done for RNN-based decoders.
In contrast to transformer-based encoders, in transformer-based decoders, the encoded output vector y_i should be a good representation of the next target vector y_{i+1} and not of the input vector itself. Additionally, the encoded output vector y_i should be conditioned on all contextualized encoding sequence $x_1, x_2, ..., x_n$. To meet these requirements each decoder block consists of a uni-directional self-attention layer, followed by a cross-attention layer and two feed-forward layers.

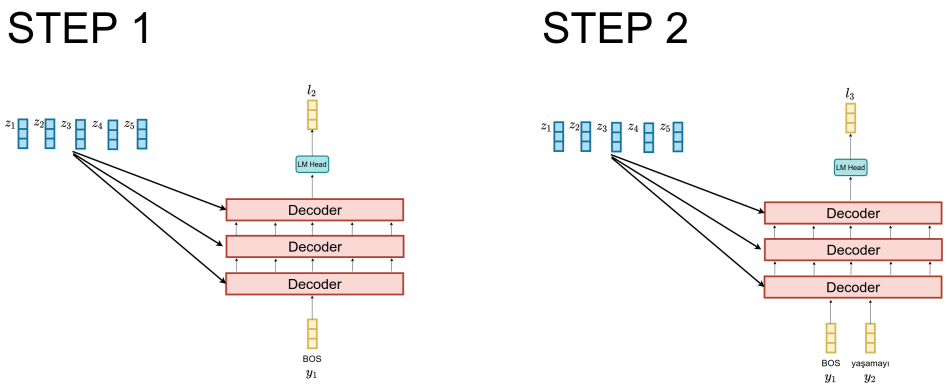


Figure 59: Decoder

7.5.3 Injecting Positional Information to Transformers

8 Language Modeling with Transformers

8.1 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding

BERT [49] is a contextual representation model, which consists of encoder layer of Transformer [48]. Formally, BERT is pre-trained on large corpus for "learning the language". Pre-training allows us to produce high quality word vectors that have contextual representations. After pre-training of BERT, the model can be used for downstream tasks; like sentence classification, question answering, sequence labeling etc.

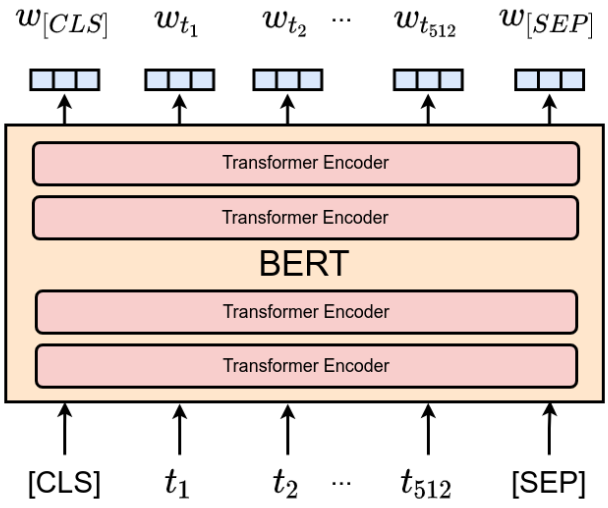


Figure 60: BERT

BERT is bidirectional, since it uses Transformer's encoder. Contextual vector representation of word w_t depends on previous, current and future words. In pre-training, BERT's objective is masked language modeling (MLM) which does not require labeled data. To pre-train BERT, input segments are obtained from an unlabeled corpus and each segment is randomly masked with special token `[MASK]`. BERT is designed for predicting this `[MASK]` token. The masked language model objective is to maximize log-likelihood of observing masked tokens $\tilde{\mathbf{x}}$ with given masked input segment $\hat{\mathbf{x}}$ for sequence length T

$$\max_{\theta} \log p_\theta(\tilde{x} \mid \mathbf{x}) \approx \sum_{t=1}^{T} \mathbb{1}(x_t) \log p_\theta(x_t \mid \hat{\mathbf{x}}) \tag{157}$$

where θ is model's parameters, $\mathbb{1}(x_t)$ is a indicator function that gives 1 for $x_t =$ `[MASK]`, otherwise zero. This masking procedure is done randomly, considering 15% of tokens in input segment. Masking of segments are static. In other words, masking of a specific input segment does not change in each epoch.

Introducing the notation: number of encoder layers as L, hidden size as H, number of self-attention heads as A; $\text{BERT}_{\text{BASE}}$ is $L = 12$, $H = 768$, $A = 12$, $\text{BERT}_{\text{LARGE}}$ is $L = 24$, $H = 1024$, $A = 16$.

Fine-tuning strategy of BERT depends on the downstream task. If we consider a sentence classification task, cotenxtualized representation of `[CLS]` (classification) token generally used. This token is introduced in pre-training. Intuitively, `[CLS]` token is a contextual representation of whole

sentence. It is "crossed" with all tokens in the sentence, and the position in the sequence is not biased. Passing the `[CLS]` to a fully connected layer allows us to classify sentences:

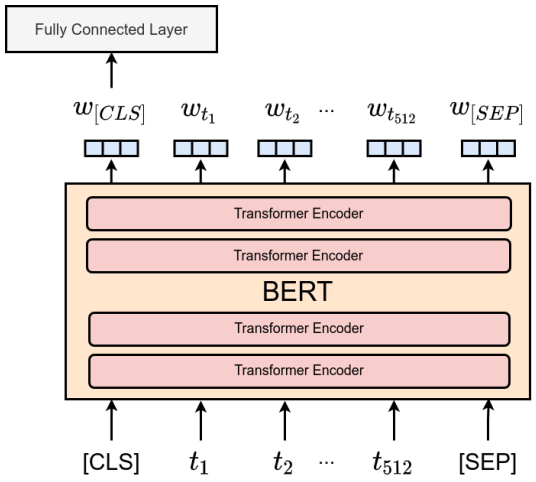


Figure 61: Sentence Classification

If we are dealing with multiple sentence classification, each sentence can be seperated with `[SEP]`, then passed to BERT: $([CLS], t_1, t_2, ..., t_n, [SEP], t_1, t_2, ..., t_n, [SEP])$.

For sequence labeling, each contextualized representation is passed to a fully connected layer. This allows us to classify each token:

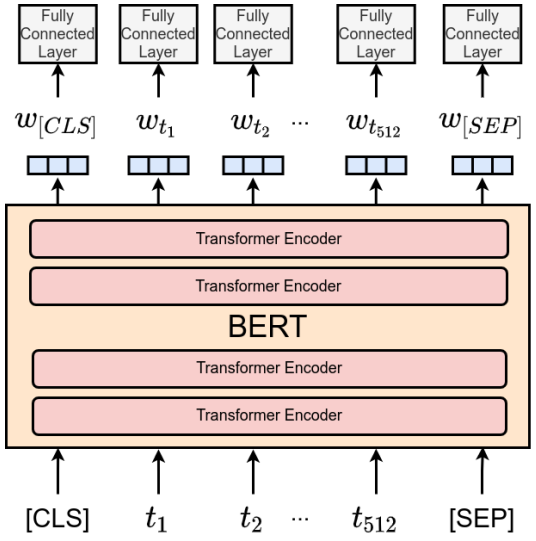


Figure 62: Sequence Labeling

For question answering, the question and the context is seperated by a `[SEP]` token:

$$([CLS], q_1, q_2, ..., q_n, [SEP], c_1, c_2, ..., c_n, [SEP])) \tag{158}$$

At the output, start and end span token representations are introduced and using those tokens, loss is calculated.

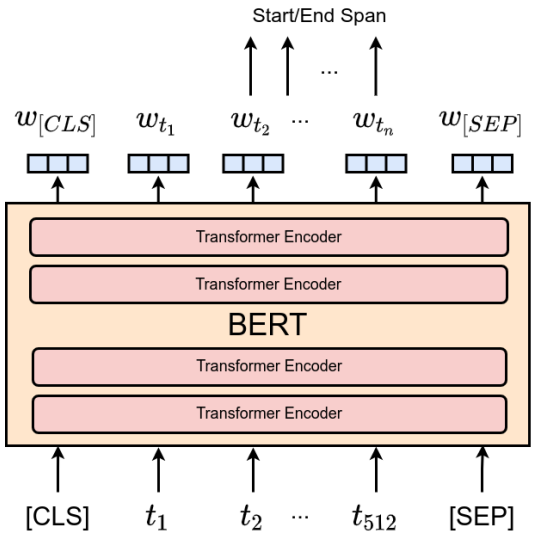


Figure 63: Question Answering

8.2 Improving Language Understanding by Generative Pre-Training

BERT model, that we explored in previous chapter, uses Transformer's encoder which is a bidirectional model. There are also different modeling of pretraining Transformers: autoregressive language modeling. As we have seen earlier, an autoregressive language model is formulated as

$$L(\mathcal{U}) = \sum_i \log p(u_i \mid u_{i-k}, ..., u_{i-1}) \tag{159}$$

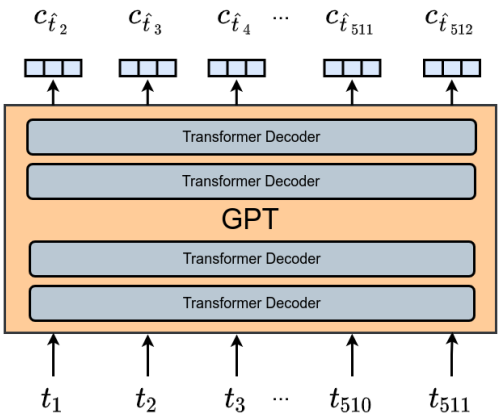


Figure 64: GPT model.

where k is the size of the context window. Authors of [50] proposed a new pre-training architecture that includes only the decoder section of Transformer, which works causal. This model is named as GPT.

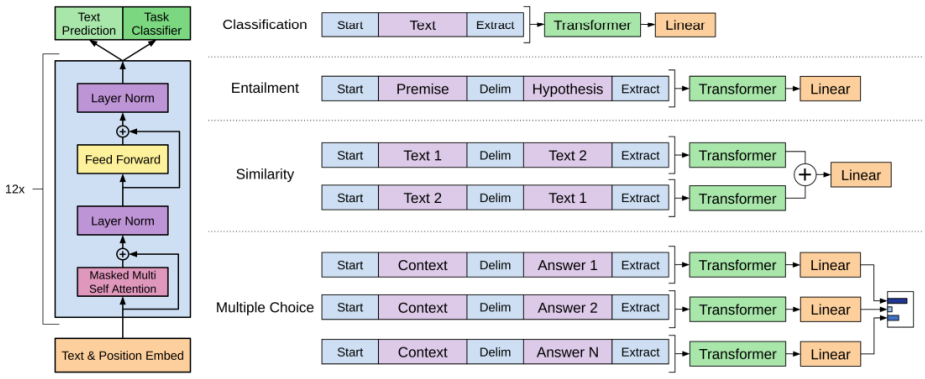


Figure 65: Finetuning approaches of GPT [50].

Finetuning approach of GPT is not much different from BERT. The input sequence $(t_1, t_2, ..., t_N)$ is passed to pre-trained GPT model, then, the last stack of decoder's last layer h_l^m is fed into fully connected layer:

$$C^* = \arg\max_C \ p(C \mid t_1, t_2, ..., t_N) = \arg\max_C \ softmax(h_l^m \cdot W_C) \tag{160}$$

Bidirectional language modeling and autoregressive language modeling have no advantage over each other. Both approaches have good and bad sides. For example, BERT has pretraining-finetuning discrepancy, in other words, `[MASK]` never occurs at finetuning but pre-training. Autoregressive language modeling does not need `[MASK]` noise due to its trivial to formulate objective. However, there is no bidirectional information flow. The representation $h_{x_{1:t-1}}$ only conditioned on the tokens up to position t.

8.3 ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators

ELECTRA is an alternative pretraining model to BERT. Instead of using Masked Language Modeling objective, ELECTRA uses Replaced Token Detection objective. This can be done by a small "generator", which replaces some tokens with plausible alternatives of input sentence. After that, "discriminator" tries to predict whether a token is replaced or not.

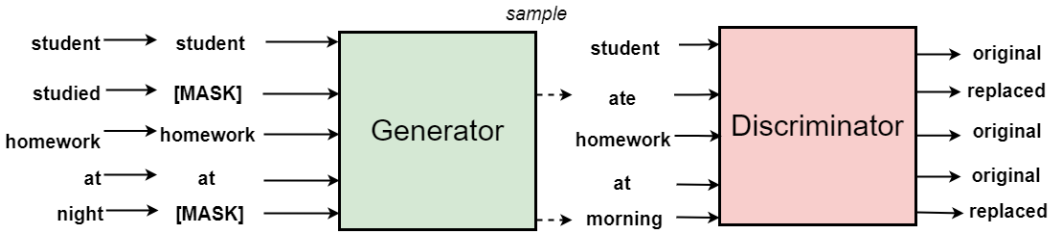


Figure 66: ELECTRA model

To make a more formal definition, input $\mathbf{x}$ is masked as in BERT model before feeding it to the generator

$$(m_i \sim U(1, N)) \in \mathbf{x}^{\text{masked}} \tag{161}$$

Then the masked token predictions are sampled then replaced by positions in the input $\mathbf{x}$

$$\hat{x}_i = p_G(x_i \mid \mathbf{x}^{\text{masked}}) \tag{162}$$

$$\mathbf{x}^{\text{corrupt}} = REPLACE(\mathbf{x}, \mathbf{m}, \hat{\mathbf{x}}) \tag{163}$$

ELECTRA is not trained adversarially as in GANs. Due to its sampling between the generator and the discriminator, it is not possible to backpropagate from end to start. Hence, the generator is simply trained with maximum likelihood and loss of the discriminator and the generator is combined over all

$$\mathcal{L}_{MLM} = \mathbb{E}\left[\sum_{i \in \mathbf{x}} -\log(p_G(x_i \mid \mathbf{x}^{\text{masked}}))\right] \tag{164}$$

$$\mathcal{L}_{Disc} = \mathbb{E}\left[\sum_{t} -\mathbb{1}(x_t^{\text{corrupt}} = x^t)\log D(\mathbf{x}^{\text{corrupt}}, t) - \mathbb{1}(x_t^{\text{corrupt}} \neq x^t)\log(1 - D(\mathbf{x}^{\text{corrupt}}, t))\right] \tag{165}$$

$$\nabla_{\theta_G, \theta_D}\mathcal{L}_{Total} = \min_{\theta_G, \theta_D} \sum_{\mathbf{x} \in X} \mathcal{L}_{MLM} + \lambda \mathcal{L}_{Disc} \tag{166}$$

8.4 XLNet: Generalized autoregressive pretraining for language understanding

8.5 BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension

9 Efficient Transformers

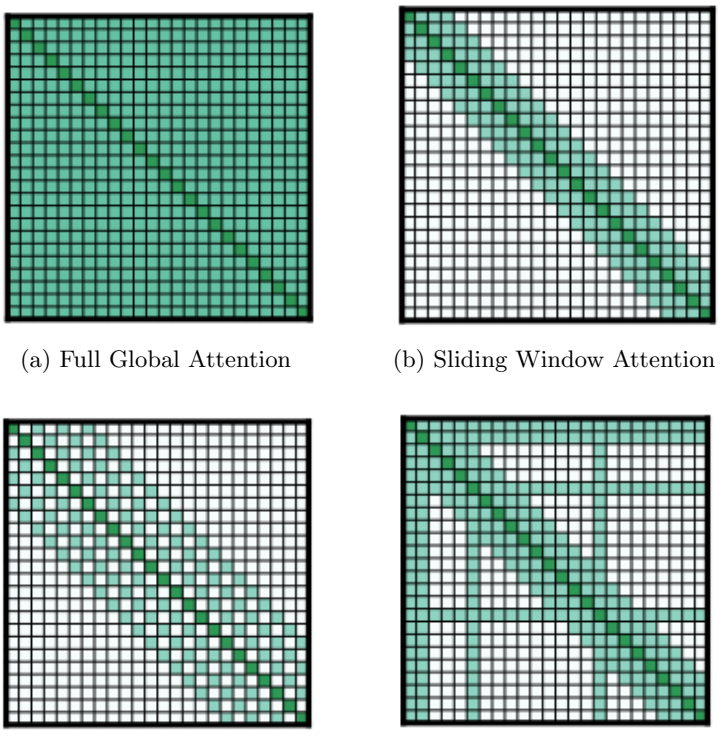
Self-attention is expensive. The memory and computational complexity is proportional to square of sequence length N. Let's bring more detailed complexity analysis.

Assume that our sequence length is N and d is the representation dimension (dimension for k, q, v). Time complexity per layer becomes $\mathcal{O}(d \cdot N^2)$.

9.1 Longformer

Longformer is proposed in [51]. Longformer is an approximation of full global attention. Longformer uses a combination of Sliding Window Attention and pre-defined Approximated Global Attention which is selected by considered task.

A **Sliding Window Attention** with window size of w computes the self-attention with $\mathcal{O}(n \times w)$. In other words, each token t_i attends to $\frac{1}{2}w$ tokens at left and right. Authors proposed that, it is efficient to use different values of w, at layer l_i.



(a) Full Global Attention (b) Sliding Window Attention

(c) Dilated Sliding Attention (d) Global + Sliding Attention

Figure 67: Different Attention Patterns

A **Dilated Sliding Window** method is inspired from dilated convolutions. Each token attends to other tokens in sliding window with dilation factor d. Authors proposed that, it is efficient to use different values of d, at layer l_i. With larger values of d, the attention mechanism focuses on longer context.

The **Approximated Global Attention** is done in task specific way. Besides the dilated slid-

ing window and sliding window attention, an approximate global attention is added by selecting random positions, in symmetric way. For example, if the task is sequence classification, global attention is used for the `[CLS]` token while in QA global attention is provided on all question tokens. If modeling longformer is in autoregressive fashion, dilated sliding window attention is used.

In Longformer, a staged training prodecure is used. The attention window size and sequence length is increased across multiple training phases (while halving the learning rate). Longformer is trained over 5 total phases with starting sequence length of 2048 and ending sequence length of 23040 on the last phase.

Pre-training Longformer is possible. It can be pre-trained with Masked Language Modeling (MLM) objective. Pre-training is expensive, on account of that, authors continue pre-training from the RoBERTa released checkpoint.

9.2 Transformer XL

9.3 Reformer

9.4 Shortformer

References

[1] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. 1st. USA: Prentice Hall PTR, 2000. ISBN: 0130950696.

[2] Martin Joos. "Description of Language Design". In: *The Journal of the Acoustical Society of America* 22.6 (Nov. 1950), pp. 701–707. DOI: 10.1121/1.1906674. URL: https://doi.org/10.1121/1.1906674.

[3] Zellig S. Harris. "Distributional Structure". In: *&lti&gtWORD&lt/i&gt* 10.2-3 (Aug. 1954), pp. 146–162. DOI: 10.1080/00437956.1954.11659520. URL: https://doi.org/10.1080/00437956.1954.11659520.

[4] J. R. Firth. "Applications of General Linguistics". In: *Transactions of the Philological Society* 56.1 (Nov. 1957), pp. 1–14. DOI: 10.1111/j.1467-968x.1957.tb00568.x. URL: https://doi.org/10.1111/j.1467-968x.1957.tb00568.x.

[5] Richard Socher et al. *Zero-Shot Learning Through Cross-Modal Transfer*. 2013. DOI: 10.48550/ARXIV.1301.3666. URL: https://arxiv.org/abs/1301.3666.

[6] Joseph Turian, Lev-Arie Ratinov, and Yoshua Bengio. "Word Representations: A Simple and General Method for Semi-Supervised Learning". In: *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Uppsala, Sweden: Association for Computational Linguistics, July 2010, pp. 384–394. URL: https://aclanthology.org/P10-1040.

[7] Ronan Collobert et al. *Natural Language Processing (almost) from Scratch*. 2011. DOI: 10.48550/ARXIV.1103.0398. URL: https://arxiv.org/abs/1103.0398.

[8] Will Y. Zou et al. "Bilingual Word Embeddings for Phrase-Based Machine Translation". In: *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, Washington, USA: Association for Computational Linguistics, Oct. 2013, pp. 1393–1398. URL: https://aclanthology.org/D13-1141.

[9] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. "Linguistic Regularities in Continuous Space Word Representations". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Atlanta, Georgia: Association for Computational Linguistics, June 2013, pp. 746–751. URL: https://aclanthology.org/N13-1090.

[10] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. DOI: 10.48550/ARXIV.1301.3781. URL: https://arxiv.org/abs/1301.3781.

[11] Carl Allen and Timothy Hospedales. "Analogies Explained: Towards Understanding Word Embeddings". In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, Sept. 2019, pp. 223–231. URL: https://proceedings.mlr.press/v97/allen19a.html.

[12] Felix Hill, Roi Reichart, and Anna Korhonen. "SimLex-999: Evaluating Semantic Models With (Genuine) Similarity Estimation". In: *Computational Linguistics* 41.4 (Dec. 2015), pp. 665–695. DOI: 10.1162/COLI_a_00237. URL: https://aclanthology.org/J15-4004.

[13] Yoshua Bengio et al. "A Neural Probabilistic Language Model". In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155. ISSN: 1532-4435.

[14] Tomas Mikolov et al. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. DOI: 10.48550/ARXIV.1310.4546. URL: https://arxiv.org/abs/1310.4546.

[15] Michael Gutmann and Aapo Hyvärinen. "Noise-contrastive estimation: A new estimation principle for unnormalized statistical models". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. Ed. by Yee Whye Teh and Mike Titterington. Vol. 9. Proceedings of Machine Learning Research. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 297–304. URL: https://proceedings.mlr.press/v9/gutmann10a.html.

[16] Jeffrey Pennington, Richard Socher, and Christopher Manning. "GloVe: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162. URL: https://aclanthology.org/D14-1162.

[17] Victoria Fromkin and Edward P. Stabler. *Linguistics: An Introduction to Linguistic Theory*. WB, 2001. URL: https://www.amazon.co.uk/Linguistics-Introduction-Linguistic-Victoria-Hayes/dp/0631197117.

[18] Piotr Bojanowski et al. *Enriching Word Vectors with Subword Information*. 2016. DOI: 10.48550/ARXIV.1607.04606. URL: https://arxiv.org/abs/1607.04606.

[19] Junhua Mao et al. "Deep Captioning with Multimodal Recurrent Neural Networks (m-RNN)". In: (2014). DOI: 10.48550/ARXIV.1412.6632. URL: https://arxiv.org/abs/1412.6632.

[20] Jeff Donahue et al. *DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition*. 2013. DOI: 10.48550/ARXIV.1310.1531. URL: https://arxiv.org/abs/1310.1531.

[21] Sébastien Jean et al. *On Using Very Large Target Vocabulary for Neural Machine Translation*. 2014. DOI: 10.48550/ARXIV.1412.2007. URL: https://arxiv.org/abs/1412.2007.

[22] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2014. DOI: 10.48550/ARXIV.1409.0473. URL: https://arxiv.org/abs/1409.0473.

[23] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. *Effective Approaches to Attention-based Neural Machine Translation*. 2015. DOI: 10.48550/ARXIV.1508.04025. URL: https://arxiv.org/abs/1508.04025.

[24] Volodymyr Mnih et al. *Recurrent Models of Visual Attention*. 2014. DOI: 10.48550/ARXIV.1406.6247. URL: https://arxiv.org/abs/1406.6247.

[25] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. *Multiple Object Recognition with Visual Attention*. 2014. DOI: 10.48550/ARXIV.1412.7755. URL: https://arxiv.org/abs/1412.7755.

[26] Kelvin Xu et al. *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention*. 2015. DOI: 10.48550/ARXIV.1502.03044. URL: https://arxiv.org/abs/1502.03044.

[27] Andrej Karpathy and Li Fei-Fei. *Deep Visual-Semantic Alignments for Generating Image Descriptions*. 2014. DOI: 10.48550/ARXIV.1412.2306. URL: https://arxiv.org/abs/1412.2306.

[28] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[29] Alexey Dosovitskiy et al. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2020. DOI: 10.48550/ARXIV.2010.11929. URL: https://arxiv.org/abs/2010.11929.

[30] Wei Liu et al. *CPTR: Full Transformer Network for Image Captioning*. 2021. DOI: 10.48550/ARXIV.2101.10804. URL: https://arxiv.org/abs/2101.10804.

[31] Wonjae Kim, Bokyung Son, and Ildoo Kim. *ViLT: Vision-and-Language Transformer Without Convolution or Region Supervision*. 2021. DOI: 10.48550/ARXIV.2102.03334. URL: https://arxiv.org/abs/2102.03334.

[32] Alec Radford et al. *Learning Transferable Visual Models From Natural Language Supervision*. 2021. DOI: 10.48550/ARXIV.2103.00020. URL: https://arxiv.org/abs/2103.00020.

[33] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[34] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey*. 2018. arXiv: 1502.05767 [cs.SC].

[35] Matthew E. Peters et al. *Deep contextualized word representations*. 2018. arXiv: 1802.05365 [cs.CL].

[36] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer International Publishing, 2018.

[37] William Merrill. *Formal Language Theory Meets Modern NLP*. 2021. arXiv: 2102.10094 [cs.CL].

[38] Graham Neubig et al. *DyNet: The Dynamic Neural Network Toolkit*. 2017. arXiv: 1701.03980 [stat.ML].

[39] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. *On the difficulty of training Recurrent Neural Networks*. 2013. arXiv: 1211.5063 [cs.LG].

[40] Christopher Olah. *Understanding LSTM Networks*. 2015. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

[41] Sepp Hochreiter and Jürgen Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[42] Kyunghyun Cho et al. *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. 2014. arXiv: 1406.1078 [cs.CL].

[43] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. 2016. arXiv: 1409.0473 [cs.CL].

[44] Sanjeev Arora et al. *Linear Algebraic Structure of Word Senses, with Applications to Polysemy*. 2018. arXiv: 1601.03764 [cs.CL].

[45] Matthew E. Peters et al. *Semi-supervised sequence tagging with bidirectional language models*. 2017. arXiv: 1705.00108 [cs.CL].

[46] Bryan McCann et al. *Learned in Translation: Contextualized Word Vectors*. 2018. arXiv: 1708.00107 [cs.CL].

[47] A. Fromkin Victoria. *Linguistics: An Introduction to Linguistic Theory*. 2011. ISBN: 0631197117.

[48] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[49] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[50] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018.

[51] Iz Beltagy, Matthew E. Peters, and Arman Cohan. *Longformer: The Long-Document Transformer*. 2020. arXiv: 2004.05150 [cs.CL].