# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part I*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Francois Chaubard, Michael Fang, Guillaume Genthial, Rohit Mundra, Richard Socher

**Keyphrases: Natural Language Processing. Word Vectors. Singular Value Decomposition. Skip-gram. Continuous Bag of Words (CBOW). Negative Sampling. Hierarchical Softmax. Word2Vec.**

This set of notes begins by introducing the concept of Natural Language Processing (NLP) and the problems NLP faces today. We then move forward to discuss the concept of representing words as numeric vectors. Lastly, we discuss popular approaches to designing word vectors.

## 1 Introduction to Natural Language Processing

We begin with a general discussion of what is NLP.

### 1.1 What is so special about NLP?

What's so special about human (natural) language? Human language is a system specifically constructed to convey meaning, and is not produced by a physical manifestation of any kind. In that way, it is very different from vision or any other machine learning task.

Natural language is a discrete/symbolic/categorical system

Most words are just symbols for an extra-linguistic entity : the word is a *signifier* that maps to a *signified* (idea or thing).

For instance, the word "rocket" refers to the concept of a rocket, and by extension can designate an instance of a rocket. There are some exceptions, when we use words and letters for expressive signaling, like in "Whooompaa". On top of this, the symbols of language can be encoded in several modalities : voice, gesture, writing, etc that are transmitted via *continuous* signals to the brain, which itself appears to encode things in a continuous manner. (A lot of work in philosophy of language and linguistics has been done to conceptualize human language and distinguish words from their references, meanings, etc. Among others, see works by Wittgenstein, Frege, Russell and Mill.)

### 1.2 Examples of tasks

There are different levels of tasks in NLP, from speech processing to semantic interpretation and discourse processing. The goal of NLP is to be able to design algorithms to allow computers to "understand"

natural language in order to perform some task. Example tasks come in varying level of difficulty:

**Easy**

- Spell Checking
- Keyword Search
- Finding Synonyms

**Medium**

- Parsing information from websites, documents, etc.

**Hard**

- Machine Translation (e.g. Translate Chinese text to English)
- Semantic Analysis (What is the meaning of query statement?)
- Coreference (e.g. What does "he" or "it" refer to given a document?)
- Question Answering (e.g. Answering Jeopardy questions).

### *1.3 How to represent words?*

The first and arguably most important common denominator across all NLP tasks is how we represent words as input to any of our models. Much of the earlier NLP work that we will not cover treats words as atomic symbols. To perform well on most NLP tasks we first need to have some notion of similarity and difference between words. With word vectors, we can quite easily encode this ability in the vectors themselves (using distance measures such as Jaccard, Cosine, Euclidean, etc).

## *2 Word Vectors*

There are an estimated 13 million tokens for the English language but are they all completely unrelated? Feline to cat, hotel to motel? I think not. Thus, we want to encode word tokens each into some vector that represents a point in some sort of "word" space. This is paramount for a number of reasons but the most intuitive reason is that perhaps there actually exists some $N$-dimensional space (such that $N \ll 13$ million) that is sufficient to encode all semantics of our language. Each dimension would encode some meaning that we transfer using speech. For instance, semantic dimensions might

indicate tense (past vs. present vs. future), count (singular vs. plural), and gender (masculine vs. feminine).

So let's dive into our first word vector and arguably the most simple, the **one-hot vector**: Represent every word as an $\mathbb{R}^{|V|\times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language. In this notation, $|V|$ is the size of our vocabulary. Word vectors in this type of encoding would appear as the following:

**One-hot** vector: Represent every word as an $\mathbb{R}^{|V|\times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language.

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{a} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

**Fun fact:** The term "one-hot" comes from digital circuit design, meaning "a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0)".

We represent each word as a completely independent entity. As we previously discussed, this word representation does not give us directly any notion of similarity. For instance,

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

So maybe we can try to reduce the size of this space from $\mathbb{R}^{|V|}$ to something smaller and thus find a subspace that encodes the relationships between words.

## 3 *SVD Based Methods*

For this class of methods to find word embeddings (otherwise known as word vectors), we first loop over a massive dataset and accumulate word co-occurrence counts in some form of a matrix $X$, and then perform Singular Value Decomposition on $X$ to get a $USV^T$ decomposition. We then use the rows of $U$ as the word embeddings for all words in our dictionary. Let us discuss a few choices of $X$.

### 3.1 *Word-Document Matrix*

As our first attempt, we make the bold conjecture that words that are related will often appear in the same documents. For instance, "banks", "bonds", "stocks", "money", etc. are probably likely to appear together. But "banks", "octopus", "banana", and "hockey" would probably not consistently appear together. We use this fact to build a word-document matrix, $X$ in the following manner: Loop over billions of documents and for each time word $i$ appears in document $j$, we add one to entry $X_{ij}$. This is obviously a very large matrix ($\mathbb{R}^{|V|\times M}$) and it scales with the number of documents ($M$). So perhaps we can try something better.

### 3.2 *Window based Co-occurrence Matrix*

The same kind of logic applies here however, the matrix $X$ stores co-occurrences of words thereby becoming an affinity matrix. In this method we count the number of times each word appears inside a window of a particular size around the word of interest. We calculate this count for all the words in corpus. We display an example below. Let our corpus contain just three sentences and the window size be 1:

1. I enjoy flying.
2. I like NLP.
3. I like deep learning.

The resulting counts matrix will then be:

$$X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \begin{bmatrix} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array}$$

### 3.3 *Applying SVD to the cooccurrence matrix*

We now perform SVD on $X$, observe the singular values (the diagonal entries in the resulting $S$ matrix), and cut them off at some index $k$ based on the desired percentage variance captured:

$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$$

We then take the submatrix of $U_{1:|V|,1:k}$ to be our word embedding matrix. This would thus give us a $k$-dimensional representation of every word in the vocabulary.

**Using Word-Word Co-occurrence Matrix:**

- Generate $|V| \times |V|$ co-occurrence matrix, $X$.
- Apply SVD on $X$ to get $X = USV^T$.
- Select the first $k$ columns of $U$ to get a $k$-dimensional word vectors.
- $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$ indicates the amount of variance captured by the first $k$ dimensions.

**Applying SVD to $X$:**

$$\begin{array}{c} \\ |V| \end{array} \begin{array}{c} \begin{array}{c} |V| \end{array} \\ \begin{bmatrix} \\ X \\ \\ \end{bmatrix} \end{array} = \begin{array}{c} \\ |V| \end{array} \begin{array}{c} \begin{array}{c} |V| \end{array} \\ \begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix} \end{array} \begin{array}{c} \\ |V| \end{array} \begin{array}{c} \begin{array}{c} |V| \end{array} \\ \begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \end{array} \begin{array}{c} \\ |V| \end{array} \begin{array}{c} \begin{array}{c} |V| \end{array} \\ \begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix} \end{array}$$

**Reducing dimensionality by selecting first $k$ singular vectors**:

$$\begin{matrix} & |V| & & k & & k & & |V| & \\ |V| & \begin{bmatrix} \\ \hat{X} \\ \\ \end{bmatrix} & = |V| & \begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix} & k & \begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} & k & \begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix} \end{matrix}$$

Both of these methods give us word vectors that are more than sufficient to encode semantic and syntactic (part of speech) information but are associated with many other problems:

- The dimensions of the matrix change very often (new words are added very frequently and corpus changes in size).
- The matrix is extremely sparse since most words do not co-occur.
- The matrix is very high dimensional in general ($\approx 10^6 \times 10^6$)
- Quadratic cost to train (i.e. to perform SVD)
- Requires the incorporation of some hacks on $X$ to account for the drastic imbalance in word frequency

SVD based methods do not scale well for big matrices and it is hard to incorporate new words or documents. Computational cost for a $m \times n$ matrix is $O(mn^2)$

Some solutions to exist to resolve some of the issues discussed above:

- Ignore function words such as "the", "he", "has", etc.
- Apply a ramp window – i.e. weight the co-occurrence count based on distance between the words in the document.
- Use Pearson correlation and set negative counts to 0 instead of using just raw count.

However, count-based method make an efficient use of the statistics

As we see in the next section, iteration based methods solve many of these issues in a far more elegant manner.

## 4 *Iteration Based Methods - Word2vec*

Let us step back and try a new approach. Instead of computing and storing global information about some huge dataset (which might be billions of sentences), we can try to create a model that will be able to learn one iteration at a time and eventually be able to encode the probability of a word given its context.

**For an overview of Word2vec**, a note map can be found here : `https://myndbook.com/view/4900`

A detailed summary of word2vec models can also be found here [Rong, 2014]

The idea is to design a model whose parameters are the word vectors. Then, train the model on a certain objective. At every iteration we run our model, evaluate the errors, and follow an update rule that has some notion of penalizing the model parameters that caused the error. Thus, we learn our word vectors. This idea is a very old

Iteration-based methods capture cooccurrence of words one at a time instead of capturing all cooccurrence counts directly like in SVD methods.

one dating back to 1986. We call this method "backpropagating" the errors (see [Rumelhart et al., 1988]). The simpler the model and the task, the faster it will be to train it.

Several approaches have been tested. [Collobert et al., 2011] design models for NLP whose first step is to transform each word in a vector. For each special task (Named Entity Recognition, Part-of-Speech tagging, etc. ) they train not only the model's parameters but also the vectors and achieve great performance, while computing good word vectors! Other interesting reading would be [Bengio et al., 2003].

In this class, we will present a simpler, more recent, probabilistic method by [Mikolov et al., 2013] : word2vec. Word2vec is a software package that actually includes :

- **2 algorithms**: continuous bag-of-words (CBOW) and skip-gram. CBOW aims to predict a center word from the surrounding context in terms of word vectors. Skip-gram does the opposite, and predicts the *distribution* (probability) of context words from a center word.

- **2 training methods**: negative sampling and hierarchical softmax. Negative sampling defines an objective by sampling *negative* examples, while hierarchical softmax defines an objective using an efficient tree structure to compute probabilities for all the vocabulary.

**Context of a word:**
The context of a word is the set of $m$ surrounding words. For instance, the $m = 2$ context of the word "fox" in the sentence "The quick brown fox jumped over the lazy dog" is {"quick", "brown", "jumped", "over"}.

This model relies on a very important hypothesis in linguistics, *distributional similarity*, the idea that similar words have similar context.

### *4.1 Language Models (Unigrams, Bigrams, etc.)*

First, we need to create such a model that will assign a probability to a sequence of tokens. Let us start with an example:

*"The cat jumped over the puddle."*

A good language model will give this sentence a high probability because this is a completely valid sentence, syntactically and semantically. Similarly, the sentence "stock boil fish is toy" should have a very low probability because it makes no sense. Mathematically, we can call this probability on any given sequence of $n$ words:

$$P(w_1, w_2, \cdots, w_n)$$

We can take the unary language model approach and break apart this probability by assuming the word occurrences are completely independent:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i)$$

**Unigram model:**

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=1}^{n} P(w_i)$$

However, we know this is a bit ludicrous because we know the next word is highly contingent upon the previous sequence of words. And the silly sentence example might actually score highly. So perhaps we let the probability of the sequence depend on the pairwise

probability of a word in the sequence and the word next to it. We call this the bigram model and represent it as:

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=2}^{n} P(w_i|w_{i-1})$$

Again this is certainly a bit naive since we are only concerning ourselves with pairs of neighboring words rather than evaluating a whole sentence, but as we will see, this representation gets us pretty far along. Note in the Word-Word Matrix with a context of size 1, we basically can learn these pairwise probabilities. But again, this would require computing and storing global information about a massive dataset.

Now that we understand how we can think about a sequence of tokens having a probability, let us observe some example models that could learn these probabilities.

**Bigram model:**

$$P(w_1, w_2, \cdots, w_n) = \prod_{i=2}^{n} P(w_i|w_{i-1})$$

### 4.2 *Continuous Bag of Words Model (CBOW)*

One approach is to treat {"The", "cat", 'over", "the', "puddle"} as a context and from these words, be able to predict or generate the center word "jumped". This type of model we call a Continuous Bag of Words (CBOW) Model.

Let's discuss the CBOW Model above in greater detail. First, we set up our known parameters. Let the known parameters in our model be the sentence represented by one-hot word vectors. The input one hot vectors or context we will represent with an $x^{(c)}$. And the output as $y^{(c)}$ and in the CBOW model, since we only have one output, so we just call this $y$ which is the one hot vector of the known center word. Now let's define our unknowns in our model.

We create two matrices, $\mathcal{V} \in \mathbb{R}^{n \times |V|}$ and $\mathcal{U} \in \mathbb{R}^{|V| \times n}$. Where $n$ is an arbitrary size which defines the size of our embedding space. $\mathcal{V}$ is the input word matrix such that the $i$-th column of $\mathcal{V}$ is the $n$-dimensional embedded vector for word $w_i$ when it is an input to this model. We denote this $n \times 1$ vector as $v_i$. Similarly, $\mathcal{U}$ is the output word matrix. The $j$-th row of $\mathcal{U}$ is an $n$-dimensional embedded vector for word $w_j$ when it is an output of the model. We denote this row of $\mathcal{U}$ as $u_j$. Note that we do in fact learn two vectors for every word $w_i$ (i.e. input word vector $v_i$ and output word vector $u_i$).

**CBOW Model:**
Predicting a center word from the surrounding context

For each word, we want to learn 2 vectors
- $v$: (input vector) when the word is in the context
- $u$: (output vector) when the word is in the center

**Notation for CBOW Model:**

- $w_i$: Word $i$ from vocabulary $V$
- $\mathcal{V} \in \mathbb{R}^{n \times |V|}$: Input word matrix
- $v_i$: $i$-th column of $\mathcal{V}$, the input vector representation of word $w_i$
- $\mathcal{U} \in \mathbb{R}^{|V| \times n}$: Output word matrix
- $u_i$: $i$-th row of $\mathcal{U}$, the output vector representation of word $w_i$

We breakdown the way this model works in these steps:

1. We generate our one hot word vectors for the input context of size $m$ : $(x^{(c-m)}, \ldots, x^{(c-1)}, x^{(c+1)}, \ldots, x^{(c+m)} \in \mathbb{R}^{|V|})$.

2. We get our embedded word vectors for the context ($v_{c-m} = \mathcal{V}x^{(c-m)}, v_{c-m+1} = \mathcal{V}x^{(c-m+1)}, \ldots, v_{c+m} = \mathcal{V}x^{(c+m)} \in \mathbb{R}^n$)
3. Average these vectors to get $\hat{v} = \frac{v_{c-m}+v_{c-m+1}+\ldots+v_{c+m}}{2m} \in \mathbb{R}^n$
4. Generate a score vector $z = \mathcal{U}\hat{v} \in \mathbb{R}^{|V|}$. As the dot product of similar vectors is higher, it will push similar words close to each other in order to achieve a high score.
5. Turn the scores into probabilities $\hat{y} = \text{softmax}(z) \in \mathbb{R}^{|V|}$.
6. We desire our probabilities generated, $\hat{y} \in \mathbb{R}^{|V|}$, to match the true probabilities, $y \in \mathbb{R}^{|V|}$, which also happens to be the one hot vector of the actual word.

The **softmax** is an operator that we'll use very frequently. It transforms a vector into a vector whose $i$-th component is $\frac{e^{\hat{y}_i}}{\sum_{k=1}^{|V|} e^{\hat{y}_k}}$.
- exponentiate to make positive
- Dividing by $\sum_{k=1}^{|V|} e^{\hat{y}_k}$ *normalizes* the vector ($\sum_{k=1}^{n} \hat{y}_k = 1$) to give probability

So now that we have an understanding of how our model would work if we had a $\mathcal{V}$ and $\mathcal{U}$, how would we learn these two matrices? Well, we need to create an objective function. Very often when we are trying to learn a probability from some true probability, we look to information theory to give us a measure of the distance between two distributions. Here, we use a popular choice of distance/loss measure, cross entropy $H(\hat{y}, y)$.

The intuition for the use of cross-entropy in the discrete case can be derived from the formulation of the loss function:

$$H(\hat{y}, y) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j)$$

Let us concern ourselves with the case at hand, which is that $y$ is a one-hot vector. Thus we know that the above loss simplifies to simply:

$$H(\hat{y}, y) = -y_i \log(\hat{y}_i)$$

In this formulation, $c$ is the index where the correct word's one hot vector is 1. We can now consider the case where our prediction was perfect and thus $\hat{y}_c = 1$. We can then calculate $H(\hat{y}, y) = -1 \log(1) = 0$. Thus, for a perfect prediction, we face no penalty or loss. Now let us consider the opposite case where our prediction was very bad and thus $\hat{y}_c = 0.01$. As before, we can calculate our loss to be $H(\hat{y}, y) = -1 \log(0.01) \approx 4.605$. We can thus see that for probability distributions, cross entropy provides us with a good measure of distance. We thus formulate our optimization objective as:

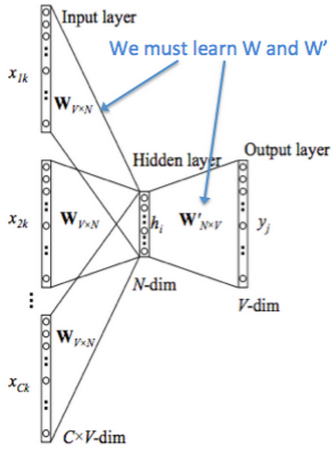



Figure 1: This image demonstrates how CBOW works and how we must learn the transfer matrices

$\hat{y} \mapsto H(\hat{y}, y)$ is minimum when $\hat{y} = y$. Then, if we found a $\hat{y}$ such that $H(\hat{y}, y)$ is close to the minimum, we have $\hat{y} \approx y$. This means that our model is very good at predicting the center word!

To learn the vectors (the matrices $U$ and $V$) CBOW defines a cost that measures how good it is at predicting the center word. Then, we optimize this cost by updating the matrices $U$ and $V$ thanks to stochastic gradient descent

$$\begin{aligned}\text{minimize } J &= -\log P(w_c|w_{c-m},\ldots,w_{c-1},w_{c+1},\ldots,w_{c+m}) \\ &= -\log P(u_c|\hat{v}) \\ &= -\log \frac{\exp(u_c^T\hat{v})}{\sum_{j=1}^{|V|}\exp(u_j^T\hat{v})} \\ &= -u_c^T\hat{v} + \log\sum_{j=1}^{|V|}\exp(u_j^T\hat{v})\end{aligned}$$

We use stochastic gradient descent to update all relevant word vectors $u_c$ and $v_j$.

Stochastic gradient descent (SGD) computes gradients for a window and updates the parameters
$\mathcal{U}_{new} \leftarrow \mathcal{U}_{old} - \alpha\nabla_{\mathcal{U}}J$
$\mathcal{V}_{old} \leftarrow \mathcal{V}_{old} - \alpha\nabla_{\mathcal{V}}J$

**Skip-Gram Model:**
Predicting surrounding context words given a center word

### 4.3 *Skip-Gram Model*

Another approach is to create a model such that given the center word "jumped", the model will be able to predict or generate the surrounding words "The", "cat", "over", "the", "puddle". Here we call the word "jumped" the context. We call this type of model a Skip-Gram model.

Let's discuss the Skip-Gram model above. The setup is largely the same but we essentially swap our $x$ and $y$ i.e. $x$ in the CBOW are now $y$ and vice-versa. The input one hot vector (center word) we will represent with an $x$ (since there is only one). And the output vectors as $y^{(j)}$. We define $\mathcal{V}$ and $\mathcal{U}$ the same as in CBOW.

**Notation for Skip-Gram Model:**

- $w_i$: Word $i$ from vocabulary $V$
- $\mathcal{V} \in \mathbb{R}^{n\times|V|}$: Input word matrix
- $v_i$: $i$-th column of $\mathcal{V}$, the input vector representation of word $w_i$
- $\mathcal{U} \in \mathbb{R}^{n\times|V|}$: Output word matrix
- $u_i$: $i$-th row of $\mathcal{U}$, the output vector representation of word $w_i$

We breakdown the way this model works in these 6 steps:

1. We generate our one hot input vector $x \in \mathbb{R}^{|V|}$ of the center word.
2. We get our embedded word vector for the center word $v_c = \mathcal{V}x \in \mathbb{R}^n$
3. Generate a score vector $z = \mathcal{U}v_c$.
4. Turn the score vector into probabilities, $\hat{y} = \text{softmax}(z)$. Note that $\hat{y}_{c-m},\ldots,\hat{y}_{c-1},\hat{y}_{c+1},\ldots,\hat{y}_{c+m}$ are the probabilities of observing each context word.
5. We desire our probability vector generated to match the true probabilities which is $y^{(c-m)},\ldots,y^{(c-1)},y^{(c+1)},\ldots,y^{(c+m)}$, the one hot vectors of the actual output.

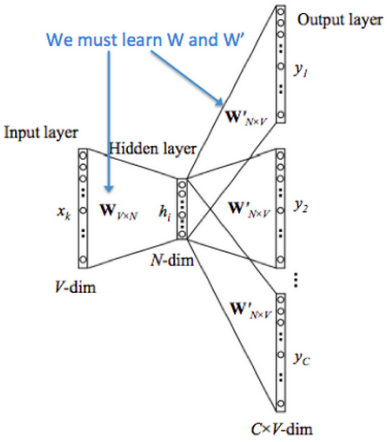



Figure 2: This image demonstrates how Skip-Gram works and how we must learn the transfer matrices

As in CBOW, we need to generate an objective function for us to evaluate the model. A key difference here is that we invoke a Naive Bayes assumption to break out the probabilities. If you have not seen this before, then simply put, it is a strong (naive) conditional

independence assumption. In other words, given the center word, all output words are completely independent.

$$\begin{aligned}
\text{minimize } J &= -\log P(w_{c-m},\ldots,w_{c-1},w_{c+1},\ldots,w_{c+m}|w_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} P(w_{c-m+j}|w_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} P(u_{c-m+j}|v_c) \\
&= -\log \prod_{j=0,j\neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|}\exp(u_k^T v_c)} \\
&= -\sum_{j=0,j\neq m}^{2m} u_{c-m+j}^T v_c + 2m\log\sum_{k=1}^{|V|}\exp(u_k^T v_c)
\end{aligned}$$

With this objective function, we can compute the gradients with respect to the unknown parameters and at each iteration update them via Stochastic Gradient Descent.

Only one probability vector $\hat{y}$ is computed. Skip-gram treats each context word equally : the models computes the probability for each word of appearing in the context independently of its distance to the center word

Note that

$$\begin{aligned}
J &= -\sum_{j=0,j\neq m}^{2m} \log P(u_{c-m+j}|v_c) \\
&= \sum_{j=0,j\neq m}^{2m} H(\hat{y}, y_{c-m+j})
\end{aligned}$$

where $H(\hat{y}, y_{c-m+j})$ is the cross-entropy between the probability vector $\hat{y}$ and the one-hot vector $y_{c-m+j}$.

### *4.4 Negative Sampling*

Loss functions $J$ for CBOW and Skip-Gram are expensive to compute because of the softmax normalization, where we sum over all $|V|$ scores!

Lets take a second to look at the objective function. Note that the summation over $|V|$ is computationally huge! Any update we do or evaluation of the objective function would take $O(|V|)$ time which if we recall is in the millions. A simple idea is we could instead just approximate it.

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. To augment our formulation of the problem to incorporate Negative Sampling, all we need to do is update the:

- objective function

- gradients
- update rules

Mikolov et al. present **Negative Sampling** in Distributed Representations of Words and Phrases and their Compositionality. While negative sampling is based on the Skip-Gram model, it is in fact optimizing a different objective. Consider a pair $(w, c)$ of word and context. Did this pair come from the training data? Let's denote by $P(D = 1|w, c)$ the probability that (w, c) came from the corpus data. Correspondingly, $P(D = 0|w, c)$ will be the probability that $(w, c)$ did not come from the corpus data. First, let's model $P(D = 1|w, c)$ with the sigmoid function:

$$P(D = 1|w, c, \theta) = \sigma(v_c^T v_w) = \frac{1}{1 + e^{(-v_c^T v_w)}}$$

The **sigmoid** function $\sigma(x) = \frac{1}{1+e^{-x}}$ is the 1D version of the softmax and can be used to model a probability

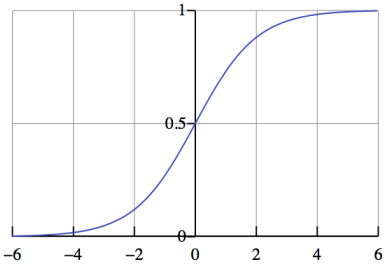


Figure 3: Sigmoid function

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. We take a simple maximum likelihood approach of these two probabilities. (Here we take $\theta$ to be the parameters of the model, and in our case it is $\mathcal{V}$ and $\mathcal{U}$.)

$$\begin{aligned}
\theta &= \underset{\theta}{\text{argmax}} \prod_{(w,c)\in D} P(D = 1|w, c, \theta) \prod_{(w,c)\in \tilde{D}} P(D = 0|w, c, \theta) \\
&= \underset{\theta}{\text{argmax}} \prod_{(w,c)\in D} P(D = 1|w, c, \theta) \prod_{(w,c)\in \tilde{D}} (1 - P(D = 1|w, c, \theta)) \\
&= \underset{\theta}{\text{argmax}} \sum_{(w,c)\in D} \log P(D = 1|w, c, \theta) + \sum_{(w,c)\in \tilde{D}} \log(1 - P(D = 1|w, c, \theta)) \\
&= \underset{\theta}{\text{argmax}} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} + \sum_{(w,c)\in \tilde{D}} \log(1 - \frac{1}{1 + \exp(-u_w^T v_c)}) \\
&= \underset{\theta}{\text{argmax}} \sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} + \sum_{(w,c)\in \tilde{D}} \log(\frac{1}{1 + \exp(u_w^T v_c)})
\end{aligned}$$

Note that maximizing the likelihood is the same as minimizing the negative log likelihood

$$J = -\sum_{(w,c)\in D} \log \frac{1}{1 + \exp(-u_w^T v_c)} - \sum_{(w,c)\in \tilde{D}} \log(\frac{1}{1 + \exp(u_w^T v_c)})$$

Note that $\tilde{D}$ is a "false" or "negative" corpus. Where we would have sentences like "stock boil fish is toy". Unnatural sentences that should get a low probability of ever occurring. We can generate $\tilde{D}$ on the fly by randomly sampling this negative from the word bank.

For skip-gram, our new objective function for observing the context word $c-m+j$ given the center word $c$ would be

$$-\log\sigma(u_{c-m+j}^T \cdot v_c) - \sum_{k=1}^{K} \log\sigma(-\tilde{u}_k^T \cdot v_c)$$

To compare with the regular softmax loss for skip-gram
$-u_{c-m+j}^T v_c + \log\sum_{k=1}^{|V|} \exp(u_k^T v_c)$

For CBOW, our new objective function for observing the center word $u_c$ given the context vector $\hat{v} = \frac{v_{c-m}+v_{c-m+1}+...+v_{c+m}}{2m}$ would be

$$-\log\sigma(u_c^T \cdot \hat{v}) - \sum_{k=1}^{K} \log\sigma(-\tilde{u}_k^T \cdot \hat{v})$$

To compare with the regular softmax loss for CBOW
$-u_c^T \hat{v} + \log\sum_{j=1}^{|V|} \exp(u_j^T \hat{v})$

In the above formulation, $\{\tilde{u}_k | k = 1 \dots K\}$ are sampled from $P_n(w)$. Let's discuss what $P_n(w)$ should be. While there is much discussion of what makes the best approximation, what seems to work best is the Unigram Model raised to the power of 3/4. Why 3/4? Here's an example that might help gain some intuition:

$$\text{is: } 0.9^{3/4} = 0.92$$
$$\text{Constitution: } 0.09^{3/4} = 0.16$$
$$\text{bombastic: } 0.01^{3/4} = 0.032$$

"Bombastic" is now 3x more likely to be sampled while "is" only went up marginally.

### 4.5 *Hierarchical Softmax*

Mikolov et al. also present hierarchical softmax as a much more efficient alternative to the normal softmax. In practice, hierarchical softmax tends to be better for infrequent words, while negative sampling works better for frequent words and lower dimensional vectors.

Hierarchical softmax uses a binary tree to represent all words in the vocabulary. Each leaf of the tree is a word, and there is a unique path from root to leaf. In this model, there is *no output representation for words*. Instead, each node of the graph (except the root and the leaves) is associated to a vector that the model is going to learn.

In this model, the probability of a word $w$ given a vector $w_i$, $P(w|w_i)$, is equal to the probability of a random walk starting in the root and ending in the leaf node corresponding to $w$. The main advantage in computing the probability this way is that the cost is only $O(\log(|V|))$, corresponding to the length of the path.

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, $L(w_2)$ in Figure 4 is 3. Let's write $n(w,i)$ as the $i$-th node on this path with associated

Hierarchical Softmax uses a binary tree where leaves are the words. The probability of a word being the output word is defined as the probability of a random walk from the root to that word's leaf. Computational cost becomes $O(\log(|V|))$ instead of O(|V|).

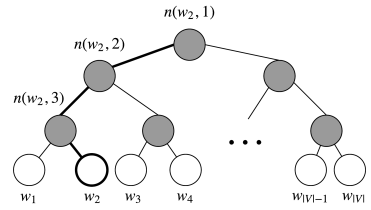



Figure 4: Binary tree for Hierarchical softmax

vector $v_{n(w,i)}$. So $n(w,1)$ is the root, while $n(w,L(w))$ is the father of $w$. 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

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

where

$$[x] = \begin{cases} 1 \text{ if } x \text{ is true} \\ -1 \text{ otherwise} \end{cases}$$

.

and $\sigma(\cdot)$ is the sigmoid function.

This formula is fairly dense, so let's examine it more closely.

First, we are computing a product of terms based on the shape of the path from the root ($n(w,1)$) to the 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 $v_n^T v_{w_i}$,

$$\sigma(v_n^T v_{w_i}) + \sigma(-v_n^T v_{w_i}) = 1$$

The normalization also ensures that $\sum_{w=1}^{|V|} P(w|w_i) = 1$, just as in the original softmax.

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. Let's run through an example. Taking $w_2$ in Figure 4, we must take two left edges and then a right edge to reach $w_2$ from the root, so

$$\begin{aligned} P(w_2|w_i) &= p(n(w_2,1), \text{left}) \cdot p(n(w_2,2), \text{left}) \cdot p(n(w_2,3), \text{right}) \\ &= \sigma(v_{n(w_2,1)}^T v_{w_i}) \cdot \sigma(v_{n(w_2,2)}^T v_{w_i}) \cdot \sigma(-v_{n(w_2,3)}^T v_{w_i}) \end{aligned}$$

To train the model, our goal is still to minimize the negative log likelihood $-\log P(w|w_i)$. But instead of updating output vectors per word, we update the vectors of the nodes in the binary tree that are in the path from root to leaf node.

The speed of this method is determined by the way in which the binary tree is constructed and words are assigned to leaf nodes. MIKOLOV ET AL. use a binary Huffman tree, which assigns frequent words shorter paths in the tree.

## *References*

[Bengio et al., 2003] Bengio, Y., Ducharme, R., Vincent, P., and Janvin, C. (2003). A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155.

[Collobert et al., 2011] Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. P. (2011). Natural language processing (almost) from scratch. *CoRR*, abs/1103.0398.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781.

[Rong, 2014] Rong, X. (2014). word2vec parameter learning explained. *CoRR*, abs/1411.2738.

[Rumelhart et al., 1988] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part II*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Author: Rohit Mundra, Emma Peng, Richard Socher, Ajay Sohmshetty

**Keyphrases: Global Vectors for Word Representation (GloVe). Intrinsic and extrinsic evaluations. Effect of hyperparameters on analogy evaluation tasks. Correlation of human judgment with word vector distances. Dealing with ambiguity in word using contexts. Window classification.**

This set of notes first introduces the GloVe model for training word vectors. Then it extends our discussion of word vectors (interchangeably called word embeddings) by seeing how they can be evaluated intrinsically and extrinsically. As we proceed, we discuss the example of word analogies as an intrinsic evaluation technique and how it can be used to tune word embedding techniques. We then discuss training model weights/parameters and word vectors for extrinsic tasks. Lastly we motivate artificial neural networks as a class of models for natural language processing tasks.

## 1 *Global Vectors for Word Representation (GloVe)*[3]

[3] This section is based on the GloVe paper by Pennington et al.:
Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. GloVe: Global Vectors for Word Representation

### 1.1 *Comparison with Previous Methods*

So far, we have looked at two main classes of methods to find word embeddings. The first set are count-based and rely on matrix factorization (e.g. LSA, HAL). While these methods effectively leverage global statistical information, they are primarily used to capture word similarities and do poorly on tasks such as word analogy, indicating a sub-optimal vector space structure. The other set of methods are shallow window-based (e.g. the skip-gram and the CBOW models), which learn word embeddings by making predictions in local context windows. These models demonstrate the capacity to capture complex linguistic patterns beyond word similarity, but fail to make use of the global co-occurrence statistics.

In comparison, GloVe consists of a weighted least squares model that trains on global word-word co-occurrence counts and thus makes efficient use of statistics. The model produces a word vector space with meaningful sub-structure. It shows state-of-the-art performance on the word analogy task, and outperforms other current methods on several word similarity tasks.

**GloVe:**

- Using global statistics to predict the probability of word j appearing in the context of word i with a least squares objective

### 1.2 Co-occurrence Matrix

Let $X$ denote the word-word co-occurrence matrix, where $X_{ij}$ indicates the number of times word $j$ occur in the context of word $i$. Let $X_i = \sum_k X_{ik}$ be the number of times any word k appears in the context of word i. Finally, let $P_{ij} = P(w_j|w_i) = \frac{X_{ij}}{X_i}$ be the probability of j appearing in the context of word i.

Populating this matrix requires a single pass through the entire corpus to collect the statistics. For large corpora, this pass can be computationally expensive, but it is a one-time up-front cost.

**Co-occurrence Matrix:**

- $X$: word-word co-occurrence matrix
- $X_{ij}$: number of times word $j$ occur in the context of word $i$
- $X_i = \sum_k X_{ik}$: the number of times any word k appears in the context of word i
- $P_{ij} = P(w_j|w_i) = \frac{X_{ij}}{X_i}$: the probability of j appearing in the context of word i

### 1.3 Least Squares Objective

Recall that for the skip-gram model, we use softmax to compute the probability of word j appears in the context of word i:

$$Q_{ij} = \frac{\exp(\vec{u}_j^T \vec{v}_i)}{\sum_{w=1}^{W} \exp(\vec{u}_w^T \vec{v}_i)}$$

Training proceeds in an on-line, stochastic fashion, but the implied global cross-entropy loss can be calculated as:

$$J = -\sum_{i \in corpus} \sum_{j \in context(i)} \log Q_{ij}$$

As the same words i and j can appear multiple times in the corpus, it is more efficient to first group together the same values for i and j:

$$J = -\sum_{i=1}^{W} \sum_{j=1}^{W} X_{ij} \log Q_{ij}$$

where the value of co-occurring frequency is given by the co-occurrence matrix $X$. One significant drawback of the cross-entropy loss is that it requires the distribution $Q$ to be properly normalized, which involves the expensive summation over the entire vocabulary. Instead, we use a least square objective in which the normalization factors in $P$ and $Q$ are discarded:

$$\hat{J} = \sum_{i=1}^{W} \sum_{j=1}^{W} X_i (\hat{P}_{ij} - \hat{Q}_{ij})^2$$

where $\hat{P}_{ij} = X_{ij}$ and $\hat{Q}_{ij} = \exp(\vec{u}_j^T \vec{v}_i)$ are the unnormalized distributions. This formulation introduces a new problem – $X_{ij}$ often takes on very large values and makes the optimization difficult. An effective change is to minimize the squared error of the logarithms of $\hat{P}$ and $\hat{Q}$:

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

Another observation is that the weighting factor $X_i$ is not guaranteed to be optimal. Instead, we introduce a more general weighting function, which we are free to take to depend on the context word as well:

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

### 1.4 *Conclusion*

In conclusion, the GloVe model efficiently leverages global statistical information by training only on the nonzero elements in a word-word co-occurrence matrix, and produces a vector space with meaningful sub-structure. It consistently outperforms *word2vec* on the word analogy task, given the same corpus, vocabulary, window size, and training time. It achieves better results faster, and also obtains the best results irrespective of speed.

## 2 *Evaluation of Word Vectors*

So far, we have discussed methods such as the *Word2Vec* and *GloVe* methods to train and discover latent vector representations of natural language words in a semantic space. In this section, we discuss how we can quantitatively evaluate the quality of word vectors produced by such techniques.

### 2.1 *Intrinsic Evaluation*

Intrinsic evaluation of word vectors is the evaluation of a set of word vectors generated by an embedding technique (such as Word2Vec or GloVe) on specific intermediate subtasks (such as analogy completion). These subtasks are typically simple and fast to compute and thereby allow us to help understand the system used to generate the word vectors. An intrinsic evaluation should typically return to us a number that indicates the performance of those word vectors on the evaluation subtask.

**Motivation:** Let us consider an example where our final goal is to create a question answering system which uses word vectors

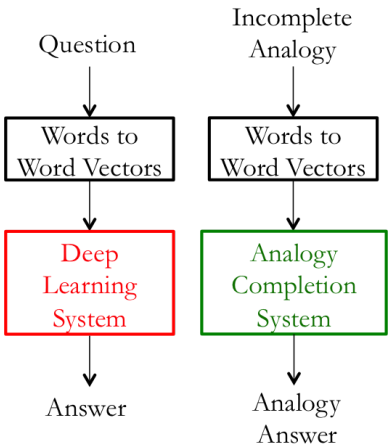



Figure 1: The left subsystem (red) being expensive to train is modified by substituting with a simpler subsystem (green) for intrinsic evaluation.

as inputs. One approach of doing so would be to train a machine learning system that:

1. Takes words as inputs
2. Converts them to word vectors
3. Uses word vectors as inputs for an elaborate machine learning system
4. Maps the output word vectors by this system back to natural language words
5. Produces words as answers

Of course, in the process of making such a state-of-the-art question-answering system, we will need to create optimal word-vector representations since they are used in downstream subsystems (such as deep neural networks). To do this in practice, we will need to tune many hyperparameters in the Word2Vec subsystem (such as the dimension of the word vector representation). While the idealistic approach is to retrain the entire system after any parametric changes in the Word2Vec subsystem, this is impractical from an engineering standpoint because the machine learning system (in step 3) is typically a deep neural network with millions of parameters that takes very long to train. In such a situation, we would want to come up with a simple intrinsic evaluation technique which can provide a measure of "goodness" of the word to word vector subsystem. Obviously, a requirement is that the intrinsic evaluation has a positive correlation with the final task performance.

**Intrinsic evaluation:**

- Evaluation on a specific, intermediate task
- Fast to compute performance
- Helps understand subsystem
- Needs positive correlation with real task to determine usefulness

### 2.2 *Extrinsic Evaluation*

Extrinsic evaluation of word vectors is the evaluation of a set of word vectors generated by an embedding technique on the real task at hand. These tasks are typically elaborate and slow to compute. Using our example from above, the system which allows for the evaluation of answers from questions is the extrinsic evaluation system. Typically, optimizing over an underperforming extrinsic evaluation system does not allow us to determine which specific subsystem is at fault and this motivates the need for intrinsic evaluation.

**Extrinsic evaluation:**

- Is the evaluation on a real task
- Can be slow to compute performance
- Unclear if subsystem is the problem, other subsystems, or internal interactions
- If replacing subsystem improves performance, the change is likely good

### 2.3 *Intrinsic Evaluation Example: Word Vector Analogies*

A popular choice for intrinsic evaluation of word vectors is its performance in completing word vector analogies. In a word vector analogy, we are given an incomplete analogy of the form:

a : b : : c : ?

The intrinsic evaluation system then identifies the word vector which maximizes the cosine similarity:

$$d = \underset{i}{\text{argmax}} \frac{(x_b - x_a + x_c)^T x_i}{\|x_b - x_a + x_c\|}$$

This metric has an intuitive interpretation. Ideally, we want $x_b - x_a = x_d - x_c$ (For instance, queen – king = actress – actor). This implies that we want $x_b - x_a + x_c = x_d$. Thus we identify the vector $x_d$ which maximizes the normalized dot-product between the two word vectors (i.e. cosine similarity).

Using intrinsic evaluation techniques such as word-vector analogies should be handled with care (keeping in mind various aspects of the corpus used for pre-training). For instance, consider analogies of the form:

City 1 : State containing City 1 : : City 2 : State containing City 2

| Input | Result Produced |
|---|---|
| Chicago : Illinois : : Houston | Texas |
| Chicago : Illinois : : Philadelphia | Pennsylvania |
| Chicago : Illinois : : Phoenix | Arizona |
| Chicago : Illinois : : Dallas | Texas |
| Chicago : Illinois : : Jacksonville | Florida |
| Chicago : Illinois : : Indianapolis | Indiana |
| Chicago : Illinois : : Austin | Texas |
| Chicago : Illinois : : Detroit | Michigan |
| Chicago : Illinois : : Memphis | Tennessee |
| Chicago : Illinois : : Boston | Massachusetts |

Table 1: Here are **semantic** word vector analogies (intrinsic evaluation) that may suffer from different cities having the same name

In many cases above, there are multiple cities/towns/villages with the same name across the US. Thus, many states would qualify as the right answer. For instance, there are at least 10 places in the US called Phoenix and thus, Arizona need not be the only correct response. Let us now consider analogies of the form:

Capital City 1 : Country 1 : : Capital City 2 : Country 2

In many of the cases above, the resulting city produced by this task has only been the capital in the recent past. For instance, prior to 1997 the capital of Kazakhstan was Almaty. Thus, we can anticipate other issues if our corpus is dated.

The previous two examples demonstrated semantic testing using word vectors. We can also test syntax using word vector analogies. The following intrinsic evaluation tests the word vectors' ability to capture the notion of superlative adjectives:

Similarly, the intrinsic evaluation shown below tests the word vectors' ability to capture the notion of past tense:

| Input | Result Produced |
|---|---|
| Abuja : Nigeria : : Accra | Ghana |
| Abuja : Nigeria : : Algiers | Algeria |
| Abuja : Nigeria : : Amman | Jordan |
| Abuja : Nigeria : : Ankara | Turkey |
| Abuja : Nigeria : : Antananarivo | Madagascar |
| Abuja : Nigeria : : Apia | Samoa |
| Abuja : Nigeria : : Ashgabat | Turkmenistan |
| Abuja : Nigeria : : Asmara | Eritrea |
| Abuja : Nigeria : : Astana | Kazakhstan |

Table 2: Here are **semantic** word vector analogies (intrinsic evaluation) that may suffer from countries having different capitals at different points in time

| Input | Result Produced |
|---|---|
| bad : worst : : big | biggest |
| bad : worst : : bright | brightest |
| bad : worst : : cold | coldest |
| bad : worst : : cool | coolest |
| bad : worst : : dark | darkest |
| bad : worst : : easy | easiest |
| bad : worst : : fast | fastest |
| bad : worst : : good | best |
| bad : worst : : great | greatest |

Table 3: Here are **syntactic** word vector analogies (intrinsic evaluation) that test the notion of superlative adjectives

### *2.4 Intrinsic Evaluation Tuning Example: Analogy Evaluations*

We now explore some of the hyperparameters in word vector embedding techniques (such as Word2Vec and GloVe) that can be tuned using an intrinsic evaluation system (such as an analogy completion system). Let us first see how different methods for creating word-vector embeddings have performed (in recent research work) under the same hyperparameters on an analogy evaluation task:

Some parameters we might consider tuning for a word embedding technique on intrinsic evaluation tasks are:

- Dimension of word vectors
- Corpus size
- Corpus souce/type
- Context window size
- Context symmetry

Can you think of other hyperparameters tunable at this stage?

| Input | Result Produced |
|---|---|
| dancing : danced : : decreasing | decreased |
| dancing : danced : : describing | described |
| dancing : danced : : enhancing | enhanced |
| dancing : danced : : falling | fell |
| dancing : danced : : feeding | fed |
| dancing : danced : : flying | flew |
| dancing : danced : : generating | generated |
| dancing : danced : : going | went |
| dancing : danced : : hiding | hid |
| dancing : danced : : hitting | hit |

Table 4: Here are **syntactic** word vector analogies (intrinsic evaluation) that test the notion of past tense

| Model | Dimension | Size | Semantics | Syntax | Total |
|---|---|---|---|---|---|
| ivLBL | 100 | 1.5B | 55.9 | 50.1 | 53.2 |
| HPCA | 100 | 1.6B | 4.2 | 16.4 | 10.8 |
| GloVE | 100 | 1.6B | 67.5 | 54.3 | 60.3 |
| SG | 300 | 1B | 61 | 61 | 61 |
| CBOW | 300 | 1.6B | 16.1 | 52.6 | 36.1 |
| vLBL | 300 | 1.5B | 54.2 | 64.8 | 60.0 |
| ivLBL | 300 | 1.5B | 65.2 | 63.0 | 64.0 |
| GloVe | 300 | 1.6B | 80.8 | 61.5 | 70.3 |
| SVD | 300 | 6B | 6.3 | 8.1 | 7.3 |
| SVD-S | 300 | 6B | 36.7 | 46.6 | 42.1 |
| SVD-L | 300 | 6B | 56.6 | 63.0 | 60.1 |
| CBOW | 300 | 6B | 63.6 | 67.4 | 65.7 |
| SG | 300 | 6B | 73.0 | 66.0 | 69.1 |
| GloVe | 300 | 6B | 77.4 | 67.0 | 71.7 |
| CBOW | 1000 | 6B | 57.3 | 68.9 | 63.7 |
| SG | 1000 | 6B | 66.1 | 65.1 | 65.6 |
| SVD-L | 300 | 42B | 38.4 | 58.2 | 49.2 |
| GloVe | 300 | 42B | 81.9 | 69.3 | 75.0 |

Table 5: Here we compare the performance of different models under the use of different hyperparameters and datasets

Inspecting the above table, we can make 3 primary observations:

- **Performance is heavily dependent on the model used for word embedding:**
  This is an expected result since different methods try embedding words to vectors using fundamentally different properties (such as co-occurrence count, singular vectors, etc.)
- **Performance increases with larger corpus sizes:**
  This happens because of the experience an embedding technique gains with more examples it sees. For instance, an analogy completion example will produce incorrect results if it has not encountered the test words previously.
- **Performance is lower for extremely low as well as for extremely high dimensional word vectors:**
  Lower dimensional word vectors are not able to capture the different meanings of the different words in the corpus. This can be viewed as a high bias problem where our model complexity is too low. For instance, let us consider the words "king", "queen", "man", "woman". Intuitively, we would need to use two dimensions such as "gender" and "leadership" to encode these into 2-bit word vectors. Any lower would fail to capture semantic differences between the four words and any more may capture noise in the corpus that doesn't help in generalization – this is also known as the high variance problem.

**Implementation Tip:** A window size of 8 around each center word typically works well for GloVe embeddings

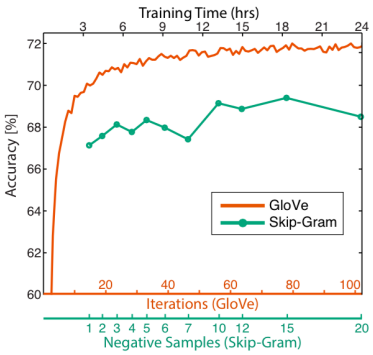



Figure 2: Here we see how training time improves training performance and helps squeeze the last few performance.

Figure 3 demonstrates how accuracy has been shown to improve with larger corpus.

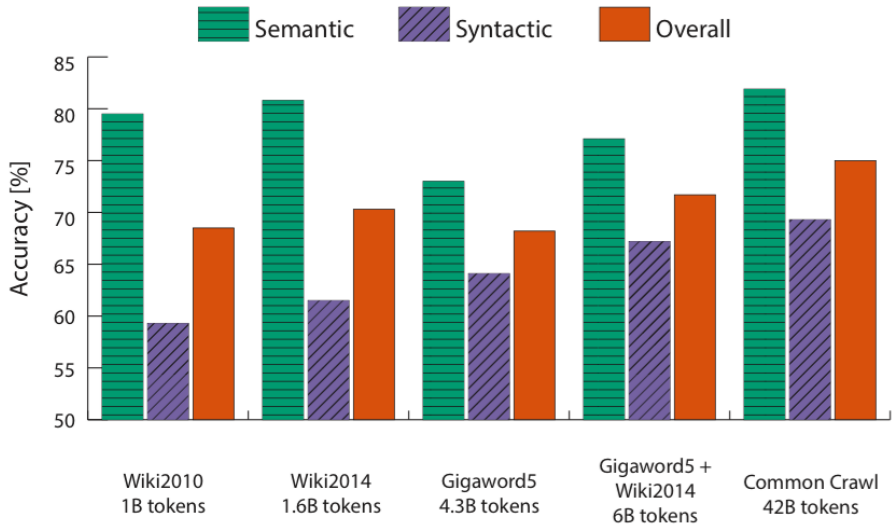
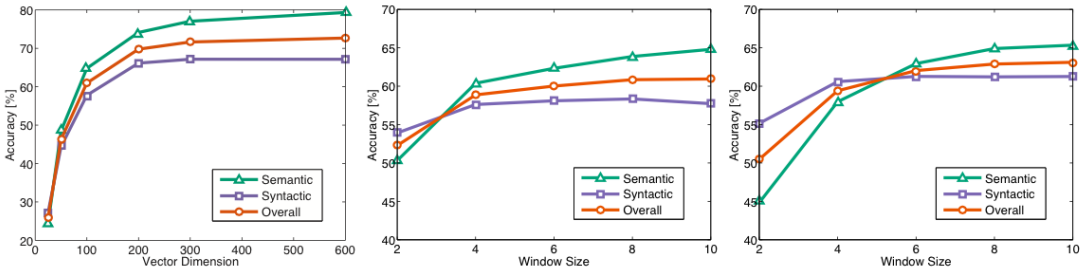



Figure 3: Here we see how performance improves with data size.

Figure 4 demonstrates how other hyperparameters have been shown to affect the accuracies using GloVe.




(a) Symmetric context (b) Symmetric context (c) Asymmetric context

Figure 4: We see how accuracies vary with vector dimension and context window size for GloVe

## 2.5 *Intrinsic Evaluation Example: Correlation Evaluation*

Another simple way to evaluate the quality of word vectors is by asking humans to assess the similarity between two words on a fixed scale (say 0-10) and then comparing this with the cosine similarity between the corresponding word vectors. This has been done on various datasets that contain human judgement survey data.

| Model | Size | WS353 | MC | RG | SCWS | RW |
|---|---|---|---|---|---|---|
| SVD | 6B | 35.3 | 35.1 | 42.5 | 38.3 | 25.6 |
| SVD-S | 6B | 56.5 | 71.5 | 71.0 | 53.6 | 34.7 |
| SVD-L | 6B | 65.7 | 72.7 | 75.1 | 56.5 | 37.0 |
| CBOW | 6B | 57.2 | 65.6 | 68.2 | 57.0 | 32.5 |
| SG | 6B | 62.8 | 65.2 | 69.7 | 58.1 | 37.2 |
| GloVe | 6B | 65.8 | 72.7 | 77.8 | 53.9 | 38.1 |
| SVD-L | 42B | 74.0 | 76.4 | 74.1 | 58.3 | 39.9 |
| GloVe | 42B | 75.9 | 83.6 | 82.9 | 59.6 | 47.8 |
| CBOW | 100B | 68.4 | 79.6 | 75.4 | 59.4 | 45.5 |

Table 6: Here we see the correlations between of word vector similarities using different embedding techniques with different human judgment datasets

## 2.6 *Further Reading: Dealing With Ambiguity*

One might wonder how we handle the situation where we want to capture the same word with different vectors for its different uses in natural language. For instance, "run" is both a noun and a verb and is used and interpreted differently based on the context. Improving Word Representations Via Global Context And Multiple Word Prototypes (Huang et al, 2012) describes how such cases can also be handled in NLP. The essence of the method is the following:

1. Gather fixed size context windows of all occurrences of the word (for instance, 5 before and 5 after)
2. Each context is represented by a weighted average of the context words' vectors (using idf-weighting)
3. Apply spherical k-means to cluster these context representations.
4. Finally, each word occurrence is re-labeled to its associated cluster and is used to train the word representation for that cluster.

For a more rigorous treatment on this topic, one should refer to the original paper.

## 3 *Training for Extrinsic Tasks*

We have so far focused on intrinsic tasks and emphasized their importance in developing a good word embedding technique. Of course, the end goal of most real-world problems is to use the resulting word vectors for some other extrinsic task. Here we discuss the general approach for handling extrinsic tasks.

### 3.1 *Problem Formulation*

Most NLP extrinsic tasks can be formulated as classification tasks. For instance, given a sentence, we can classify the sentence to have positive, negative or neutral sentiment. Similarly, in named-entity recognition (NER), given a context and a central word, we want to classify the central word to be one of many classes. For the input, "Jim bought 300 shares of Acme Corp. in 2006", we would like a classified output "$[\text{Jim}]_{\text{Person}}$ bought 300 shares of $[\text{Acme Corp.}]_{\text{Organization}}$ in $[2006]_{\text{Time}}$."

For such problems, we typically begin with a training set of the form:

$$\{x^{(i)}, y^{(i)}\}_1^N$$

where $x^{(i)}$ is a $d$-dimensional word vector generated by some word embedding technique and $y^{(i)}$ is a $C$-dimensional one-hot vector which indicates the labels we wish to eventually predict (sentiments, other words, named entities, buy/sell decisions, etc.).

In typical machine learning tasks, we usually hold input data and target labels fixed and train weights using optimization techniques (such as gradient descent, L-BFGS, Newton's method, etc.). In NLP applications however, we introduce the idea of retraining the input word vectors when we train for extrinsic tasks. Let us discuss when and why we should consider doing this.

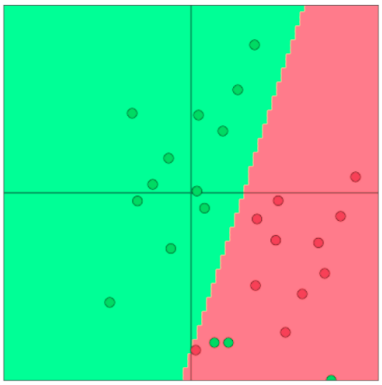


Figure 5: We can classify word vectors using simple linear decision boundaries such as the one shown here (2-D word vectors) using techniques such as logistic regression and SVMs

**Implementation Tip:** Word vector retraining should be considered for large training datasets. For small datasets, retraining word vectors will likely worsen performance.

### 3.2 *Retraining Word Vectors*

As we have discussed so far, the word vectors we use for extrinsic tasks are initialized by optimizing them over a simpler intrinsic task. In many cases, these pretrained word vectors are a good proxy for optimal word vectors for the extrinsic task and they perform well at the extrinsic task. However, it is also possible that the pretrained word vectors could be trained further (i.e. retrained) using the extrinsic task this time to perform better. However, retraining word vectors can be risky.

If we retrain word vectors using the extrinsic task, we need to ensure that the training set is large enough to cover most words from the vocabulary. This is because Word2Vec or GloVe produce semantically related words to be located in the same part of the word space. When we retrain these words over a small set of the vocabulary, these words are shifted in the word space and as a result, the performance over the final task could actually reduce. Let us explore this idea further using an example. Consider the pretrained vectors to be in a two dimensional space as shown in Figure 6. Here, we see that the word vectors are classified correctly on some extrinsic classification task. Now, if we retrain only two of those vectors because of a limited training set size, then we see in Figure 7 that one of the words gets misclassified because the boundary shifts as a result of word vector updates.

Thus, word vectors should not be retrained if the training data set is small. If the training set is large, retraining may improve performance.

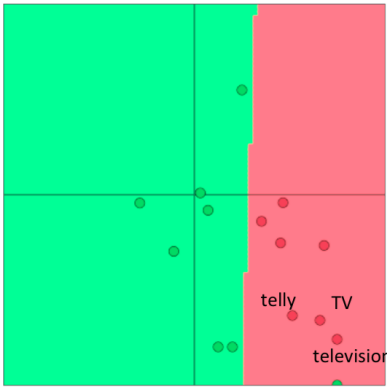



Figure 6: Here, we see that the words "Telly", "TV", and "Television" are classified correctly before retraining. "Telly" and "TV" are present in the extrinsic task training set while "Television" is only present in the test set.

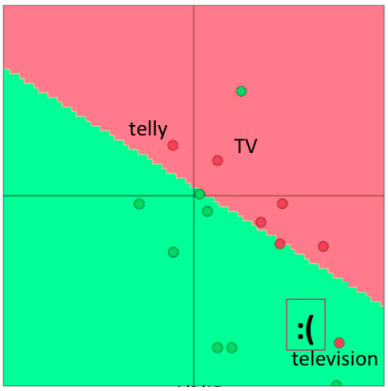



Figure 7: Here, we see that the words "Telly" and "TV" are classified correctly after traininng, but "Television" is not since it was not present in the training set.

### 3.3 *Softmax Classification and Regularization*

Let us consider using the Softmax classification function which has the form:

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

Here, we calculate the probability of word vector $x$ being in class $j$. Using the Cross-entropy loss function, we calculate the loss of such a training example as:

$$-\sum_{j=1}^{C} y_j \log(p(y_j = 1|x)) = -\sum_{j=1}^{C} y_j \log\left(\frac{\exp(W_{j\cdot}x)}{\sum_{c=1}^{C}\exp(W_{c\cdot}x)}\right)$$

Of course, the above summation will be a sum over $(C-1)$ zero values since $y_j$ is 1 only at a single index (at least for now) implying that $x$ belongs to only 1 correct class. Thus, let us define $k$ to be the

index of the correct class. Thus, we can now simplify our loss to be:

$$-\log\left(\frac{\exp(W_{k\cdot}x)}{\sum_{c=1}^{C}\exp(W_{c\cdot}x)}\right)$$

We can then extend the above loss to a dataset of $N$ points:

$$-\sum_{i=1}^{N}\log\left(\frac{\exp(W_{k(i)\cdot}x^{(i)})}{\sum_{c=1}^{C}\exp(W_{c\cdot}x^{(i)})}\right)$$

The only difference above is that $k(i)$ is now a function that returns the correct class index for example $x^{(i)}$.

Let us now try to estimate the number of parameters that would be updated if we consider training both, model weights ($W$), as well word vectors ($x$). We know that a simple linear decision boundary would require a model that takes in at least one $d$-dimensional input word vector and produces a distribution over $C$ classes. Thus, to update the model weights, we would be updating $C \cdot d$ parameters. If we update the word vectors for every word in the vocabulary $V$ as well, then we would be updating as many as $|V|$ word vectors, each of which is $d$-dimensional. Thus, the total number of parameters would be as many as $C \cdot d + |V| \cdot d$ for a simple linear classifier:

$$\nabla_\theta J(\theta) = \begin{bmatrix} \nabla_{W_{\cdot 1}} \\ \vdots \\ \nabla_{W_{\cdot d}} \\ \nabla_{x_{aardvark}} \\ \vdots \\ \nabla_{x_{zebra}} \end{bmatrix}$$

This is an extremely large number of parameters considering how simple the model's decision boundary is - such a large number of parameters is highly prone to overfitting.

To reduce overfitting risk, we introduce a regularization term which poses the Bayesian belief that the parameters ($\theta$) should be small is magnitude (i.e. close to zero):

$$-\sum_{i=1}^{N}\log\left(\frac{\exp(W_{k(i)\cdot}x^{(i)})}{\sum_{c=1}^{C}\exp(W_{c\cdot}x^{(i)})}\right) + \lambda\sum_{k=1}^{C\cdot d+|V|\cdot d}\theta_k^2$$

Minimizing the above cost function reduces the likelihood of the parameters taking on extremely large values just to fit the training set well and may improve generalization if the relative objective weight $\lambda$ is tuned well. The idea of regularization becomes even more of a requirement once we explore more complex models (such as Neural Networks) which have far more parameters.

### 3.4 *Window Classification*


Figure 8: Here, we see a central word with a symmetric window of length 2. Such context may help disambiguate between the place Paris and the name Paris.

So far we have primarily explored the idea of predicting in extrinsic tasks using a single word vector $x$. In reality, this is hardly done because of the nature of natural languages. Natural languages tend to use the same word for very different meanings and we typically need to know the context of the word usage to discriminate between meanings. For instance, if you were asked to explain to someone what "to sanction" meant, you would immediately realize that depending on the context "to sanction" could mean "to permit" or "to punish". In most situations, we tend to use a sequence of words as input to the model. A sequence is a central word vector preceded and succeeded by context word vectors. The number of words in the context is also known as the context window size and varies depending on the problem being solved. Generally, narrower window sizes lead to better performance in syntactic tests while wider windows lead to better performance in semantic tests.

Generally, narrower window sizes lead to better performance in syntactic tests while wider windows lead to better performance in semantic tests.

In order to modify the previously discussed Softmax model to use windows of words for classification, we would simply substitute $x^{(i)}$ with $x^{(i)}_{window}$ in the following manner:

$$x^{(i)}_{window} = \begin{bmatrix} x^{(i-2)} \\ x^{(i-1)} \\ x^{(i)} \\ x^{(i+1)} \\ x^{(i+2)} \end{bmatrix}$$

As a result, when we evaluate the gradient of the loss with respect to the words, we will receive gradients for the word vectors:

$$\delta_{window} = \begin{bmatrix} \nabla_{x^{(i-2)}} \\ \nabla_{x^{(i-1)}} \\ \nabla_{x^{(i)}} \\ \nabla_{x^{(i+1)}} \\ \nabla_{x^{(i+2)}} \end{bmatrix}$$

The gradient will of course need to be distributed to update the corresponding word vectors in implementation.

### 3.5 *Non-linear Classifiers*

We now introduce the need for non-linear classification models such as neural networks. We see in Figure 9 that a linear classifier misclassifies many datapoints. Using a non-linear decision boundary as shown in Figure 10, we manage to classify all training points accurately. Although oversimplified, this is a classic case demonstrating the need for non-linear decision boundaries. In the next set of notes,

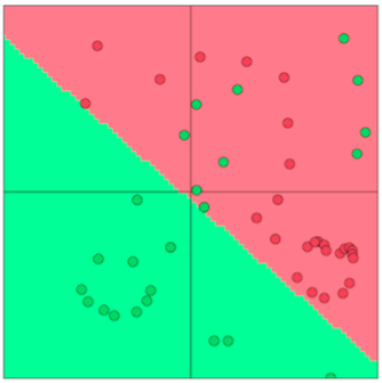


Figure 9: Here, we see that many examples are wrongly classified even though the best linear decision boundary is chosen. This is due linear decision boundaries have limited model capacity for this dataset.

we study neural networks as a class of non-linear models that have performed particularly well in deep learning applications.

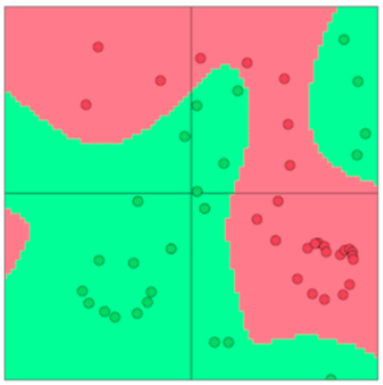


Figure 10: Here, we see that the non-linear decision boundary allows for much better classification of datapoints.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part III*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Author: Rohit Mundra, Amani Peddada, Richard Socher, Qiaojing Yan

**Keyphrases: Neural networks. Forward computation. Backward propagation. Neuron Units. Max-margin Loss. Gradient checks. Xavier parameter initialization. Learning rates. Adagrad.**
This set of notes introduces single and multilayer neural networks, and how they can be used for classification purposes. We then discuss how they can be trained using a distributed gradient descent technique known as backpropagation. We will see how the chain rule can be used to make parameter updates sequentially. After a rigorous mathematical discussion of neural networks, we will discuss some practical tips and tricks in training neural networks involving: neuron units (non-linearities), gradient checks, Xavier parameter initialization, learning rates, Adagrad, etc. Lastly, we will motivate the use of recurrent neural networks as a language model.

## 1 Neural Networks: Foundations

We established in our previous discussions the need for non-linear classifiers since most data are not linearly separable and thus, our classification performance on them is limited. Neural networks are a family of classifiers with non-linear decision boundary as seen in Figure 1. Now that we know the sort of decision boundaries neural networks create, let us see how they manage doing so.

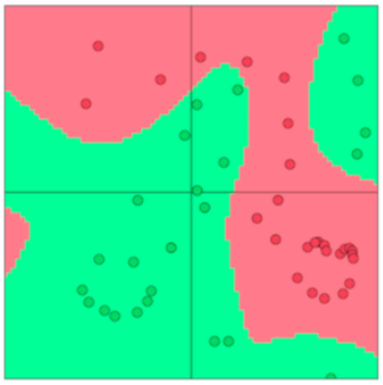


Figure 1: We see here how a non-linear decision boundary separates the data very well. This is the prowess of neural networks.

**Fun Fact:**
Neural networks are biologically inspired classifiers which is why they are often called "artificial neural networks" to distinguish them from the organic kind. However, in reality human neural networks are so much more capable and complex from artificial neural networks that it is usually better to not draw too many parallels between the two.

### 1.1 A Neuron

A neuron is a generic computational unit that takes $n$ inputs and produces a single output. What differentiates the outputs of different neurons is their parameters (also referred to as their weights). One of the most popular choices for neurons is the "sigmoid" or "binary logistic regression" unit. This unit takes an $n$-dimensional input vector $x$ and produces the scalar activation (output) $a$. This neuron is also associated with an $n$-dimensional weight vector, $w$, and a bias scalar, $b$. The output of this neuron is then:

$$a = \frac{1}{1 + \exp(-(w^T x + b))}$$

We can also combine the weights and bias term above to equiva-

**Neuron:**
A neuron is the fundamental building block of neural networks. We will see that a neuron can be one of many functions that allows for non-linearities to accrue in the network.

lently formulate:

$$a = \frac{1}{1 + \exp(-[w^T \quad b] \cdot [x \quad 1])}$$

This formulation can be visualized in the manner shown in Figure 2.

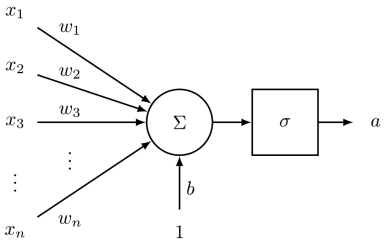



Figure 2: This image captures how in a sigmoid neuron, the input vector $x$ is first scaled, summed, added to a bias unit, and then passed to the squashing sigmoid function.

### 1.2 *A Single Layer of Neurons*

We extend the idea above to multiple neurons by considering the case where the input $x$ is fed as an input to multiple such neurons as shown in Figure 3.

If we refer to the different neurons' weights as $\{w^{(1)}, \cdots, w^{(m)}\}$ and the biases as $\{b_1, \cdots, b_m\}$, we can say the respective activations are $\{a_1, \cdots, a_m\}$:

$$a_1 = \frac{1}{1 + \exp(w^{(1)T} x + b_1)}$$

$$\vdots$$

$$a_m = \frac{1}{1 + \exp(w^{(m)T} x + b_m)}$$

Let us define the following abstractions to keep the notation simple and useful for more complex networks:

$$\sigma(z) = \begin{bmatrix} \frac{1}{1+\exp(z_1)} \\ \vdots \\ \frac{1}{1+\exp(z_m)} \end{bmatrix}$$

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix} \in \mathbb{R}^m$$

$$W = \begin{bmatrix} - & w^{(1)T} & - \\ & \cdots & \\ - & w^{(m)T} & - \end{bmatrix} \in \mathbb{R}^{m \times n}$$

We can now write the output of scaling and biases as:

$$z = Wx + b$$

The activations of the sigmoid function can then be written as:

$$\begin{bmatrix} a^{(1)} \\ \vdots \\ a^{(m)} \end{bmatrix} = \sigma(z) = \sigma(Wx + b)$$

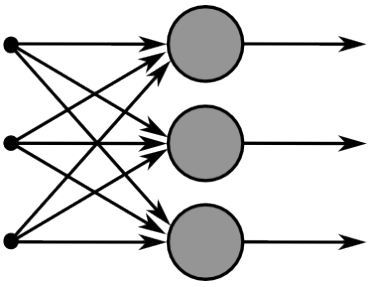


Figure 3: This image captures how multiple sigmoid units are stacked on the right, all of which receive the same input $x$.

So what do these activations really tell us? Well, one can think of these activations as indicators of the presence of some weighted combination of features. We can then use a combination of these activations to perform classification tasks.

### 1.3 Feed-forward Computation

So far we have seen how an input vector $x \in \mathbb{R}^n$ can be fed to a layer of sigmoid units to create activations $a \in \mathbb{R}^m$. But what is the intuition behind doing so? Let us consider the following named-entity recognition (NER) problem in NLP as an example:

*"Museums in Paris are amazing"*

Here, we want to classify whether or not the center word *"Paris"* is a named-entity. In such cases, it is very likely that we would not just want to capture the presence of words in the window of word vectors but some other interactions between the words in order to make the classification. For instance, maybe it should matter that *"Museums"* is the first word only if *"in"* is the second word. Such non-linear decisions can often not be captured by inputs fed directly to a Softmax function but instead require the scoring of the intermediate layer discussed in Section 1.2. We can thus use another matrix $U \in R^{m \times 1}$ to generate an unnormalized score for a classification task from the activations:

$$s = U^T a = U^T f(Wx + b)$$

where $f$ is the activation function.

**Analysis of Dimensions:** If we represent each word using a 4-dimensional word vector and we use a 5-word window as input (as in the above example), then the input $x \in \mathbb{R}^{20}$. If we use 8 sigmoid units in the hidden layer and generate 1 score output from the activations, then $W \in \mathbb{R}^{8 \times 20}$, $b \in \mathbb{R}^8$, $U \in \mathbb{R}^{8 \times 1}$, $s \in \mathbb{R}$.

### 1.4 Maximum Margin Objective Function

Like most machine learning models, neural networks also need an optimization objective, a measure of error or goodness which we want to minimize or maximize respectively. Here, we will discuss a popular error metric known as the maximum margin objective. The idea behind using this objective is to ensure that the score computed for "true" labeled data points is higher than the score computed for "false" labeled data points.

Using the previous example, if we call the score computed for the "true" labeled window *"Museums in Paris are amazing"* as $s$ and the

**Dimensions for a single hidden layer neural network:** If we represent each word using a 4-dimensional word vector and we use a 5-word window as input, then the input $x \in \mathbb{R}^{20}$. If we use 8 sigmoid units in the hidden layer and generate 1 score output from the activations, then $W \in \mathbb{R}^{8 \times 20}$, $b \in \mathbb{R}^8$, $U \in \mathbb{R}^{8 \times 1}$, $s \in \mathbb{R}$. The stage-wise feed-forward computation is then:

$$z = Wx + b$$
$$a = \sigma(z)$$
$$s = U^T a$$

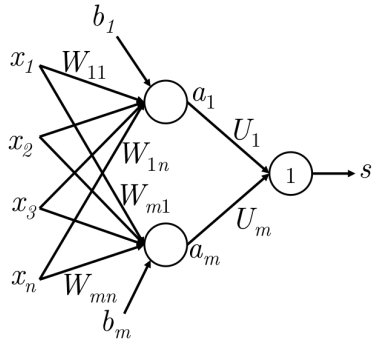



Figure 4: This image captures how a simple feed-forward network might compute its output.

score computed for the "false" labeled window *"Not all museums in Paris"* as $s_c$ (subscripted as $c$ to signify that the window is "corrupt").

Then, our objective function would be to maximize $(s - s_c)$ or to minimize $(s_c - s)$. However, we modify our objective to ensure that error is only computed if $s_c > s \Rightarrow (s_c - s) > 0$. The intuition behind doing this is that we only care the the "true" data point have a higher score than the "false" data point and that the rest does not matter. Thus, we want our error to be $(s_c - s)$ if $s_c > s$ else 0. Thus, our optimization objective is now:

$$\text{minimize } J = \max(s_c - s, 0)$$

However, the above optimization objective is risky in the sense that it does not attempt to create a margin of safety. We would want the "true" labeled data point to score higher than the "false" labeled data point by some positive margin $\Delta$. In other words, we would want error to be calculated if $(s - s_c < \Delta)$ and not just when $(s - s_c < 0)$. Thus, we modify the optimization objective:

$$\text{minimize } J = \max(\Delta + s_c - s, 0)$$

We can scale this margin such that it is $\Delta = 1$ and let the other parameters in the optimization problem adapt to this without any change in performance. For more information on this, read about functional and geometric margins - a topic often covered in the study of Support Vector Machines. Finally, we define the following optimization objective which we optimize over all training windows:

The max-margin objective function is most commonly associated with Support Vector Machines (SVMs)

$$\text{minimize } J = \max(1 + s_c - s, 0)$$

In the above formulation $s_c = U^T f(W x_c + b)$ and $s = U^T f(W x + b)$.

### 1.5 Training with Backpropagation – Elemental

In this section we discuss how we train the different parameters in the model when the cost $J$ discussed in Section 1.4 is positive. No parameter updates are necessary if the cost is 0. Since we typically update parameters using gradient descent (or a variant such as SGD), we typically need the gradient information for any parameter as required in the update equation:

$$\theta^{(t+1)} = \theta^{(t)} - \alpha \nabla_{\theta^{(t)}} J$$

Backpropagation is technique that allows us to use the chain rule of differentiation to calculate loss gradients for any parameter used in the feed-forward computation on the model. To understand this further, let us understand the toy network shown in Figure 5 for which we will perform backpropagation.

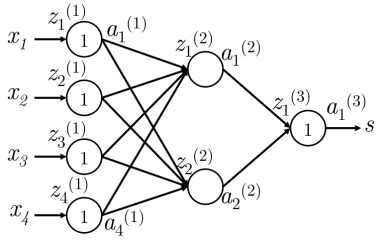



Figure 5: This is a 4-2-1 neural network where neuron $j$ on layer $k$ receives input $z_j^{(k)}$ and produces activation output $a_j^{(k)}$.

Here, we use a neural network with a single hidden layer and a single unit output. Let us establish some **notation** that will make it easier to generalize this model later:

- $x_i$ is an input to the neural network.
- $s$ is the output of the neural network.
- Each layer (including the input and output layers) has neurons which receive an input and produce an output. The $j$-th neuron of layer $k$ receives the scalar input $z_j^{(k)}$ and produces the scalar activation output $a_j^{(k)}$.
- We will call the backpropagated error calculated at $z_j^{(k)}$ as $\delta_j^{(k)}$.
- Layer 1 refers to the input layer and not the first hidden layer. For the input layer, $x_j = z_j^{(1)} = a_j^{(1)}$.
- $W^{(k)}$ is the transfer matrix that maps the output from the $k$-th layer to the input to the $(k+1)$-th Thus, $W^{(1)} = W$ and $W^{(2)} = U$ to put this new generalized notation in perspective of Section 1.3.

**Backpropagation Notation:**

- $x_i$ is an input to the neural network.
- $s$ is the output of the neural network.
- The $j$-th neuron of layer $k$ receives the scalar input $z_j^{(k)}$ and produces the scalar activation output $a_j^{(k)}$.
- For the input layer, $x_j = z_j^{(1)} = a_j^{(1)}$.
- $W^{(k)}$ is the transfer matrix that maps the output from the $k$-th layer to the input to the $(k+1)$-th. Thus, $W^{(1)} = W$ and $W^{(2)} = U^T$ using notation from Section 1.3.

**Let us begin:** Suppose the cost $J = (1 + s_c - s)$ is positive and we want to perform the update of parameter $W_{14}^{(1)}$ (in Figure 5 and Figure 6), we must realize that $W_{14}^{(1)}$ only contributes to $z_1^{(2)}$ and thus $a_1^{(2)}$. This fact is crucial to understanding backpropagation – backpropagated gradients are only affected by values they contribute to. $a_1^{(2)}$ is consequently used in the forward computation of score by multiplication with $W_1^{(2)}$. We can see from the max-margin loss that:

$$\frac{\partial J}{\partial s} = -\frac{\partial J}{\partial s_c} = -1$$

Therefore we will work with $\frac{\partial s}{\partial W_{ij}^{(1)}}$ here for simplicity. Thus,

$$\begin{aligned}
\frac{\partial s}{\partial W_{ij}^{(1)}} &= \frac{\partial W^{(2)} a^{(2)}}{\partial W_{ij}^{(1)}} = \frac{\partial W_i^{(2)} a_i^{(2)}}{\partial W_{ij}^{(1)}} = W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} \\
\Rightarrow W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial W_{ij}^{(1)}} &= W_i^{(2)} \frac{\partial a_i^{(2)}}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} \frac{f(z_i^{(2)})}{\partial z_i^{(2)}} \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}} \\
&= W_i^{(2)} f'(z_i^{(2)}) \frac{\partial z_i^{(2)}}{\partial W_{ij}^{(1)}}
\end{aligned}$$

$$
\begin{aligned}
&= W_i^{(2)} f'(z_i^{(2)}) \frac{\partial}{\partial W_{ij}^{(1)}} (b_i^{(1)} + a_1^{(1)} W_{i1}^{(1)} + a_2^{(1)} W_{i2}^{(1)} + a_3^{(1)} W_{i3}^{(1)} + a_4^{(1)} W_{i4}^{(1)}) \\
&= W_i^{(2)} f'(z_i^{(2)}) \frac{\partial}{\partial W_{ij}^{(1)}} (b_i^{(1)} + \sum_k a_k^{(1)} W_{ik}^{(1)}) \\
&= W_i^{(2)} f'(z_i^{(2)}) a_j^{(1)} \\
&= \delta_i^{(2)} \cdot a_j^{(1)}
\end{aligned}
$$

We see above that the gradient reduces to the product $\delta_i^{(2)} \cdot a_j^{(1)}$ where $\delta_i^{(2)}$ is essentially the error propagating backwards from the $i$-th neuron in layer 2. $a_j^{(1)}$ is an input fed to $i$-th neuron in layer 2 when scaled by $W_{ij}$.

Let us discuss the "error sharing/distribution" interpretation of backpropagation better using Figure 6 as an example. Say we were to update $W_{14}^{(1)}$:

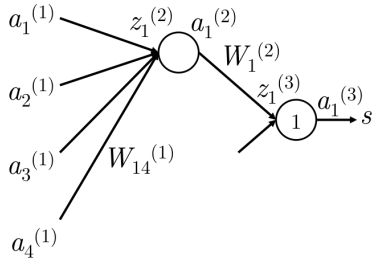



Figure 6: This subnetwork shows the relevant parts of the network required to update $W_{ij}^{(1)}$

1. We start with the an error signal of 1 propagating backwards from $a_1^{(3)}$.
2. We then multiply this error by the local gradient of the neuron which maps $z_1^{(3)}$ to $a_1^{(3)}$. This happens to be 1 in this case and thus, the error is still 1. This is now known as $\delta_1^{(3)} = 1$.
3. At this point, the error signal of 1 has reached $z_1^{(3)}$. We now need to distribute the error signal so that the "fair share" of the error reaches to $a_1^{(2)}$.
4. This amount is the (error signal at $z_1^{(3)} = \delta_1^{(3)}) \times W_1^{(2)} = W_1^{(2)}$. Thus, the error at $a_1^{(2)} = W_1^{(2)}$.
5. As we did in step 2, we need to move the error across the neuron which maps $z_1^{(2)}$ to $a_1^{(2)}$. We do this by multiplying the error signal at $a_1^{(2)}$ by the local gradient of the neuron which happens to be $f'(z_1^{(2)})$.
6. Thus, the error signal at $z_1^{(2)}$ is $f'(z_1^{(2)}) W_1^{(2)}$. This is known as $\delta_1^{(2)}$.
7. Finally, we need to distribute the "fair share" of the error to $W_{14}^{(1)}$ by simply multiplying it by the input it was responsible for forwarding, which happens to be $a_4^{(1)}$.
8. Thus, the gradient of the loss with respect to $W_{14}^{(1)}$ is calculated to be $a_4^{(1)} f'(z_1^{(2)}) W_1^{(2)}$.

Notice that the result we arrive at using this approach is exactly the same as that we arrived at using explicit differentiation earlier.

Thus, we can calculate error gradients with respect to a parameter in the network using either the chain rule of differentiation or using an error sharing and distributed flow approach – both of these approaches happen to do the exact same thing but it might be helpful to think about them one way or another.

**Bias Updates:** Bias terms (such as $b_1^{(1)}$) are mathematically equivalent to other weights contributing to the neuron input ($z_1^{(2)}$) as long as the input being forwarded is 1. As such, the bias gradients for neuron $i$ on layer $k$ is simply $\delta_i^{(k)}$. For instance, if we were updating $b_1^{(1)}$ instead of $W_{14}^{(1)}$ above, the gradient would simply be $f'(z_1^{(2)})W_1^{(2)}$.

**Generalized steps to propagate $\delta^{(k)}$ to $\delta^{(k-1)}$:**

1. We have error $\delta_i^{(k)}$ propagating backwards from $z_i^{(k)}$, i.e. neuron $i$ at layer $k$. See Figure 7.
2. We propagate this error backwards to $a_j^{(k-1)}$ by multiplying $\delta_i^{(k)}$ by the path weight $W_{ij}^{(k-1)}$.
3. Thus, the error received at $a_j^{(k-1)}$ is $\delta_i^{(k)}W_{ij}^{(k-1)}$.
4. However, $a_j^{(k-1)}$ may have been forwarded to multiple nodes in the next layer as shown in Figure 8. It should receive responsibility for errors propagating backward from node $m$ in layer $k$ too, using the exact same mechanism.
5. Thus, error received at $a_j^{(k-1)}$ is $\delta_i^{(k)}W_{ij}^{(k-1)} + \delta_m^{(k)}W_{mj}^{(k-1)}$.
6. In fact, we can generalize this to be $\sum_i \delta_i^{(k)}W_{ij}^{(k-1)}$.
7. Now that we have the correct error at $a_j^{(k-1)}$, we move it across neuron $j$ at layer $k-1$ by multiplying with with the local gradient $f'(z_j^{(k-1)})$.
8. Thus, the error that reaches $z_j^{(k-1)}$, called $\delta_j^{(k-1)}$ is $f'(z_j^{(k-1)})\sum_i \delta_i^{(k)}W_{ij}^{(k-1)}$

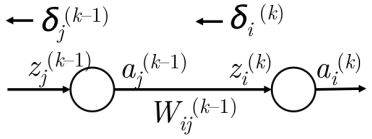



Figure 7: Propagating error from $\delta^{(k)}$ to $\delta^{(k-1)}$

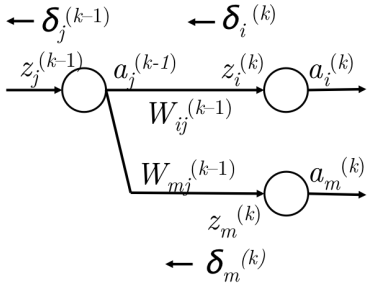



Figure 8: Propagating error from $\delta^{(k)}$ to $\delta^{(k-1)}$

### 1.6 Training with Backpropagation – Vectorized

So far, we discussed how to calculate gradients for a given parameter in the model. Here we will generalize the approach above so that we update weight matrices and bias vectors all at once. Note that these are simply extensions of the above model that will help build intuition for the way error propagation can be done at a matrix-vector level.

For a given parameter $W_{ij}^{(k)}$, we identified that the error gradient is simply $\delta_i^{(k+1)} \cdot a_j^{(k)}$. As a reminder, $W^{(k)}$ is the matrix that maps $a^{(k)}$ to $z^{(k+1)}$. We can thus establish that the error gradient for the entire matrix $W^{(k)}$ is:

$$\nabla_{W^{(k)}} = \begin{bmatrix} \delta_1^{(k+1)} a_1^{(k)} & \delta_1^{(k+1)} a_2^{(k)} & \cdots \\ \delta_2^{(k+1)} a_1^{(k)} & \delta_2^{(k+1)} a_2^{(k)} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \delta^{(k+1)} a^{(k)T}$$

Thus, we can write an entire matrix gradient using the outer product of the error vector propagating into the matrix and the activations forwarded by the matrix.

Now, we will see how we can calculate the error vector $\delta^{(k)}$. We established earlier using Figure 8 that $\delta_j^{(k)} = f'(z_j^{(k)}) \sum_i \delta_i^{(k+1)} W_{ij}^{(k)}$. This can easily generalize to matrices such that:

$$\delta^{(k)} = f'(z^{(k)}) \circ (W^{(k)T} \delta^{(k+1)})$$

In the above formulation, the $\circ$ operator corresponds to an element wise product between elements of vectors ($\circ : \mathbb{R}^N \times \mathbb{R}^N \to \mathbb{R}^N$).

**Error propagates** from layer $(k+1)$ to $(k)$ in the following manner:

$$\delta^{(k)} = f'(z^{(k)}) \circ (W^{(k)T} \delta^{(k+1)})$$

Of course, this assumes that in the forward propagation the signal $z^{(k)}$ first goes through activation neurons $f$ to generate activations $a^{(k)}$ and are then linearly combined to yield $z^{(k+1)}$ via transfer matrix $W^{(k)}$.

**Computational efficiency:** Having explored element-wise updates as well as vector-wise updates, we must realize that the vectorized implementations run substantially faster in scientific computing environments such as MATLAB or Python (using NumPy/SciPy packages). Thus, we should use vectorized implementation in practice. Furthermore, we should also reduce redundant calculations in backpropagation - for instance, notice that $\delta^{(k)}$ depends directly on $\delta^{(k+1)}$. Thus, we should ensure that when we update $W^{(k)}$ using $\delta^{(k+1)}$, we save $\delta^{(k+1)}$ to later derive $\delta^{(k)}$ – and we then repeat this for $(k-1) \dots (1)$. Such a recursive procedure is what makes backpropagation a computationally affordable procedure.

## 2 Neural Networks: Tips and Tricks

Having discussed the mathematical foundations of neural networks, we will now dive into some tips and tricks commonly employed

when using neural networks in practice.

## 2.1 Gradient Check

In the last section, we discussed in detail how to calculate error gradients/updates for parameters in a neural network model via calculus-based (analytic) methods. Here we now introduce a technique of *numerically* approximating these gradients – though too computationally inefficient to be used directly for training the networks, this method will allow us to very precisely estimate the derivative with respect to any parameter; it can thus serve as a useful sanity check on the correctness of our analytic derivatives. Given a model with parameter vector $\theta$ and loss function $J$, the numerical gradient around $\theta_i$ is simply given by **centered difference formula**:

$$f'(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

where $\epsilon$ is a small number (usually around $1e^{-5}$). The term $J(\theta^{(i+)})$ is simply the error calculated on a forward pass for a given input when we perturb the parameter $\theta$'s $i^{th}$ element by $+\epsilon$. Similarly, the term $J(\theta^{(i-)})$ is the error calculated on a forward pass for the same input when we perturb the parameter $\theta$'s $i^{th}$ element by $-\epsilon$. Thus, using two forward passes, we can approximate the gradient with respect to any given parameter element in the model. We note that this definition of the numerical gradient follows very naturally from the definition of the derivative, where, in the scalar case,

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Of course, there is a slight difference – the definition above only perturbs $x$ in the positive direction to compute the gradient. While it would have been perfectly acceptable to define the numerical gradient in this way, in practice it is often more precise and stable to use the centered difference formula, where we perturb a parameter in both directions. The intuition is that to get a better approximation of the derivative/slope around a point, we need to examine the function $f$'s behavior both to the left and right of that point. It can also be shown using Taylor's theorem that the centered difference formula has an error proportional to $\epsilon^2$, which is quite small, whereas the derivative definition is more error-prone.

Now, a natural question you might ask is, if this method is so precise, why do we not use it to compute all of our network gradients instead of applying back-propagation? The simple answer, as hinted earlier, is inefficiency – recall that every time we want to compute the gradient with respect to an element, we need to make two forward

**Gradient checks** are a great way to compare analytical and numerical gradients. Analytical gradients should be close and numerical gradients can be calculated using:

$$f'(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon}$$

$J(\theta^{(i+)})$ and $J(\theta^{(i-)})$ can be evaluated using two forward passes. An implementation of this can be seen in Snippet 2.1.

passes through the network, which will be computationally expensive. Furthermore, many large-scale neural networks can contain millions of parameters,and computing two passes per parameter is clearly not optimal. And, since in optimization techniques such as SGD, we must compute the gradients once per iteration for several thousands of iterations, it is obvious that this method quickly grows intractable. This inefficiency is why we only use gradient check to verify the correctness of our analytic gradients, which are much quicker to compute. A standard implementation of gradient check is shown below:

**Snippet 2.1**

```
def eval_numerical_gradient(f, x):
  """
  a naive implementation of numerical gradient of f at x
  - f should be a function that takes a single argument
  - x is the point (numpy array) to evaluate the gradient
  at
  """

  fx = f(x) # evaluate function value at original point
  grad = np.zeros(x.shape)
  h = 0.00001

  # iterate over all indexes in x
  it = np.nditer(x, flags=['multi_index'],
                  op_flags=['readwrite'])
  while not it.finished:

    # evaluate function at x+h
    ix = it.multi_index
    old_value = x[ix]
    x[ix] = old_value + h # increment by h
    fxh_left = f(x) # evaluate f(x + h)
    x[ix] = old_value - h # decrement by h
    fxh_right = f(x) # evaluate f(x - h)
    x[ix] = old_value # restore to previous value (very
           important!)

    # compute the partial derivative
    grad[ix] = (fxh_left - fxh_right) / (2*h) # the slope
    it.iternext() # step to next dimension
  return grad
```

### 2.2 *Regularization*

As with many machine learning models, neural networks are highly prone to overfitting, where a model is able to obtain near perfect performance on the training dataset, but loses the ability to generalize to unseen data. A common technique used to address overfitting (an issue also known as the "high-variance problem") is the incorporation of an $L_2$ regularization penalty. The idea is that we will simply append an extra term to our loss function $J$, so that the overall cost is now calculated as:

$$J_R = J + \lambda \sum_{i=1}^{L} \left\| W^{(i)} \right\|_F$$

In the above formulation, $\left\| W^{(i)} \right\|_F$ is the Frobenius norm of the matrix $W^{(i)}$ (the $i$-th weight matrix in the network) and $\lambda$ is the hyper-parameter controlling how much weight the regularization term has relative to the original cost function. Since we are trying to minimize $J_R$, what regularization is essentially doing is penalizing weights for being too large while optimizing over the original cost function. Due to the quadratic nature of the Frobenius norm (which computes the sum of the squared elements of a matrix), $L_2$-regularization effectively reduces the flexibility of the model and thereby reduces the overfitting phenomenon. Imposing such a constraint can also be interpreted as the prior Bayesian belief that the optimal weights are close to zero – how close depends on the value of $\lambda$. Choosing the right value of $\lambda$ is critical, and must be chosen via hyperparameter-tuning. Too high a value of $\lambda$ causes most of the weights to be set too close to 0, and the model does not learn anything meaningful from the training data, often obtaining poor accuracy on training, validation, and testing sets. Too low a value, and we fall into the domain of overfitting once again. It must be noted that the bias terms are not regularized and do not contribute to the cost term above – try thinking about why this is the case!

There are indeed other types of regularization that are sometimes used, such as $L_1$ regularization, which sums over the absolute values (rather than squares) of parameter elements – however, this is less commonly applied in practice since it leads to sparsity of parameter weights. In the next section, we discuss *dropout*, which effectively acts as another form of regularization by randomly dropping (i.e. setting to zero) neurons in the forward pass.

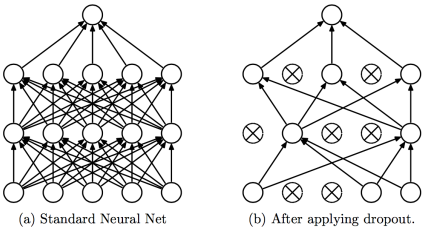
The **Frobenius Norm** of a matrix $U$ is defined as follows:

$$||U||_F = \sum_i \sum_j U_{ij^2}$$

### 2.3 Dropout

Dropout is a powerful technique for regularization, first introduced by Srivastava et al. in *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. The idea is simple yet effective – during training, we will randomly "drop" with some probability $(1-p)$ a subset of neurons during each forward/backward pass (or equivalently, we will keep alive each neuron with a probability $p$). Then, during testing, we will use the full network to compute our predictions. The result is that the network typically learns more meaningful information from the data, is less likely to overfit, and usually obtains higher performance overall on the task at hand. One intuitive reason why this technique should be so effective is that what dropout is doing is essentially doing is training exponentially many smaller networks at once and averaging over their predictions.

In practice, the way we introduce dropout is that we take the output $h$ of each layer of neurons, and keep each neuron with probability $p$, and else set it to 0. Then, during back-propagation, we only pass gradients through neurons that were kept alive during the forward pass. Finally, during testing, we compute the forward pass using *all* of the neurons in the network. However, a key subtlety is that in order for dropout to work effectively, the expected output of a neuron during testing should be approximately the same as it was during training – else the magnitude of the outputs could be radically different, and the behavior of the network is no longer well-defined. Thus, we must typically divide the outputs of each neuron during testing by a certain value – it is left as an exercise to the reader to determine what this value should be in order for the expected outputs during training and testing to be equivalent.



(a) Standard Neural Net (b) After applying dropout.

Dropout applied to an artificial neural network. Image credits to Srivastava et al.

### 2.4 Neuron Units

So far we have discussed neural networks that contain sigmoidal neurons to introduce nonlinearities; however in many applications better networks can be designed using other activation functions. Some common choices are listed here with their function and gradient definitions and these can be substituted with the sigmoidal functions discussed above.

**Sigmoid:** This is the default choice we have discussed; the activation function $\sigma$ is given by:

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

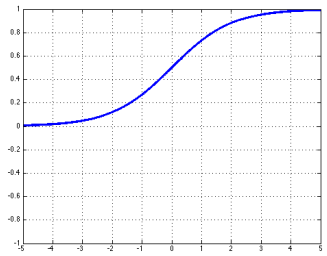


Figure 9: The response of a sigmoid nonlinearity

$$\text{where } \sigma(z) \in (0, 1)$$

The gradient of $\sigma(z)$ is:

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

**Tanh:** The tanh functionis an alternative to the sigmoid function that is often found to converge faster in practice. The primary difference between tanh and sigmoid is that tanh output ranges from $-1$ to 1 while the sigmoid ranges from 0 to 1.

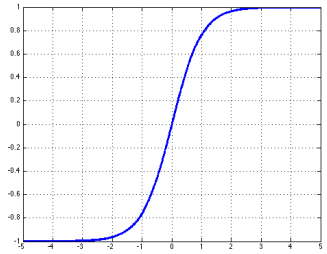


Figure 10: The response of a tanh nonlinearity

$$\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)} = 2\sigma(2z) - 1$$
$$\text{where } \tanh(z) \in (-1, 1)$$

The gradient of $\tanh(z)$ is:

$$\tanh'(z) = 1 - \left(\frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}\right)^2 = 1 - \tanh^2(z)$$

**Hard tanh:** The hard tanh function is sometimes preferred over the tanh function since it is computationally cheaper. It does however saturate for magnitudes of $z$ greater than 1. The activation of the hard tanh is:

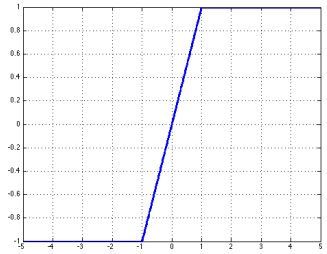


Figure 11: The response of a hard tanh nonlinearity

$$\text{hardtanh}(z) = \begin{cases} -1 & : z < -1 \\ z & : -1 \leq z \leq 1 \\ 1 & : z > 1 \end{cases}$$

The derivative can also be expressed in a piecewise functional form:

$$\text{hardtanh}'(z) = \begin{cases} 1 & : -1 \leq z \leq 1 \\ 0 & : \text{otherwise} \end{cases}$$

**Soft sign:** The soft sign function is another nonlinearity which can be considered an alternative to tanh since it too does not saturate as easily as hard clipped functions:

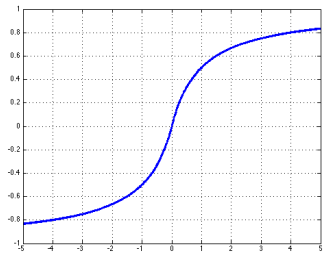


Figure 12: The response of a soft sign nonlinearity

$$\text{softsign}(z) = \frac{z}{1+|z|}$$

The derivative is the expressed as:

$$\text{softsign}'(z) = \frac{\text{sgn}(z)}{(1+z)^2}$$

where sgn is the signum function which returns $\pm 1$ depending on the sign of $z$

**ReLU:** The ReLU (Rectified Linear Unit) function is a popular choice of activation since it does not saturate even for larger values of $z$ and has found much success in computer vision applications:

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

The derivative is then the piecewise function:

$$\text{rect}'(z) = \begin{cases} 1 & : z > 0 \\ 0 & : \text{otherwise} \end{cases}$$

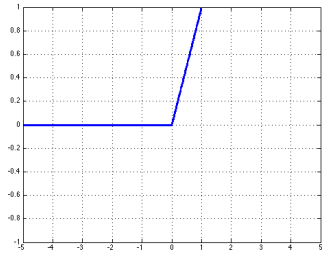


Figure 13: The response of a ReLU nonlinearity

**Leaky ReLU:** Traditional ReLU units by design do not propagate any error for non-positive $z$ – the leaky ReLU modifies this such that a small error is allowed to propagate backwards even when $z$ is negative:

$$\text{leaky}(z) = \max(z, k \cdot z)$$

$$\text{where } 0 < k < 1$$

This way, the derivative is representable as:

$$\text{leaky}'(z) = \begin{cases} 1 & : z > 0 \\ k & : \text{otherwise} \end{cases}$$

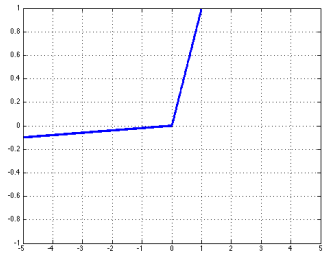


Figure 14: The response of a leaky ReLU nonlinearity

## 2.5 *Data Preprocessing*

As is the case with machine learning models generally, a key step to ensuring that your model obtains reasonable performance on the task at hand is to perform basic preprocessing on your data. Some common techniques are outlined below.

**Mean Subtraction**

Given a set of input data $X$, it is customary to zero-center the data by subtracting the mean feature vector of $X$ from $X$. An important point is that in practice, the mean is calculated only across the training set, and this mean is subtracted from the training, validation, and testing sets.

**Normalization**

Another frequently used technique (though perhaps less so than mean subtraction) is to scale every input feature dimension to have similar ranges of magnitudes. This is useful since input features are often measured in different "units", but we often want to initially consider all features as equally important. The way we accomplish this is by simply dividing the features by their respective standard deviation calculated across the training set.

**Whitening**
Not as commonly used as mean-subtraction + normalization, whitening essentially converts the data to a have an identity covariance matrix – that is, features become uncorrelated and have a variance of 1. This is done by first mean-subtracting the data, as usual, to get $X'$. We can then take the Singular Value Decomposition (SVD) of $X'$ to get matrices $U, S, V$. We then compute $UX'$ to project $X'$ into the basis defined by the columns of $U$. We finally divide each dimension of the result by the corresponding singular value in $S$ to scale our data appropriately (if a singular value is zero, we can just divide by a small number instead).

## 2.6 *Parameter Initialization*

A key step towards achieving superlative performance with a neural network is initializing the parameters in a reasonable way. A good starting strategy is to initialize the weights to small random numbers normally distributed around 0 – and in practice, this often words acceptably well. However, in UNDERSTANDING THE DIFFICULTY OF TRAINING DEEP FEEDFORWARD NEURAL NETWORKS (2010), XAVIER ET AL study the effect of different weight and bias initialization schemes on training dynamics. The empirical findings suggest that for sigmoid and tanh activation units, faster convergence and lower error rates are achieved when the weights of a matrix $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$ are initialized randomly with a uniform distribution as follows:

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)}+n^{(l+1)}}}\right]$$

Where $n^{(l)}$ is the number of input units to $W$ (fan-in) and $n^{(l+1)}$ is the number of output units from $W$ (fan-out). In this parameter initialization scheme, bias units are initialized to 0. This approach attempts to maintain activation variances as well as backpropagated gradient variances across layers. Without such initialization, the gradient variances (which are a proxy for information) generally decrease with backpropagation across layers.

## 2.7 *Learning Strategies*

The rate/magnitude of model parameter updates during training can be controlled using the learning rate. In the following naive Gradient Descent formulation, $\alpha$ is the learning rate:

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

You might think that for fast convergence rates, we should set $\alpha$ to larger values – however faster convergence is not guaranteed with larger convergence rates. In fact, with very large learning rates, we might experience that the loss function actually diverges because the parameters update causes the model to overshoot the convex minima as shown in Figure 15. In non-convex models (most of those we work with), the outcome of a large learning rate is unpredictable, but the chances of diverging loss functions are very high .

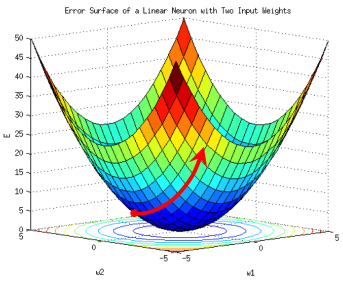



Figure 15: Here we see that updating parameter $w_2$ with a large learning rate can lead to divergence of the error.

The simple solution to avoiding a diverging loss is to use a very small learning rate so that we carefully scan the parameter space – of course, if we use too small a learning rate, we might not converge in a reasonable amount of time, or might get caught in a local minima. Thus, as with any other hyperparameter, the learning rate must be tuned effectively.

Since training is the most expensive phase in a deep learning system, some research has attempted to improve this naive approach to setting learning learning rates. For instance, Ronan Collobert scales the learning rate of a weight $W_{ij}$ (where $W \in \mathbb{R}^{n^{(l+1)} \times n^{(l)}}$) by the inverse square root of the fan-in of the neuron ($n^{(l)}$).

There are several other techniques that have proven to be effective as well – one such method is **annealing**, where, after several iterations, the learning rate is reduced in some way – this method ensures that we start off with a high learning rate and approach a minima quickly; as we get closer to the minima, we start lowering our learning rate so that we can find the optima under a more fine-grained scope. A common way to perform annealing is to reduce the learning rate $\alpha$ by a factor $x$ after every $n$ iterations of learning. Exponential decay is also common, where, the learning rate $\alpha$ at iteration $t$ is given by $\alpha(t) = \alpha_0 e^{-kt}$, where $\alpha_0$ is the initial learning rate, and $k$ is a hyperparameter. Another approach is to allow the learning rate to decrease over time such that:

$$\alpha(t) = \frac{\alpha_0 \tau}{\max(t, \tau)}$$

In the above scheme, $\alpha_0$ is a tunable parameter and represents the starting learning rate. $\tau$ is also a tunable parameter and represents the time at which the learning rate should start reducing. In practice, this method has been found to work quite well. In the next section we discuss another method for adaptive gradient descent which does not require hand-set learning rates.

### 2.8 *Momentum Updates*

Momentum methods, a variant of gradient descent inspired by the study of dynamics and motion in physics, attempt to use the "veloc-

ity" of updates as a more effective update scheme. Pseudocode for momentum updates is shown below:

**Snippet 2.2**
```
# Computes a standard momentum update
# on parameters x
v = mu*v - alpha*grad_x
x += v
```

## 2.9 *Adaptive Optimization Methods*

AdaGrad is an implementation of standard stochastic gradient descent (SGD) with one key difference: the learning rate can vary for each parameter. The learning rate for each parameter depends on the history of gradient updates of that parameter in a way such that parameters with a scarce history of updates are updated faster using a larger learning rate. In other words, parameters that have not been updated much in the past are likelier to have higher learning rates now. Formally:

$$\theta_{t,i} = \theta_{t-1,i} - \frac{\alpha}{\sqrt{\sum_{\tau=1}^{t} g_{\tau,i}^2}} g_{t,i} \text{ where } g_{t,i} = \frac{\partial}{\partial \theta_i^t} J_t(\theta)$$

In this technique, we see that if the RMS of the history of gradients is extremely low, the learning rate is very high. A simple implementation of this technique is:

**Snippet 2.3**
```
# Assume the gradient dx and parameter vector x
cache += dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

Other common adaptive methods are RMSProp and Adam, whose update rules are shown below (courtesy of Andrej Karpathy):

**Snippet 2.4**
```
# Update rule for RMS prop
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / (np.sqrt(cache) + eps)
```

**Snippet 2.5**
```
# Update rule for Adam
```

```
m = beta1*m + (1-beta1)*dx
v = beta2*v + (1-beta2)*(dx**2)
x += - learning_rate * m / (np.sqrt(v) + eps)
```

RMSProp is a variant of AdaGrad that utilizes a moving average of squared gradients – in particular, unlike AdaGrad, its updates do not become monotonically smaller. The Adam update rule is in turn a variant of RMSProp, but with the addition of momentum-like updates. We refer the reader to the respective sources of these methods for more detailed analyses of their behavior.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part IV*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Lisa Wang, Juhi Naik, and Shayne Longpre

**Keyphrases: Dependency Parsing.**

## 1 *Dependency Grammar and Dependency Structure*

Parse trees in NLP, analogous to those in compilers, are used to analyze the syntactic structure of sentences. There are two main types of structures used - constituency structures and dependency structures.

Constituency Grammar uses phrase structure grammar to organize words into nested constituents. This will be covered in more detail in following chapters. We now focus on Dependency Parsing.

Dependency structure of sentences shows which words depend on (modify or are arguments of) which other words. These binary asymmetric relations between the words are called dependencies and are depicted as arrows going from the **head** (or governor, superior, regent) to the **dependent** (or modifier, inferior, subordinate). Usually these dependencies form a tree structure. They are often typed with the name of grammatical relations (subject, prepositional object, apposition, etc.). An example of such a dependency tree is shown in Figure 1. Sometimes a fake ROOT node is added as the head to the whole tree so that every word is a dependent of exactly one node.

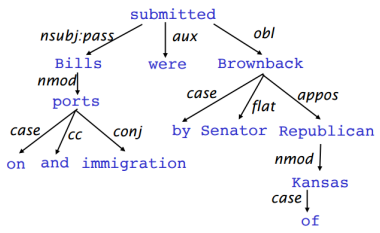



Figure 1: Dependency tree for the sentence "Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas"

### 1.1 *Dependency Parsing*

Dependency parsing is the task of analyzing the syntactic dependency structure of a given input sentence $S$. The output of a dependency parser is a dependency tree where the words of the input sentence are connected by typed dependency relations. Formally, the dependency parsing problem asks to create a mapping from the input sentence with words $S = w_0 w_1 ... w_n$ (where $w_0$ is the ROOT) to its dependency tree graph $G$. Many different variations of dependency-based methods have been developed in recent years, including neural network-based methods, which we will describe later.

To be precise, there are two subproblems in dependency parsing (adapted from Kuebler et al., chapter 1.2):

1. *Learning:* Given a training set $D$ of sentences annotated with dependency graphs, induce a parsing model $M$ that can be used to parse new sentences.

2. *Parsing:* Given a parsing model $M$ and a sentence $S$, derive the optimal dependency graph $D$ for $S$ according to $M$.

### 1.2 Transition-Based Dependency Parsing

Transition-based dependency parsing relies on a state machine which defines the possible transitions to create the mapping from the input sentence to the dependency tree. The *learning problem* is to induce a model which can predict the next transition in the state machine based on the transition history. The *parsing problem* is to construct the optimal sequence of transitions for the input sentence, given the previously induced model. Most transition-based systems do not make use of a formal grammar.

### 1.3 Greedy Deterministic Transition-Based Parsing

This system was introduced by Nivre in 2003 and was radically different from other methods in use at that time.

This transition system is a state machine, which consists of *states* and *transitions* between those states. The model induces a sequence of transitions from some *initial* state to one of several *terminal* states.

**States:**

For any sentence $S = w_0 w_1 ... w_n$, a state can be described with a triple $c = (\sigma, \beta, A)$:

1. a stack $\sigma$ of words $w_i$ from $S$,
2. a buffer $\beta$ of words $w_i$ from $S$,
3. a set of dependency arcs $A$ of the form $(w_i, r, w_j)$, where $w_i, w_j$ are from $S$, and $r$ describes a dependency relation.

It follows that for any sentence $S = w_0 w_1 ... w_n$,

1. an *initial* state $c_0$ is of the form $([w_0]_\sigma, [w_1, ..., w_n]_\beta, \emptyset)$ (only the ROOT is on the stack $\sigma$, all other words are in the buffer $\beta$ and no actions have been chosen yet),
2. a terminal state has the form $(\sigma, []_\beta, A)$.

**Transitions:**

There are three types of transitions between states:

1. SHIFT: Remove the first word in the buffer and push it on top of the stack. (Pre-condition: buffer has to be non-empty.)
2. LEFT-ARC$_r$: Add a dependency arc $(w_j, r, w_i)$ to the arc set $A$, where $w_i$ is the word second to the top of the stack and $w_j$ is the

1. Shift $\quad \sigma, w_i|\beta, A \rightarrow \sigma|w_i, \beta, A$
2. Left-Arc$_r$ $\quad \sigma|w_i|w_j, \beta, A \rightarrow \sigma|w_j, \beta, A \cup \{r(w_j, w_i)\}$
3. Right-Arc$_r$ $\quad \sigma|w_i|w_j, \beta, A \rightarrow \sigma|w_i, \beta, A \cup \{r(w_i, w_j)\}$

Figure 2: Transitions for Dependency Parsing.

word at the top of the stack. Remove $w_i$ from the stack. (Precondition: the stack needs to contain at least two items and $w_i$ cannot be the ROOT.)

3. RIGHT-ARC$_r$: Add a dependency arc $(w_i, r, w_j)$ to the arc set $A$, where $w_i$ is the word second to the top of the stack and $w_j$ is the word at the top of the stack. Remove $w_j$ from the stack. (Precondition: The stack needs to contain at least two items.)

A more formal definition of these three transitions is presented in Figure 2.

### 1.4 *Neural Dependency Parsing*

While there are many deep models for dependency parsing, this section focuses specifically on greedy, transition-based neural dependency parsers. This class of model has demonstrated comparable performance and significantly better efficiency than traditional feature-based discriminative dependency parsers. The primary distinction from previous models is the reliance on dense rather than sparse feature representations.

The model we will describe employs the arc-standard system for transitions, as presented in section 1.3. Ultimately, the aim of the model is to predict a transition sequence from some initial configuration $c$ to a terminal configuration, in which the dependency parse tree is encoded. As the model is greedy, it attempts to correctly predict one transition $T \in \{\text{SHIFT}, \text{LEFT-ARC}_r, \text{RIGHT-ARC}_r\}$ at a time, based on features extracted from the current configuration $c = (\sigma, \beta, A)$. Recall, $\sigma$ is the stack, $\beta$ the buffer, and $A$ the set of dependency arcs for a given sentence.

**Feature Selection:**

Depending on the desired complexity of the model, there is flexibility in defining the input to the neural network. The features for a given sentence $S$ generally include some subset of:

1. $S_{word}$: Vector representations for some of the words in $S$ (and their dependents) at the top of the stack $\sigma$ and buffer $\beta$.

2. $S_{tag}$: Part-of-Speech (POS) tags for some of the words in $S$. POS tags comprise a small, discrete set: $\mathcal{P} = \{NN, NNP, NNS, DT, JJ, ...\}$

3. $S_{label}$: The arc-labels for some of the words in $S$. The arc-labels comprise a small, discrete set, describing the dependency relation: $\mathcal{L} = \{amod, tmod, nsubj, csubj, dobj, ...\}$

For each feature type, we will have a corresponding embedding matrix, mapping from the feature's one hot encoding, to a $d$-dimensional dense vector representation. The full embedding matrix for $S_{word}$ is $E^w \in \mathbb{R}^{d \times N_w}$ where $N_w$ is the dictionary/vocabulary size. Correspondingly, the POS and label embedding matrices are $E^t \in \mathbb{R}^{d \times N_t}$ and $E^l \in \mathbb{R}^{d \times N_l}$ where $N_t$ and $N_l$ are the number of distinct POS tags and arc labels.

Lastly, let the number of chosen elements from each set of features be denoted as $n_{word}$, $n_{tag}$, and $n_{label}$ respectively.

**Feature Selection Example:**

As an example, consider the following choices for $S_{word}$, $S_{tag}$, and $S_{label}$.

1. $S_{word}$: The top 3 words on the stack and buffer: $s_1, s_2, s_3, b_1, b_2, b_3$. The first and second leftmost / rightmost children of the top two words on the stack: $lc_1(s_i), rc_1(s_i), lc_2(s_i), rc_2(s_i)$, $i = 1, 2$. The leftmost of leftmost / rightmost of rightmost children of the top two words on the stack: $lc_1(lc_1(s_i)), rc_1(rc_1(s_i))$, $i = 1, 2$. In total $S_{word}$ contains $n_w = 18$ elements.
2. $S_{tag}$: The corresponding POS tags for $S_{tag}$ ($n_t = 18$).
3. $S_{label}$: The corresponding arc labels of words, excluding those 6 words on the stack/buffer ($n_l = 12$).

Note that we use a special NULL token for non-existent elements: when the stack and buffer are empty or dependents have not been assigned yet. For a given sentence example, we select the words, POS tags and arc labels given the schematic defined above, extract their corresponding dense feature representations produced from the embedding matrices $E^w$, $E^t$, and $E^l$, and concatenate these vectors into our inputs $[x^w, x^t, x^l]$. At training time we backpropagate into the dense vector representations, as well as the parameters at later layers.

**Feedforward Neural Network Model:**

The network contains an input layer $[x^w, x^t, x^l]$, a hidden layer, and a final softmax layer with a cross-entropy loss function. We can either define a single weight matrix in the hidden layer, to operate on a concatenation of $[x^w, x^t, x^l]$, or we can use three weight matrices $[W_1^w, W_1^t, W_1^l]$, one for each input type, as shown in Figure 3. We then apply a non-linear function and use one more affine layer $[W_2]$ so that there are an equivalent number of softmax probabilities to the number of possible transitions (the output dimension).

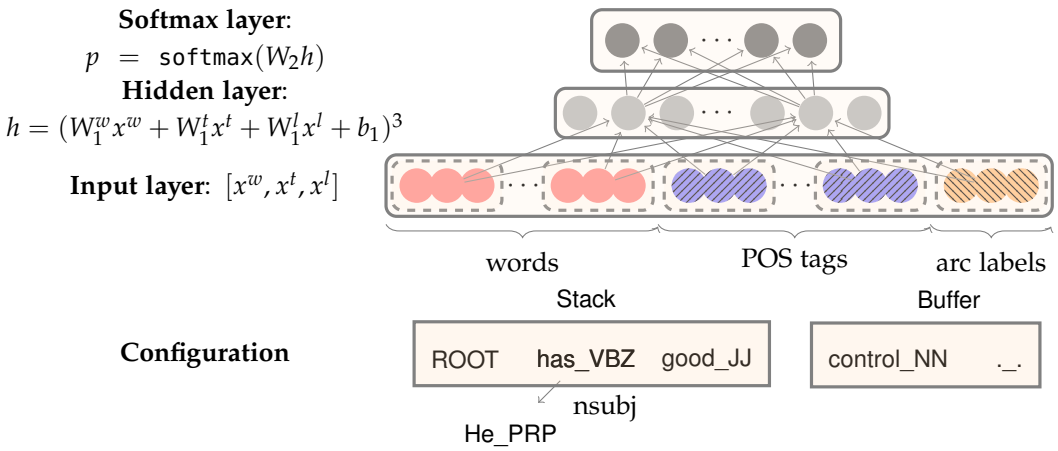



Figure 3: The neural network architecture for greedy, transition-based dependency parsing.

Note that in Figure 3, $f(x) = x^3$ is the non-linear function used.

For a more complete explanation of a greedy transition-based neural dependency parser, refer to "A Fast and Accurate Dependency Parser using Neural Networks" under Further Reading.

**Further reading:**

Danqi Chen, and Christopher D. Manning. "A Fast and Accurate Dependency Parser using Neural Networks." EMNLP. 2014.

Kuebler, Sandra, Ryan McDonald, and Joakim Nivre. "Dependency parsing." Synthesis Lectures on Human Language Technologies 1.1 (2009): 1-127.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part V*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Milad Mohammadi, Rohit Mundra, Richard Socher, Lisa Wang

**Keyphrases: Language Models. RNN. Bi-directional RNN. Deep RNN. GRU. LSTM.**

## 1 Language Models

Language models compute the probability of occurrence of a number of words in a particular sequence. The probability of a sequence of $m$ words $\{w_1, ..., w_m\}$ is denoted as $P(w_1, ..., w_m)$. Since the number of words coming before a word, $w_i$, varies depending on its location in the input document, $P(w_1, ..., w_m)$ is usually conditioned on a window of $n$ previous words rather than all previous words:

$$P(w_1, ..., w_m) = \prod_{i=1}^{i=m} P(w_i|w_1, ..., w_{i-1}) \approx \prod_{i=1}^{i=m} P(w_i|w_{i-n}, ..., w_{i-1}) \quad (1)$$

Equation 1 is especially useful for speech and translation systems when determining whether a word sequence is an accurate translation of an input sentence. In existing language translation systems, for each phrase / sentence translation, the software generates a number of alternative word sequences (e.g. *{I have, I had, I has, me have, me had}*) and scores them to identify the most likely translation sequence.

In machine translation, the model chooses the best word ordering for an input phrase by assigning a *goodness* score to each output word sequence alternative. To do so, the model may choose between different word ordering or word choice alternatives. It would achieve this objective by running all word sequence candidates through a probability function that assigns each a score. The sequence with the highest score is the output of the translation. For example, the machine would give a higher score to *"the cat is small"* compared to *"small the is cat"*, and a higher score to *"walking home after school"* compare do *"walking house after school"*. To compute these probabilities, the count of each n-gram would be compared against the frequency of each word. For instance, if the model takes bi-grams, the frequency of each bi-gram, calculated via combining a word with its previous word, would be divided by the frequency of the corresponding uni-gram. Equations 2 and 3 show this relationship for

bigram and trigram models.

$$p(w_2|w_1) = \frac{count(w_1, w_2)}{count(w_1)} \tag{2}$$

$$p(w_3|w_1, w_2) = \frac{count(w_1, w_2, w_3)}{count(w_1, w_2)} \tag{3}$$

The relationship in Equation 3 focuses on making predictions based on a fixed window of context (i.e. the $n$ previous words) used to predict the next word. In some cases, the window of past consecutive $n$ words may not be sufficient to capture the context. For instance, consider a case where an article discusses the history of Spain and France and somewhere later in the text, it reads "The two countries went on a battle"; clearly the information presented in this sentence alone is not sufficient to identify the name of the two countries. Bengio et al. introduced the first large-scale deep learning for natural language processing model that enables capturing this type of context via *learning a distributed representation of words*; Figure 1 shows the corresponding neural network architecture. In this model, input word vectors are used by both the hidden layer and the output layer. Equation 4 shows the parameters of the softmax() function consisting of the standard tanh() function (i.e. the hidden layer) as well as the linear function, $W^{(3)}x + b^{(3)}$, that captures all the previous $n$ input word vectors.

$$\hat{y} = softmax(W^{(2)}tanh(W^{(1)}x + b^{(1)}) + W^{(3)}x + b^{(3)}) \tag{4}$$

Note that the weight matrix $W^{(1)}$ is applied to the word vectors (solid green arrows in Figure 1), $W^{(2)}$ is applied to the hidden layer (also solid green arrow) and $W^{(3)}$ is applied to the word vectors (dashed green arrows).

In all conventional language models, the memory requirements of the system grows exponentially with the window size $n$ making it nearly impossible to model large word windows without running out of memory.

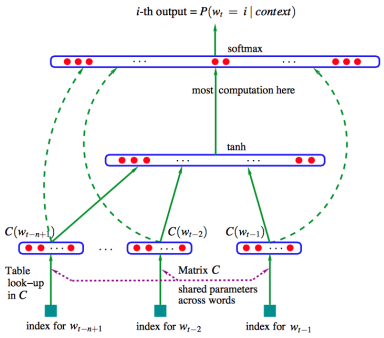



Figure 1: The first deep neural network architecture model for NLP presented by Bengio et al.

## 2 *Recurrent Neural Networks (RNN)*

Unlike the conventional translation models, where only a finite window of previous words would be considered for conditioning the language model, Recurrent Neural Networks (RNN) are capable of conditioning the model on *all* previous words in the corpus.

Figure 2 introduces the RNN architecture where rectangular box is a hidden layer at a time-step, $t$. Each such layer holds a number of neurons, each of which performing a linear matrix operation on

its inputs followed by a non-linear operation (e.g. tanh()). At each time-step, the output of the previous step along with the next word vector in the document, $x_t$, are inputs to the hidden layer to produce a prediction output $\hat{y}$ and output features $h_t$ (Equations 5 and 6). The inputs and outputs of each single neuron are illustrated in Figure 3.

$$h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_{[t]}) \tag{5}$$

$$\hat{y}_t = softmax(W^{(S)}h_t) \tag{6}$$

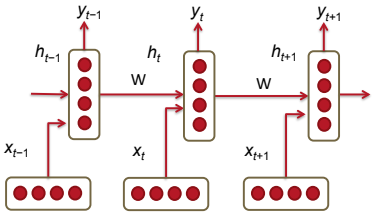



Figure 2: A Recurrent Neural Network (RNN). Three time-steps are shown.

Below are the details associated with each parameter in the network:

- $x_1, ..., x_{t-1}, x_t, x_{t+1}, ...x_T$: the word vectors corresponding to a corpus with T words.
- $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$: the relationship to compute the hidden layer output features at each time-step $t$
  - $x_t \in \mathbb{R}^d$: input word vector at time $t$.
  - $W^{hx} \in \mathbb{R}^{D_h \times d}$: weights matrix used to condition the input word vector, $x_t$
  - $W^{hh} \in \mathbb{R}^{D_h \times D_h}$: weights matrix used to condition the output of the previous time-step, $h_{t-1}$
  - $h_{t-1} \in \mathbb{R}^{D_h}$: output of the non-linear function at the previous time-step, $t-1$. $h_0 \in \mathbb{R}^{D_h}$ is an initialization vector for the hidden layer at time-step $t = 0$.
  - $\sigma()$: the non-linearity function (sigmoid here)
- $\hat{y}_t = softmax(W^{(S)}h_t)$: the output probability distribution over the vocabulary at each time-step $t$. Essentially, $\hat{y}_t$ is the next predicted word given the document context score so far (i.e. $h_{t-1}$) and the last observed word vector $x^{(t)}$. Here, $W^{(S)} \in \mathbb{R}^{|V| \times D_h}$ and $\hat{y} \in \mathbb{R}^{|V|}$ where $|V|$ is the vocabulary.

The loss function used in RNNs is often the cross entropy error introduced in earlier notes. Equation 7 shows this function as the sum over the entire vocabulary at time-step $t$.

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j}) \tag{7}$$

The cross entropy error over a corpus of size $T$ is:

$$J = -\frac{1}{T}\sum_{t=1}^{T} J^{(t)}(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{j=1}^{|V|} y_{t,j} \times log(\hat{y}_{t,j}) \tag{8}$$

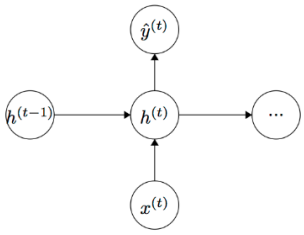



Figure 3: The inputs and outputs to a neuron of a RNN

Equation 9 is called the *perplexity* relationship; it is basically 2 to the power of the negative log probability of the cross entropy error function shown in Equation 8. Perplexity is a measure of confusion where lower values imply more confidence in predicting the next word in the sequence (compared to the ground truth outcome).

$$Perplexity = 2^J \tag{9}$$

The amount of memory required to run a layer of RNN is proportional to the number of words in the corpus. For instance, a sentence with k words would have k word vectors to be stored in memory. Also, the RNN must maintain two pairs of $W, b$ matrices. While the size of W could be very large, it does not scale with the size of the corpus (unlike the traditional language models). For a RNN with 1000 recurrent layers, the matrix would be $1000 \times 1000$ regardless of the corpus size.

Figure 4 is an alternative representation of RNNs used in some publications. It representsRNNs the RNN hidden layer as a loop.

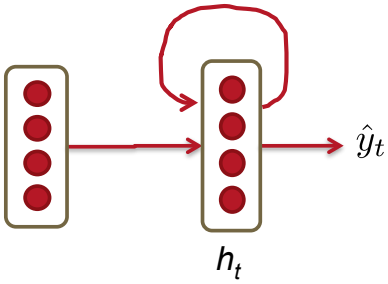



Figure 4: The illustration of a RNN as a loop over time-steps

### 2.1 *Vanishing Gradient & Gradient Explosion Problems*

Recurrent neural networks propagate weight matrices from one time-step to the next. Recall the goal of a RNN implementation is to enable propagating context information through faraway time-steps. For example, consider the following two sentences:

Sentence 1

"Jane walked into the room. John walked in too. Jane said hi to ___"

Sentence 2

"Jane walked into the room. John walked in too. It was late in the day, and everyone was walking home after a long day at work. Jane said hi to ___"

In both sentences, given their context, one can tell the answer to both blank spots is most likely "John". It is important that the RNN predicts the next word as "John", the second person who has appeared several time-steps back in both contexts. Ideally, this should be possible given what we know about RNNs so far. In practice, however, it turns out RNNs are more likely to correctly predict the blank spot in Sentence 1 than in Sentence 2. This is because during the back-propagation phase, the contribution of gradient values gradually vanishes as they propagate to earlier time-steps. Thus, for long sentences, the probability that "John" would be recognized as the next word reduces with the size of

the context. Below, we discuss the mathematical reasoning behind the vanishing gradient problem.

Consider Equations 5 and 6 at a time-step $t$; to compute the RNN error, $dE/dW$, we sum the error at each time-step. That is, $dE_t/dW$ for every time-step, $t$, is computed and accumulated.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \frac{\partial E_t}{\partial W} \tag{10}$$

The error for each time-step is computed through applying the chain rule differentiation to Equations 6 and 5; Equation 11 shows the corresponding differentiation. Notice $dh_t/dh_k$ refers to the partial derivative of $h_t$ with respect to *all* previous $k$ time-steps.

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial W} \tag{11}$$

Equation 12 shows the relationship to compute each $dh_t/dh_k$; this is simply a chain rule differentiation over all hidden layers within the $[k, t]$ time interval.

$$\frac{\partial h_t}{\partial h_k} = \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} W^T \times diag[f'(j_{j-1})] \tag{12}$$

Because $h \in \mathbb{R}^{D_n}$, each $\partial h_j/\partial h_{j-1}$ is the Jacobian matrix for $h$:

$$\frac{\partial h_j}{\partial h_{j-1}} = [\frac{\partial h_j}{\partial h_{j-1,1}} \cdots \frac{\partial h_j}{\partial h_{j-1,D_n}}] = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,1}}{\partial h_{j-1,D_n}} \\ \cdot & \cdot & & & \cdot \\ \cdot & & \cdot & & \cdot \\ \cdot & & & \cdot & \cdot \\ \frac{\partial h_{j,D_n}}{\partial h_{j-1,1}} & \cdot & \cdot & \cdot & \frac{\partial h_{j,D_n}}{\partial h_{j-1,D_n}} \end{bmatrix} \tag{13}$$

Putting Equations 10, 11, 12 together, we have the following relationship.

$$\frac{\partial E}{\partial W} = \sum_{t=1}^{T} \sum_{k=1}^{t} \frac{\partial E_t}{\partial y_t} \frac{\partial y_t}{\partial h_t} (\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}}) \frac{\partial h_k}{\partial W} \tag{14}$$

Equation 15 shows the norm of the Jacobian matrix relationship in Equation 13. Here, $\beta_W$ and $\beta_h$ represent the upper bound values for the two matrix norms. The norm of the partial gradient at each time-step, $t$, is therefore, calculated through the relationship shown in Equation 15.

$$\| \frac{\partial h_j}{\partial h_{j-1}} \| \leq \| W^T \| \| diag[f'(h_{j-1})] \| \leq \beta_W \beta_h \tag{15}$$

The norm of both matrices is calculated through taking their L2-norm. The norm of $f'(h_{j-1})$ can only be as large as 1 given the sigmoid non-linearity function.

$$\| \frac{\partial h_t}{\partial h_k} \| = \| \prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} \| \leq (\beta_W \beta_h)^{t-k} \tag{16}$$

The exponential term $(\beta_W \beta_h)^{t-k}$ can easily become a very small or large number when $\beta_W \beta_h$ is much smaller or larger than 1 and $t - k$ is sufficiently large. Recall a large $t - k$ evaluates the cross entropy error due to faraway words. The contribution of faraway words to predicting the next word at time-step $t$ diminishes when the gradient vanishes early on.

During experimentation, once the gradient value grows extremely large, it causes an overflow (i.e. NaN) which is easily detectable at runtime; this issue is called the *Gradient Explosion Problem*. When the gradient value goes to zero, however, it can go undetected while drastically reducing the learning quality of the model for far-away words in the corpus; this issue is called the *Vanishing Gradient Problem*.

To gain practical intuition about the vanishing gradient problem, you may visit the following example website.

### 2.2 *Solution to the Exploding & Vanishing Gradients*

Now that we gained intuition about the nature of the vanishing gradients problem and how it manifests itself in deep neural networks, let us focus on a simple and practical heuristic to solve these problems.

To solve the problem of exploding gradients, Thomas Mikolov first introduced a simple heuristic solution that *clips* gradients to a small number whenever they explode. That is, whenever they reach a certain threshold, they are set back to a small number as shown in Algorithm 1.

Algorithm 1: Psudo-code for norm clipping in the gradients whenever they explode

$\hat{g} \leftarrow \frac{\partial E}{\partial W}$
**if** $\| \hat{g} \| \geq threshold$ **then**
  $\hat{g} \leftarrow \frac{threshold}{\| \hat{g} \|} \hat{g}$
**end if**

Figure 5 visualizes the effect of gradient clipping. It shows the decision surface of a small recurrent neural network with respect to its $W$ matrix and its bias terms, $b$. The model consists of a single unit of recurrent neural network running through a small number of time-steps; the solid arrows illustrate the training progress on each gradient descent step. When the gradient descent model hits the high error wall

in the objective function, the gradient is pushed off to a far-away location on the decision surface. The clipping model produces the dashed line where it instead pulls back the error gradient to somewhere close to the original gradient landscape.

To solve the problem of vanishing gradients, we introduce two techniques. The first technique is that instead of initializing $W^{(hh)}$ randomly, start off from an identify matrix initialization.

The second technique is to use the Rectified Linear Units (ReLU) instead of the sigmoid function. The derivative for the ReLU is either 0 or 1. This way, gradients would flow through the neurons whose derivative is 1 without getting attenuated while propagating back through time-steps.

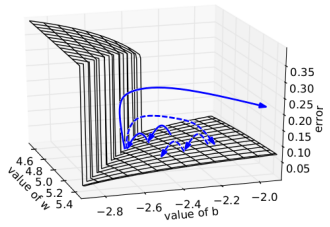



Figure 5: Gradient explosion clipping visualization

### 2.3 *Deep Bidirectional RNNs*

So far, we have focused on RNNs that look into the past words to predict the next word in the sequence. It is possible to make predictions based on future words by having the RNN model read through the corpus backwards. Irsoy et al. shows a bi-directional deep neural network; at each time-step, $t$, this network maintains two hidden layers, one for the left-to-right propagation and another for the right-to-left propagation. To maintain two hidden layers at any time, this network consumes twice as much memory space for its weight and bias parameters. The final classification result, $\hat{y}_t$, is generated through combining the score results produced by both RNN hidden layers. Figure 6 shows the bi-directional network architecture, and Equations 17 and 18 show the mathematical formulation behind setting up the bi-directional RNN hidden layer. The only difference between these two relationships is in the direction of recursing through the corpus. Equation 19 shows the classification relationship used for predicting the next word via summarizing past and future word representations.

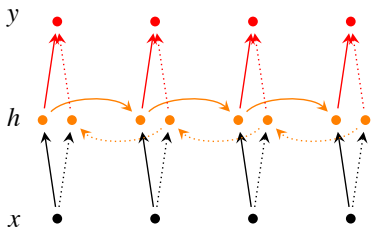



Figure 6: A bi-directional RNN model

$$\overrightarrow{h}_t = f(\overrightarrow{W}x_t + \overrightarrow{V}\overrightarrow{h}_{t-1} + \overrightarrow{b}) \tag{17}$$

$$\overleftarrow{h}_t = f(\overleftarrow{W}x_t + \overleftarrow{V}\overleftarrow{h}_{t+1} + \overleftarrow{b}) \tag{18}$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t; \overleftarrow{h}_t] + c) \tag{19}$$

Figure 7 shows a multi-layer bi-directional RNN where each lower layer feeds the next layer. As shown in this figure, in this network architecture, at time-step $t$ each intermediate neuron receives one set of parameters from the previous time-step (in the same RNN layer), and two sets of parameters from the previous RNN hidden layer; one

input comes from the left-to-right RNN and the other from the right-to-left RNN.

To construct a Deep RNN with $L$ layers, the above relationships are modified to the relationships in Equations 20 and 21 where the input to each intermediate neuron at level $i$ is the output of the RNN at layer $i-1$ at the same time-step, $t$. The output, $\hat{y}$, at each time-step is the result of propagating input parameters through all hidden layers (Equation 22).

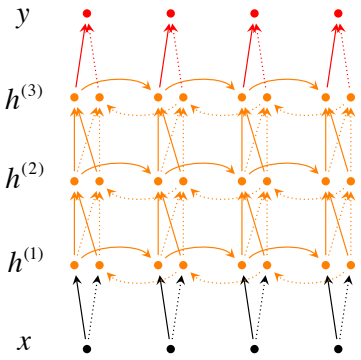



Figure 7: A deep bi-directional RNN with three RNN layers.

$$\overrightarrow{h}_t^{(i)} = f(\overrightarrow{W}^{(i)} h_t^{(i-1)} + \overrightarrow{V}^{(i)} \overrightarrow{h}_{t-1}^{(i)} + \overrightarrow{b}^{(i)}) \tag{20}$$

$$\overleftarrow{h}_t^{(i)} = f(\overleftarrow{W}^{(i)} h_t^{(i-1)} + \overleftarrow{V}^{(i)} \overleftarrow{h}_{t+1}^{(i)} + \overleftarrow{b}^{(i)}) \tag{21}$$

$$\hat{y}_t = g(Uh_t + c) = g(U[\overrightarrow{h}_t^{(L)}; \overleftarrow{h}_t^{(L)}] + c) \tag{22}$$

### 2.4 *Application: RNN Translation Model*

Traditional translation models are quite complex; they consist of numerous machine learning algorithms applied to different stages of the language translation pipeline. In this section, we discuss the potential for adopting RNNs as a replacement to traditional translation modules. Consider the RNN example model shown in Figure 8; here, the German phrase *Echt dicke Kiste* is translated to *Awesome sauce*. The first three hidden layer time-steps *encode* the German language words into some language word features ($h_3$). The last two time-steps *decode* $h_3$ into English word outputs. Equation 23 shows the relationship for the Encoder stage and Equations 24 and 25 show the equation for the Decoder stage.

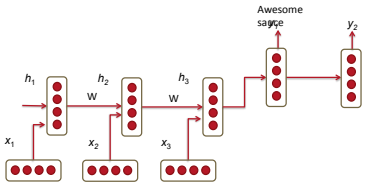



Figure 8: A RNN-based translation model. The first three RNN hidden layers belong to the source language model encoder, and the last two belong to the destination language model decoder.

$$h_t = \phi(h_{t-1}, x_t) = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t) \tag{23}$$

$$h_t = \phi(h_{t-1}) = f(W^{(hh)} h_{t-1}) \tag{24}$$

$$y_t = softmax(W^{(S)} h_t) \tag{25}$$

One may naively assume this RNN model along with the cross-entropy function shown in Equation 26 can produce high-accuracy translation results. In practice, however, several extensions are to be added to the model to improve its translation accuracy performance.

$$max_\theta \frac{1}{N} \sum_{n=1}^{N} \times log(p_\theta(y^{(n)}|x^{(n)})) \tag{26}$$

*Extension I:* train different RNN weights for encoding and decoding. This decouples the two units and allows for more accuracy prediction of each of the two RNN modules. This means the $\phi()$ functions in Equations 23 and 24 would have different $W^{(hh)}$ matrices.

*Extension II:* compute every hidden state in the decoder using three different inputs:

- The previous hidden state (standard)
- Last hidden layer of the encoder ($c = h_T$ in Figure 9)
- Previous predicted output word, $\hat{y}_{t-1}$

Combining the above three inputs transforms the $\phi$ function in the decoder function of Equation 24 to the one in Equation 27. Figure 9 illustrates this model.

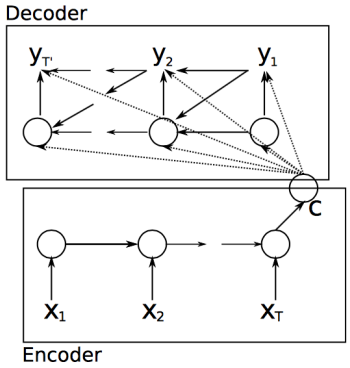



Figure 9: Language model with three inputs to each decoder neuron: $(h_{t-1}, c, y_{t-1})$

$$h_t = \phi(h_{t-1}, c, y_{t-1}) \tag{27}$$

*Extension III:* train deep recurrent neural networks using multiple RNN layers as discussed earlier in this chapter. Deeper layers often improve prediction accuracy due to their higher learning capacity. Of course, this implies a large training corpus must be used to train the model.

*Extension IV:* train bi-directional encoders to improve accuracy similar to what was discussed earlier in this chapter.

*Extension V:* given a word sequence $A\ B\ C$ in German whose translation is $X\ Y$ in English, instead of training the RNN using $A\ B\ C \rightarrow X\ Y$, train it using $C\ B\ A \rightarrow X\ Y$. The intutition behind this technique is that $A$ is more likely to be translated to $X$. Thus, given the vanishing gradient problem discussed earlier, reversing the order of the input words can help reduce the error rate in generating the output phrase.

## 3 *Gated Recurrent Units*

Beyond the extensions discussed so far, RNNs have been found to perform better with the use of more complex units for activation. So far, we have discussed methods that transition from hidden state $h_{t-1}$ to $h_t$ using an affine transformation and a point-wise nonlinearity. Here, we discuss the use of a gated activation function thereby modifying the RNN architecture. What motivates this? Well, although RNNs can theoretically capture long-term dependencies, they are very hard

to actually train to do this. Gated recurrent units are designed in a manner to have more persistent memory thereby making it easier for RNNs to capture long-term dependencies. Let us see mathematically how a GRU uses $h_{t-1}$ and $x_t$ to generate the next hidden state $h_t$. We will then dive into the intuition of this architecture.

$$
\begin{aligned}
z_t &= \sigma(W^{(z)}x_t + U^{(z)}h_{t-1}) && \text{(Update gate)} \\
r_t &= \sigma(W^{(r)}x_t + U^{(r)}h_{t-1}) && \text{(Reset gate)} \\
\tilde{h}_t &= \tanh(r_t \circ Uh_{t-1} + Wx_t) && \text{(New memory)} \\
h_t &= (1 - z_t) \circ \tilde{h}_t + z_t \circ h_{t-1} && \text{(Hidden state)}
\end{aligned}
$$

The above equations can be thought of a GRU's four fundamental operational stages and they have intuitive interpretations that make this model much more intellectually satisfying (see Figure 10):

1. **New memory generation:** A new memory $\tilde{h}_t$ is the consolidation of a new input word $x_t$ with the past hidden state $h_{t-1}$. Anthropomorphically, this stage is the one who knows the recipe of combining a newly observed word with the past hidden state $h_{t-1}$ to summarize this new word in light of the contextual past as the vector $\tilde{h}_t$.

2. **Reset Gate:** The reset signal $r_t$ is responsible for determining how important $h_{t-1}$ is to the summarization $\tilde{h}_t$. The reset gate has the ability to completely diminish past hidden state if it finds that $h_{t-1}$ is irrelevant to the computation of the new memory.

3. **Update Gate:** The update signal $z_t$ is responsible for determining how much of $h_{t-1}$ should be carried forward to the next state. For instance, if $z_t \approx 1$, then $h_{t-1}$ is almost entirely copied out to $h_t$. Conversely, if $z_t \approx 0$, then mostly the new memory $\tilde{h}_t$ is forwarded to the next hidden state.

4. **Hidden state:** The hidden state $h_t$ is finally generated using the past hidden input $h_{t-1}$ and the new memory generated $\tilde{h}_t$ with the advice of the update gate.

It is important to note that to train a GRU, we need to learn all the different parameters: $W, U, W^{(r)}, U^{(r)}, W^{(z)}, U^{(z)}$. These follow the same backpropagation procedure we have seen in the past.

## 4 *Long-Short-Term-Memories*

Long-Short-Term-Memories are another type of complex activation unit that differ a little from GRUs. The motivation for using these is similar to those for GRUs however the architecture of such units does differ.

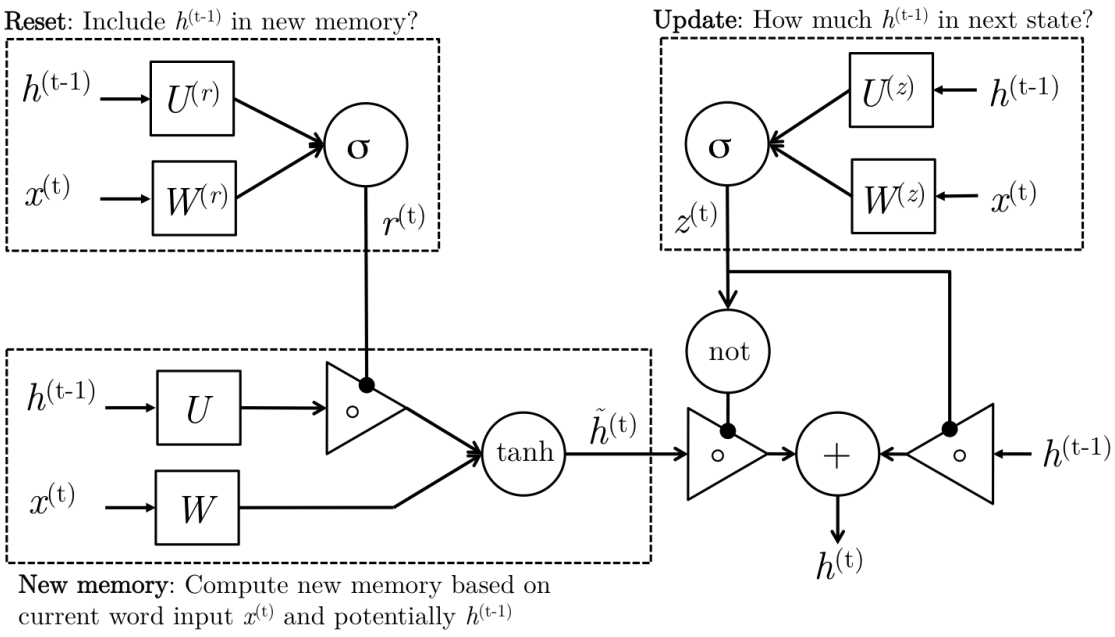



Figure 10: The detailed internals of a GRU

Let us first take a look at the mathematical formulation of LSTM units before diving into the intuition behind this design:

$$
\begin{aligned}
i_t &= \sigma(W^{(i)}x_t + U^{(i)}h_{t-1}) && \text{(Input gate)} \\
f_t &= \sigma(W^{(f)}x_t + U^{(f)}h_{t-1}) && \text{(Forget gate)} \\
o_t &= \sigma(W^{(o)}x_t + U^{(o)}h_{t-1}) && \text{(Output/Exposure gate)} \\
\tilde{c}_t &= \tanh(W^{(c)}x_t + U^{(c)}h_{t-1}) && \text{(New memory cell)} \\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t && \text{(Final memory cell)} \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

We can gain intuition of the structure of an LSTM by thinking of its architecture as the following stages:

1. **New memory generation:** This stage is analogous to the new memory generation stage we saw in GRUs. We essentially use the input word $x_t$ and the past hidden state $h_{t-1}$ to generate a new memory $\tilde{c}_t$ which includes aspects of the new word $x^{(t)}$.

2. **Input Gate:** We see that the new memory generation stage doesn't check if the new word is even important before generating the new memory – this is exactly the input gate's function. The input gate uses the input word and the past hidden state to determine whether or not the input is worth preserving and thus is used to gate the new memory. It thus produces $i_t$ as an indicator of this information.

3. **Forget Gate:** This gate is similar to the input gate except that it does not make a determination of usefulness of the input word –

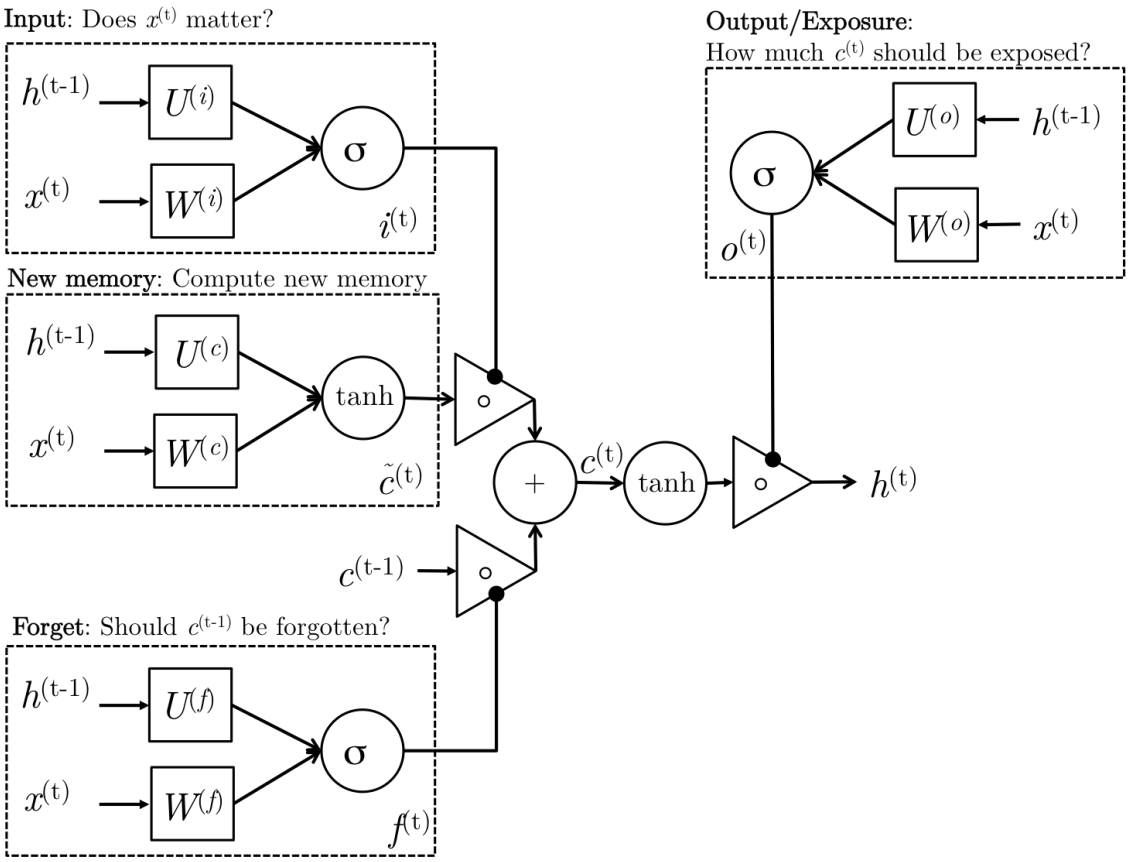



Figure 11: The detailed internals of a LSTM

instead it makes an assessment on whether the past memory cell is useful for the computation of the current memory cell. Thus, the forget gate looks at the input word and the past hidden state and produces $f_t$.

4. **Final memory generation:** This stage first takes the advice of the forget gate $f_t$ and accordingly forgets the past memory $c_{t-1}$. Similarly, it takes the advice of the input gate $i_t$ and accordingly gates the new memory $\tilde{c}_t$. It then sums these two results to produce the final memory $c_t$.

5. **Output/Exposure Gate:** This is a gate that does not explicitly exist in GRUs. It's purpose is to separate the final memory from the hidden state. The final memory $c_t$ contains a lot of information that is not necessarily required to be saved in the hidden state. Hidden states are used in every single gate of an LSTM and thus, this gate makes the assessment regarding what parts of the memory $c_t$ needs to be exposed/present in the hidden state $h_t$. The signal it produces to indicate this is $o_t$ and this is used to gate the point-wise tanh of the memory.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: Part VI*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Guillaume Genthial, Lucas Liu, Barak Oshri, Kushal Ranjan

**Keyphrases: Seq2Seq and Attention Mechanisms, Neural Machine Translation, Speech Processing**

## *1 Neural Machine Translation with Seq2Seq*

So far in this class, we've dealt with problems of predicting a single output: an NER label for a word, the single most likely next word in a sentence given the past few, and so on. However, there's a whole class of NLP tasks that rely on *sequential output*, or outputs that are sequences of potentially varying length. For example,

- **Translation:** taking a sentence in one language as input and outputting the same sentence in another language.
- **Conversation:** taking a statement or question as input and responding to it.
- **Summarization:** taking a large body of text as input and outputting a summary of it.

In these notes, we'll look at sequence-to-sequence models, a deep learning-based framework for handling these types of problems. This framework proved to be very effective, and has, in fewer than 3 years, become the standard for machine translation.

### *1.1 Brief Note on Historical Approaches*

In the past, translation systems were based on probabilistic models constructed from:

- a **translation model**, telling us what a sentence/phrase in a source language most likely translates into
- a **language model**, telling us how likely a given sentence/phrase is overall.

These components were used to build translation systems based on words or phrases. As you might expect, a naive word-based system would completely fail to capture differences in ordering between languages (e.g. where negation words go, location of subject vs. verb in a sentence, etc).

Phrase-based systems were most common prior to Seq2Seq. A phrase-based translation system can consider inputs and outputs in terms of sequences of phrases and can handle more complex syntaxes than word-based systems. However, long-term dependencies are still difficult to capture in phrase-based systems.

The advantage that Seq2Seq brought to the table, especially with its use of LSTMs, is that modern translation systems can generate arbitrary output sequences after seeing the *entire* input. They can even focus in on specific parts of the input automatically to help generate a useful translation.

### 1.2 Sequence-to-sequence Basics

Sequence-to-sequence, or "Seq2Seq", is a relatively new paradigm, with its first published usage in 2014 for English-French translation [3]. At a high level, a sequence-to-sequence model is an end-to-end model made up of two recurrent neural networks:

[3] Sutskever et al. 2014, "Sequence to Sequence Learning with Neural Networks"

- an *encoder*, which takes the model's input sequence as input and encodes it into a fixed-size "context vector", and
- a *decoder*, which uses the context vector from above as a "seed" from which to generate an output sequence.

For this reason, Seq2Seq models are often referred to as "encoder-decoder models." We'll look at the details of these two networks separately.

### 1.3 Seq2Seq architecture - encoder

The encoder network's job is to read the input sequence to our Seq2Seq model and generate a fixed-dimensional context vector $C$ for the sequence. To do so, the encoder will use a recurrent neural network cell – usually an LSTM – to read the input tokens one at a time. The final hidden state of the cell will then become $C$. However, because it's so difficult to compress an arbitrary-length sequence into a single fixed-size vector (especially for difficult tasks like translation), the encoder will usually consist of *stacked* LSTMs: a series of LSTM "layers" where each layer's outputs are the input sequence to the next layer. The *final* layer's LSTM hidden state will be used as $C$.

Seq2Seq encoders will often do something strange: they will process the input sequence *in reverse*. This is actually done on purpose. The idea is that, by doing this, the *last* thing that the encoder sees will (roughly) corresponds to the *first* thing that the model outputs; this makes it easier for the decoder to "get started" on the output, which makes then gives the decoder an easier time generating a

proper output sentence. In the context of translation, we're allowing the network to translate the first few words of the input as soon as it sees them; once it has the first few words translated correctly, it's much easier to go on to construct a correct sentence than it is to do so from scratch. See Fig. 1 for an example of what such an encoder network might look like.

### 1.4 Seq2Seq architecture - decoder

The decoder is also an LSTM network, but its usage is a little more complex than the encoder network. Essentially, we'd like to use it as a language model that's "aware" of the words that it's generated so far *and* of the input. To that end, we'll keep the "stacked" LSTM architecture from the encoder, but we'll initialize the hidden state of our first layer with the context vector from above; the decoder will literally use the context of the input to generate an output.

Once the decoder is set up with its context, we'll pass in a special token to signify the start of output generation; in literature, this is usually an <EOS> token appended to the end of the input (there's also one at the end of the output). Then, we'll run all three layers of LSTM, one after the other, following up with a softmax on the final layer's output to generate the first output word. Then, we *pass that word into the first layer*, and repeat the generation. This is how we get the LSTMs to act like a language model. See Fig. 2 for an example of a decoder network.

Once we have the output sequence, we use the same learning strategy as usual. We define a loss, the cross entropy on the prediction sequence, and we minimize it with a gradient descent algorithm and back-propagation. Both the encoder and decoder are trained at the same time, so that they both learn the same context vector representation.

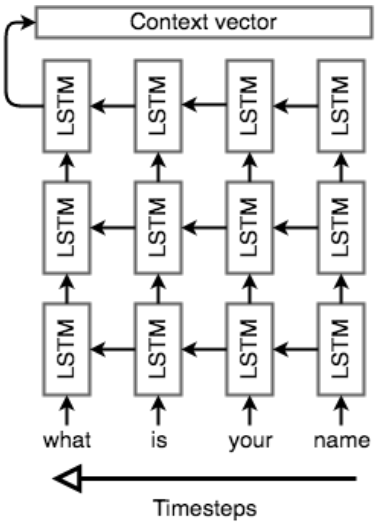



Figure 1: Example of a Seq2Seq encoder network. This model may be used to translate the English sentence "what is your name?" Note that the input tokens are read in reverse. Note that the network is unrolled; each column is a timestep and each row is a single layer, so that horizontal arrows correspond to hidden states and vertical arrows are LSTM inputs/outputs.

### 1.5 Recap & Basic NMT Example

Note that there is no connection between the lengths of the input and output; any length input can be passed in and any length output can be generated. However, Seq2Seq models are known to lose effectiveness on very long inputs, a consequence of the practical limits of LSTMs.

To recap, let's think about what a Seq2Seq model does in order to translate the English "what is your name" into the French "comment t'appelles tu". First, we start with 4 one-hot vectors for the input. These inputs may or may not (for translation, they usually are) embedded into a dense vector representation. Then, a stacked LSTM network reads the sequence in reverse and *encodes* it into a context

vector. This context vector is a vector space representation of the notion of asking someone for their name. It's used to initialize the first layer of another stacked LSTM. We run one step of each layer of this network, perform softmax on the last layer's output, and use that to select our first output word. This word is fed back into the network as input, and the rest of the sentence "comment t'appelles tu" is *decoded* in this fashion. During backpropagation, the encoder's LSTM weights are updated so that it learns a better vector space representation for sentences, while the decoder's LSTM weights are trained to allow it to generate gramatically correct sentences that are relevant to the context vector.

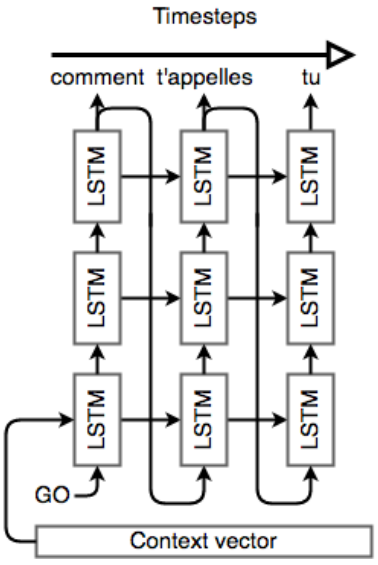



Figure 2: Example of a Seq2Seq decoder network. This decoder is decoding the context vector for "what is your name" (see Fig. 1 into its French translation, "comment t'appeles tu?" Note the special "GO" token used at the start of generation, and that generation is in the forward direction as opposed to the input which is read in reverse. Note also that the input and output do not need to be the same length.

### 1.6 *Bidirectional RNNs*

Recall from earlier in this class that dependencies in sentences don't just work in one direction; a word can have a dependency on another word before *or* after it. The formulation of Seq2Seq that we've talked about so far doesn't account for that; at any timestep, we're only considering information (via the LSTM hidden state) from words *before* the current word. For NMT, we need to be able to effectively encode any input, regardless of dependency directions within that input, so this won't cut it.

Bidirectional RNNs fix this problem by traversing a sequence in both directions and concatenating the resulting outputs (both cell outputs and final hidden states). For every RNN cell, we simply add another cell but feed inputs to it in the opposite direction; the output $o_t$ corresponding to the $t$'th word is the concatenated vector $\begin{bmatrix} o_t^{(f)} & o_t^{(b)} \end{bmatrix}$, where $o_t^{(f)}$ is the output of the forward-direction RNN on word $t$ and $o_t^{(b)}$ is the corresponding output from the reverse-direction RNN. Similarly, the final hidden state is $h = \begin{bmatrix} h^{(f)} & h^{(b)} \end{bmatrix}$, where $h^{(f)}$ is the final hidden state of the forward RNN and $h^{(b)}$ is the final hidden state of the reverse RNN. See Fig. 6 for an example of a bidirectional LSTM encoder.

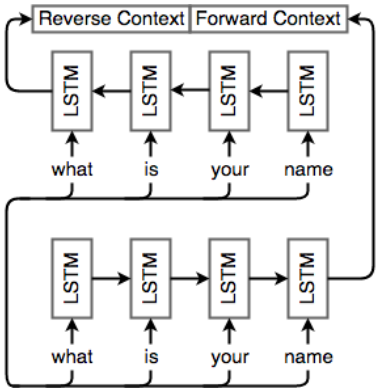



Figure 3: Example of a single-layer bidirectional LSTM encoder network. Note that the input is fed into two different LSTM layers, but in different directions, and the hidden states are concatenated to get the final context vector.

## 2 *Attention Mechanism*

### 2.1 *Motivation*

When you hear the sentence "the ball is on the field," you don't assign the same importance to all 6 words. You primarily take note of the words "ball," "on," and "field," since those are the words that are most "important" to you. Similarly, Bahdanau et al. noticed the flaw in using the final RNN hidden state as the single "context vector" for sequence-to-sequence models: often, different parts of an input have

different levels of significance. Moreover, different parts of the output may even consider different parts of the input "important." For example, in translation, the first word of output is *usually* based on the first few words of the input, but the last word is likely based on the last few words of input.

Attention mechanisms make use of this observation by providing the decoder network with a look at the *entire input sequence* at every decoding step; the decoder can then decide what input words are important at any point in time. There are many types of encoder mechanisms, but we'll examine the one introduced by Bahdanau et al. [4],

[4] Bahdanau et al. 2014, "Neural Machine Translation by Jointly Learning to Align and Translate"

### 2.2 *Bahdanau et al. NMT model*

Remember that our seq2seq model is made of two parts, an **encoder** that encodes the input sentence, and a **decoder** that leverages the information extracted by the decoder to produce the translated sentence. Basically, our input is a sequence of words $x_1, \ldots, x_n$ that we want to translate, and our target sentence is a sequence of words $y_1, \ldots, y_m$.

1. **Encoder**

   Let $(h_1, \ldots, h_n)$ be the hidden vectors representing the input sentence. These vectors are the output of a bi-LSTM for instance, and capture contextual representation of each word in the sentence.

2. **Decoder**

   We want to compute the hidden states $s_i$ of the decoder using a recursive formula of the form

   $$s_i = f(s_{i-1}, y_{i-1}, c_i)$$

   where $s_{i-1}$ is the previous hidden vector, $y_{i-1}$ is the generated word at the previous step, and $c_i$ is a context vector that capture the context from the original sentence that is relevant to the time step $i$ of the decoder.

   The context vector $c_i$ captures relevant information for the $i$-th decoding time step (unlike the standard Seq2Seq in which there's only one context vector). For each hidden vector from the original sentence $h_j$, compute a score

   $$e_{i,j} = a(s_{i-1}, h_j)$$

   where $a$ is any function with values in $\mathbb{R}$, for instance a single layer fully-connected neural network. Then, we end up with a

sequence of scalar values $e_{i,1}, \ldots, e_{i,n}$. Normalize these scores into a vector $\alpha_i = (\alpha_{i,1}, \ldots, \alpha_{i,n})$, using a *softmax* layer.

The vector $\alpha_i$ is called the *attention* vector

$$\alpha_{i,j} = \frac{exp(e_{i,j})}{\sum_{k=1}^{n} exp(e_{i,k})}$$

Then, compute the context vector $c_i$ as the weighted average of the hidden vectors from the original sentence

The context vector is extracted thanks to the attention vector and captures the relevant context

$$c_i = \sum_{j=1}^{n} \alpha_{i,j} h_j$$

Intuitively, this vector captures the relevant contextual information from the original sentence for the $i$-th step of the decoder.

### 2.3 *Connection with translation alignment*

The attention-based model learns to assign significance to different parts of the input for each step of the output. In the context of translation, attention can be thought of as "alignment." Bahdanau et al. argue that the attention scores $\alpha_{ij}$ at decoding step $i$ signify the words in the source sentence that align with word $i$ in the target. Noting this, we can use attention scores to build an alignment table – a table mapping words in the source to corresponding words in the target sentence – based on the learned encoder and decoder from our Seq2Seq NMT system.

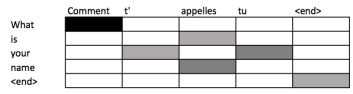



Figure 4: Example of an alignment table

### 2.4 *Performance on long sentences*

The major advantage of attention-based models is their ability to efficiently translate long sentences. As the size of the input grows, models that do not use attention will miss information and precision if they only use the final representation. Attention is a clever way to fix this issue and experiments indeed confirm the intuition.

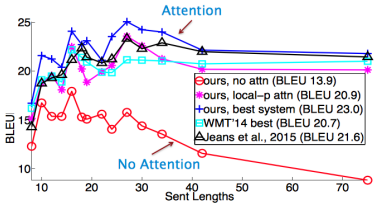



Figure 5: Performance on long sentence of different NMT models - image taken from Luong et al.

## 3 *Other Models*

### 3.1 *Huong et al. NMT model*

We present a variant of this first model, with two different mechanisms of attention, from Luong et al.[5].

- **Global attention** We run our vanilla Seq2Seq NMT. We call the hidden states given by the encoder $h_1, \ldots, h_n$, and the hidden states of the decoder $\bar{h_1}, \ldots, \bar{h_n}$. Now, for each $\bar{h_i}$, we compute an

[5] *Effective Approaches to Attention-based Neural Machine Translation* by Minh-Thang Luong, Hieu Pham and Christopher D. Manning

attention vector over the encoder hidden. We can use one of the following scoring functions:

$$score(h_i, \bar{h}_j) = \begin{cases} h_i^T \bar{h}_j \\ h_i^T W \bar{h}_j & \in \mathbb{R} \\ W[h_i, \bar{h}_j] \end{cases}$$

Now that we have a vector of scores, we can compute a context vector in the same way as Bahdanau et al. First, we normalize the scores via a softmax layer to obtain a vector $\alpha_i = (\alpha_{i,1}, \ldots, \alpha_{i,n})$, where $\alpha_{i,j} = \frac{exp(score(h_j, \bar{h}_i))}{\sum_{k=1}^{n} exp(score(h_k, \bar{h}_i))}$

$$c_i = \sum_{j=1}^{n} \alpha_{i,j} h_j$$

and we can use the context vector and the hidden state to compute a new vector for the $i$-th time step of the decoder

$$\tilde{h}_i = f([\bar{h}_i, c_i])$$

The final step is to use the $\tilde{h}_i$ to make the final prediction of the decoder. To address the issue of coverage, Luong et al. also use an input-feeding approach. The attentional vectors $\tilde{h}_i$ are fed as input to the decoder, instead of the final prediction. This is similar to Bahdanau et al., who use the context vectors to compute the hidden vectors of the decoder.

- **Local attention** the model predicts an aligned position in the input sequence. Then, it computes a context vector using a window centered on this position. The computational cost of this attention step is constant and does not explode with the length of the sentence.

The main takeaway of this discussion is to show that they are lots of ways of doing attention.

### 3.2 *Google's new NMT*

As a brief aside, Google recently made a major breakthrough for NMT via their own translation system [6]. Rather than maintain a full Seq2Seq model for every pair of language that they support – each of which would have to be trained individually, which is a tremendous feat in terms of both data and compute time required – they built a single system that can translate between any two languages. This is a Seq2Seq model that accepts as input a sequence of words *and* a token

[6] Johnson et el. 2016, "Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation"

specifying what language to translate into. The model uses shared parameters to translate into any target language.

The new multilingual model not only improved their translation performance, it also enabled "zero-shot translation," in which we can translate between two languages *for which we have no translation training data*. For instance, if we only had examples of Japanese-English translations and Korean-English translations, Google's team found that the multilingual NMT system trained on this data could actually generate reasonable Japanese-Korean translations. The powerful implication of this finding is that part of the decoding process is not language-specific, and the model is in fact maintaining an internal representation of the input/output sentences independent of the actual languages involved.

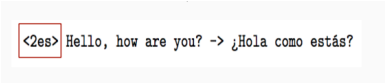



Figure 6: Example of Google's system

### 3.3 More advanced papers using attention

- *Show, Attend and Tell: Neural Image Caption Generation with Visual Attention* by Kelvin Xu, Jimmy Lei Ba,Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhutdinov, Richard S. Zemel and Yoshua Bengio. This paper learns words/image alignment.
- *Modeling Coverage for Neural Machine Translation* by Zhaopeng Tu, Zhengdong Lu, Yang Liu, Xiaohua Liu and Hang Li. Their model uses a coverage vector that takes into account the attention history to help future attention.
- *Incorporating Structural Alignment Biases into an Attentional Neural Translation Model* by Cohn, Hoang, Vymolova, Yao, Dyer, Haffari. This paper improves the attention by incorporating other traditional linguistic ideas.

## 4 Sequence model decoders

Another approach to machine translation comes from statistical machine translation. Consider a model that computes the probability $\mathbb{P}(\bar{s}|s)$ of a translation $\bar{s}$ given the original sentence $s$. We want to pick the translation $\bar{s}*$ that has the best probability. In other words, we want

$$\bar{s}* = \text{argmax}_{\bar{s}}(\mathbb{P}(\bar{s}|s))$$

As the search space can be huge, we need to shrink its size. Here is a list of sequence model decoders (both good ones and bad ones).

- **Exhaustive search** : this is the simplest idea. We compute the probability of every possible sequence, and we chose the sequence

with the highest probability. However, this technique does not scale *at all* to large outputs as the search space is exponential in the size of the input. Decoding in this case is an NP-complete problem.

- **Ancestral sampling** : at time step $t$, we sample $x_t$ based on the conditional probability of the word at step $t$ given the past. In other words,

$$x_t \sim \mathbb{P}(x_t|x_1, \ldots, x_n)$$

Theoretically, this technique is efficient and asymptotically exact. However, in practice, it can have low performance and high variance.

- **Greedy Search** : At each time step, we pick the most probable token. In other words

$$x_t = \text{argmax}_{\tilde{x_t}} \mathbb{P}(\tilde{x_t}|x_1, \ldots, x_n)$$

This technique is efficient and natural, however it explores a small part of the search space and if we make a mistake at one time step, the rest of the sentence could be heavily impacted.

- **Beam search** : the idea is to maintain $K$ candidates at each time step.

$$\mathcal{H}_t = \{(x_1^1, \ldots, x_t^1), \ldots, (x_1^K, \ldots, x_t^K)\}$$

and compute $\mathcal{H}_{t+1}$ by expanding $\mathcal{H}_t$ and keeping the best $K$ candidates. In other words, we pick the best $K$ sequence in the following set

$$\tilde{\mathcal{H}}_{t+1} = \bigcup_{k=1}^{K} \mathcal{H}_{t+1}^{\tilde{k}}$$

where

$$\mathcal{H}_{t+1}^{\tilde{k}} = \{(x_1^k, \ldots, x_t^k, v_1), \ldots, (x_1^k, \ldots, x_t^k, v_{|V|})\}$$

As we increase $K$, we gain precision and we are asymptotically exact. However, the improvement is not monotonic and we can set a $K$ that combines reasonable performance and computational efficiency. For this reason, beam search is the most commonly used technique in NMT.

## 5 *Evaluation of Machine Translation Systems*

Now that we know the basics about machine translation systems, we discuss some ways that these models are evaluated. Evaluating the quality of translations is a notoriously tricky and subjective task. In real-life, if you give a paragraph of text to ten different translators, you will get back ten different translations. Translations are imperfect and noisy in practice. They attend to different information and emphasize different meanings. One translation can preserve metaphors and the integrity of long-randing ideas, while the other can achieve a more faithful reconstruction of syntax and style, attempting a word-to-word translation. Note that this flexibility is not a burden; it is a testament to the complexity of language and our abilities to decode and interpret meaning, and is a wonderful aspect of our communicative faculty.

At this point, you should note that there is a difference between the objective *loss* function of your model and the *evaluation* methods we are going to discuss. Since loss functions are in essence an evaluation of your model prediction, it can be easy to confuse the two ideas. The evaluation metrics ahead offer a final, summative assessment of your model against some measurement criterion, and no one measurement is superior to all others, though some have clear advantages and majority preference.

Evaluating the quality of machine learning translations has become it own entire research area, with many proposals like TER, METEOR, MaxSim, SEPIA, and RTE-MT. We will focus in these notes on two baseline evaluation methods and BLEU.

### 5.1 *Human Evaluation*

The first and maybe least surprising method is to have people manually evaluate the correctness, adequacy, and fluency of your system. Like the Turing Test, if you can fool a human into not being able to distinguish a human-made translation with your system translation, your model passes the test for looking like a real-life sentence! The obvious problem with this method is that it is costly and inefficient, though it remains the gold standard for machine translation.

### 5.2 *Evaluation against another task*

A common way of evaluating machine learning models that output a useful *representation* of some data (a representation being a translation or summary) is that if your predictions are useful for solving some challenging task, then the model must be encoding relevant information in your predictions. For example, you might think of

training your translation predictions on a question-answering task in the translated language. That is, you use the outputs of your system as inputs to a model for some other task (the question-answering). If your second task can perform as well on your predictions as it can on well-formed data in the translated language, it means that your inputs have the relevant information or patterns for meeting the demands of the task.

The issue with this method is that the second task may not be affected by many of the finer points of translation. For example, if you measured the quality of translation on a query-retrieval task (like pulling up the right webpage for a search query), you would find that a translation that preserves the main topic words of the documents but ignores syntax and grammar might still fit the task well. But this itself doesn't mean that the quality of your translations is accurate or faithful. Therefore, determining the quality of the translation model is just shifted to determining the quality of the task itself, which may or may not be a good standard.

### 5.3 *Bilingual Evaluation Understudy (BLEU)*

In 2002, IBM researchers developed the Bilingual Evaluation Understudy (BLEU) that remains, with its many variants to this day, one of the most respected and reliable methods for machine translation.

The BLEU algorithm evaluates the precision score of a candidate machine translation against a reference human translation. The reference human translation is a assumed to be a *model* example of a translation, and we use n-gram matches as our metric for how similar a candidate translation is to it. Consider a reference sentence A and candidate translation B:

A there are many ways to evaluate the quality of a translation, like comparing the number of n-grams between a candidate translation and reference.

B the quality of a translation is evaluate of n-grams in a reference and with translation.

The BLEU score looks for whether n-grams in the machine translation also appear in the reference translation. Color-coded below are some examples of different size n-grams that are shared between the reference and candidate translation.

A there are many ways to evaluate the quality of a translation, like comparing the number of n-grams between a candidate translation and reference.

B the quality of a translation is evaluate of n-grams in a reference and with translation.

The BLEU algorithm identifies all such matches of n-grams above, including the unigram matches, and evaluates the strength of the match with the *precision* score. The precision score is the fraction of n-grams in the translation that also appear in the reference.

The algorithm also satisfies two other constraints. For each n-gram size, a gram in the reference translation cannot be matched more than once. For example, the unigram "a" appears twice in B but only once in A. This only counts for one match between the sentences. Additionally, we impose a brevity penalty so that very small sentences that would achieve a 1.0 precision (a "perfect" matching) are not considered good translations. For example, the single word "there" would achieve a 1.0 precision match, but it is obviously not a good match.

Let us see how to actually compute the BLEU score. First let $k$ be the maximum n-gram that we want to evaluate our score on. That is, if $k = 4$, the BLUE score only counts the number of n-grams with length less than or equal to 4, and ignores larger n-grams. Let

$$p_n = \text{\# matched n-grams / \# n-grams in candidate translation}$$

the precision score for the grams of length $n$. Finally, let $w_n = 1/2^n$ be a geometric weighting for the precision of the $n$'th gram. Our brevity penalty is defined as

$$\beta = e^{\min(0, 1 - \frac{\text{len}_{\text{ref}}}{\text{len}_{\text{MT}}})}$$

where $\text{len}_{\text{ref}}$ is the length of the reference translation and $\text{len}_{\text{MT}}$ is the length of the machine translation.

The BLEU score is then defined as

$$\text{BLEU} = \beta \prod_{i=1}^{k} p_n^{w_n}$$

The BLEU score has been reported to correlate well with human judgment of good translations, and so remains a benchmark for all evaluation metrics following it. However, it does have many limitations. It only works well on the corpus level because any zeros in precision scores will zero the entire BLEU score. Additionally, this BLEU score as presented suffers for only comparing a candidate translation against a single reference, which is surely a noisy representation of the relevant n-grams that need to be matched. Variants of BLEU have modified the algorithm to compare the candidate with multiple reference examples. Additionally, BLEU scores may only be

a necessary but not sufficient benchmark to pass for a good machine translation system. Many researchers have optimized BLEU scores until they have begun to approach the same BLEU scores between reference translations, but the true quality remains far below human translations.

## 6 Dealing with the large output vocabulary

Despite the success of modern NMT systems, they have a hard time dealing with large vocabulary size. Specifically, these Seq2Seq models predict the next word in the sequence by computing a target probabilistic distribution over the entire vocabulary using *softmax*. It turns out that **softmax** can be quite expensive to compute with a large vocabulary and its complexity also scales proportionally to the vocabulary size. We will now examine a number of approaches to address this issue.

### 6.1 Scaling softmax

A very natural idea is to ask "can we find more efficient ways to compute the target probabilistic distribution?" The answer is Yes! In fact, we've already learned two methods that can reduce the complexity of "softmax", which we'll present a high-level review below (see details in lecture note 1).

1. **Noise Contrastive Estimation**

   The idea of NCE is to approximate "softmax" by randomly sampling K words from negative samples. As a result, we are reducing the computational complexity by a factor of $\frac{|V|}{K}$, where $|V|$ is the vocabulary size. This method has been proven sucessful in word2vec. A recent work by Zoph et al.[7] applied this technique to learning LSTM language models and they also introduced a trick by using the same samples per mini-batch to make the training GPU-efficient.

2. **Hierarchical Softmax**

   Morin et al.[8] introduced a binary tree structure to more efficently compute "softmax". Each probability in the target distribution is calculated by taking a path down the tree which only takes $O(log|V|)$ steps. Notably, even though Hierarchical Softmax saves computation, it cannot be easily parallelized to run efficiently on GPU. This method is used by Kim et al. [9] to train character-based language models which will be covered in lecture 13.

One limitation for both methods is that they only save computation during training step (when target word is known). At test time,

[7] Zoph et al. 2016, *Simple, Fast Noise-Contrastive Estimation for Large RNN Vocabularies*

[8] Morin et al. 2005, *Hierarchical Probabilistic Neural Network Language Model*

[9] Kim et al. 2015, *Character-Aware Neural Language Models*

one still has to compute the probability of all words in the vocabulary in order to make predictions.

### 6.2 *Reducing vocabulary*

Instead of optimizing "softmax", one can also try to reduce the effective vocabulary size which will speed up both training and test steps. A naive way of doing this is to simply limit the vocabulary size to a small number and replace words outside the vocabulary with a tag <UNK>. Now, both training and test time can be significantly reduced but this is obviously not ideal because we may generate outputs with lots of <UNK>.

Jean et al. [10] proposed a method to maintain a constant vocabulary size $|V'|$ by partitioning the training data into subsets with $\tau$ unique target words, where $\tau = |V'|$. One subset can be found by sequentially scanning the original data set until $\tau$ unique target words are detected( Figure 7). And this process is iterated over the entire data set to produce all mini-batch subsets. In practice, we can achieve about 10x saving with $|V| = 500K$ and $|V'| = 30K, 50K$.

[10] Jean et al. 2015, *On Using Very Large Target Vocabulary for Neural Machine Translation*

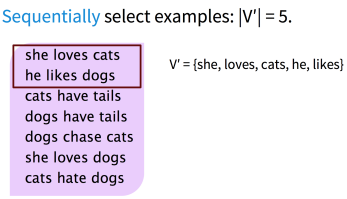



Figure 7: Training data partition

This concept is very similar to NCE in that for any given word, the output vocabulary contains the target word and $|V'| - 1$ negative (noise) samples. However, the main difference is that these negative samples are sampled from a biased distribution $Q$ for each subset V′ where

$$Q(y_t) = \begin{cases} \frac{1}{|V'|}, if y_t \in |V'| \\ 0, otherwise \end{cases}$$

At test time, one can similarly predict target word out of a selected subset, called *candidate list*, of the entire vocabulary. The challenge is that the correct target word is unknown and we have to "guess" what the target word might be. In the paper, the authors proposed to construct a candidate list for each source sentence using K most frequent words (based on unigram probability) and K′ likely target words for each source word in the sentence. In Figure 8), an example is shown with $K' = 3$ and the candidate list consists of all the words in purple boxes. In practice, one can choose the following values: $K = 15k, 30k, 50k$ and $K' = 10, 20$.

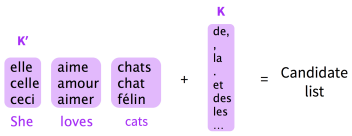



Figure 8: Candidate list

### 6.3 *Handling unknown words*

When NMT systems use the techniques mentioned above to reduce effective vocabulary size, inevitably, certain words will get mapped to <UNK>. For instance, this could happen when the predicted word, usually rare word, is out of the candidate list or when we encounter

unseen words at test time. We need new mechanisms to address the rare and unknown word problems.

One idea introduced by Gulcehre et al.[11] to deal with these problems is to learn to "copy" from source text. The model (Figure 9) applies attention distribution $l_t$ to decide *where* to point in the source text and uses the decoder hidden state $S_t$ to predict a binary variable $Z_t$ which decides *when* to copy from source text. The final prediction is either the word $y_t^w$ chosen by softmax over candidate list, as in previous methods, or $y_t^l$ copied from source text depending on the value of $Z_t$. They showed that this method improves performance in tasks like machine translation and text summarization.

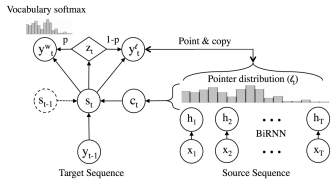



Figure 9: Pointer network Architecture

[11] Gulcehre et al. 2016, *Pointing the Unknown Words*

As one can imagine, there are of course limitations to this method. It is important to point out a comment from Google's NMT paper[12] on this method, " *this approach is both unreliable at scale — the attention mechanism is unstable when the network is deep — and copying may not always be the best strategy for rare words — sometimes transliteration is more appropriate*".

[12] Wu et al. 2016, *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*

## 7 *Word and character-based models*

As discussed in section 6, "copy" mechanisms are still not sufficient in dealing with rare or unknown words. Another direction to address these problems is to operate at sub-word levels. One trend is to use the same seq2seq architecture but operate on a smaller unit — word segmentation, character-based models. Another trend is to embrace hybrid architectures for words and characters.

### 7.1 *Word segmentation*

Sennrich et al. [13] proposed a method to enable open-vocabulary translation by representing rare and unknown words as a sequence of subword units.

This is achieved by adapting a compression algorithm called **Byte Pair Encoding**. The essential idea is to start with a vocabulary of characters and keep extending the vocabulary with most frequent n-gram pairs in the data set. For instance, in Figure 10, our data set contains 4 words with their frequencies on the left, i.e. "low" appears 5 times. Denote $(p, q, f)$ as a n-gram pair p, q with frequency f. In this figure, we've already selected most frequent n-gram pair (e,s,9) and now we are adding current most frequent n-gram pair (es,t,9). This process is repeated until all n-gram pairs are selected or vocabulary size reaches some threshold.

One can choose to either build separate vocabularies for training and test sets or build one vocabulary jointly. After the vocabulary

[13] Sennrich et al. 2016, *Neural Machine Translation of Rare Words with Subword Units*

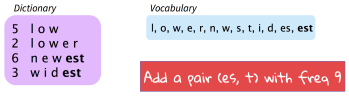



Figure 10: Byte Pair Encoding

is built, an NMT system with some seq2seq architecture (the paper used Bahdanau et al. [14]), can be directly trained on these word segments. Notably, this method won top places in WMT 2016.

[14] Bahdanau et al. 2014, "Neural Machine Translation by Jointly Learning to Align and Translate"

### 7.2 *Character-based model*

Ling et al. [15] proposed a character-based model to enable open-vocabulary word representation.

[15] Ling, et al. 2015, "Finding Function in Form: Compositional Character Models for Open Vocabulary Word Representation"

For each word $w$ with $m$ characters, instead of storing a word embedding, this model iterates over all characters $c_1, c_2 \dots c_m$ to look up the character embeddings $e_1, e_2 \dots e_m$. These character embeddings are then fed into a biLSTM to get the final hidden states $h_f, h_b$ for forward and backward directions respectively. The final word embedding is computed by an affine transformation of two hidden states:

$$e_w = W_f H_f + W_b H_b + b$$

There are also a family of CNN character-based models which will be covered in lecture 13.

### 7.3 *Hybrid NMT*

Luong et al. [16] proposed a Hybrid Word-Character model to deal with unknown words and achieve open-vocabulary NMT. The system translates mostly at word-level and consults the character components for rare words. On a high level, the character-level recurrent neural networks compute source word representations and recover unknown target words when needed. The twofold advantage of such a hybrid approach is that it is much faster and easier to train than character-based ones; at the same time, it never produces unknown words as in the case of word-based models.

[16] Luong et al. 2016, *Achieving Open Vocabulary Neural Machine Translation with Hybrid Word-Character Models*

**Word-based Translation as a Backbone** The core of the hybrid NMT is a deep LSTM encoder-decoder that translates at the word level. We maintain a vocabulary of size $|V|$ per language and use <*unk*> to represent out of vocabulary words.

**Source Character-based Representation** In regular word-based NMT, a universal embedding for <*unk*> is used to represent all out-of-vocabulary words. This is problematic because it discards valuable information about the source words. Instead, we learn a deep LSTM model over characters of rare words, and use the final hidden state of the LSTM as the representation for the rare word (Figure 11).

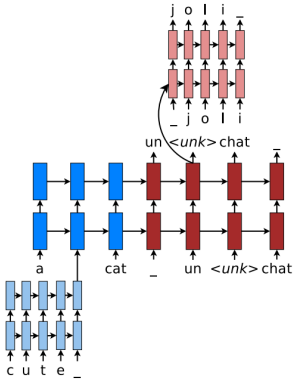



Figure 11: Hybrid NMT

**Target Character-level Generation** General word-based NMT allows generation of <*unk*> in the target output. Instead, the goal here is to create a coherent framework that handles an unlimited output vocabulary. The solution is to have a separate deep LSTM that

"translates" at the character level given the current word-level state. Note that the current word context is used to initialize the character-level encoder. The system is trained such that whenever the word-level NMT produces an <*unk*>, the character-level decoder is asked to recover the correct surface form of the unknown target word.

# *CS224n: Deep Learning for NLP*[1]

*Lecture Notes: Part VII* [2]

*Winter 2017*

[1] Course Instructor: Richard Socher

[2] Authors: Francois Chaubard, Richard Socher

**Keyphrases: RNN, Recursive Neural Networks, MV-RNN, RNTN**

This set of notes discusses and describes the many variants on the RNN (Recursive Neural Networks) and their application and successes in the field of NLP.

## 1 *Recursive Neural Networks*

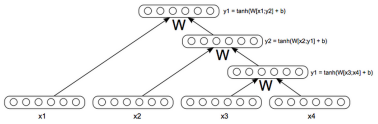



Figure 1: A standard Recursive Neural Network

In these notes, we introduce and discuss a new type of model that is indeed a superset of the previously discussed Recurrent Neural Network. Recursive Neural Networks (RNNs) are perfect for settings that have nested hierarchy and an intrinsic recursive structure. Well if we think about a sentence, doesn't this have such a structure? Take the sentence "A small crowd quietly enters the historical church". First, we break apart the sentence into its respective Noun Phrase, Verb Phrase, "A small crowd" and "quietly enters the historical church", respectively. But there is a noun phrase, verb phrase within that verb phrase right? "quietly enters" and "historical church". etc, etc. Seems pretty recursive to me.

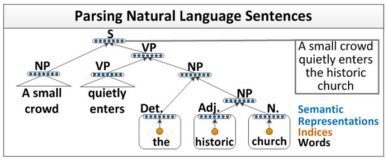



Figure 2: Parse Tree of a sentence.

The syntactic rules of language are highly recursive. So we take advantage of that recursive structure with a model that respects it! Another added benefit of modeling sentences with RNN's is that we can now input sentences of arbitrary length, which was a huge head scratcher for using Neural Nets in NLP, with very clever tricks to make the input vector of the sentence to be of equal size despite the length of the sentences not being equal. (see Bengio et al., 2003; Henderson, 2003; Collobert & Weston, 2008)

Let's imagine our task is to take a sentence and represent it as a vector in the same semantic space as the words themselves. So that phrases like "I went to the mall yesterday", "We went shopping last week", and "They went to the store", would all be pretty close in distance to each other. Well we have seen ways to train unigram word vectors, should we do the same for bigrams, trigrams, etc. This very well may work but there are two major issues with this thinking. 1) There are literally an infinite amount of possible combinations of words. Storing and training an infinite amount of vectors would just be absurd. 2) Some combinations of words while they might be completely reasonable to hear in language, may never be represented in our training/dev corpus. So we would never learn them.

We need a way to take a sentence and its respective words vectors,

and derive what the embedded vector should be. Now lets first ask a very debated question. Is it naive to believe that the vector space that we used to represent all words, is sufficiently expressive to also be able to represent all sentences of any length? While this may be unintuitive, the performance of these models suggest that this is actually a reasonable thing to do.

Let's first discuss the difference between semantic and grammatical understanding of a sentence. Semantic analysis is an understanding of the meaning of a sentence, being able to represent the phrase as a vector in a structured semantic space, where similar sentences are very nearby, and unrelated sentences are very far away. The grammatical understanding is one where we have identified the underlying grammatical structure of the sentence, which part of the sentence depends on which other part, what words are modifying what other words, etc. The output of such an understanding is usually represented as a parse tree as displayed in Figure 2.

Now for the million dollar question. If we want to know the semantic representation, is it an advantage, nay, required, to have a grammatical understanding? Well some might disagree but for now we will treat this semantic composition task the following way. First, we need to understand words. Then, we need to know the way words are put together, Then, finally, we can get to a meaning of a phrase or sentence by leveraging these two previous concepts.

So lets begin with our first model built on this principle. Let's imagine we were given a sentence, and we knew the parse tree for that sentence, such as the one displayed in Figure 2, could we figure out an encoding for the sentence and also perhaps a sentiment score just from the word vectors that are in the sentence? We observe how a Simple RNN can perform this task.

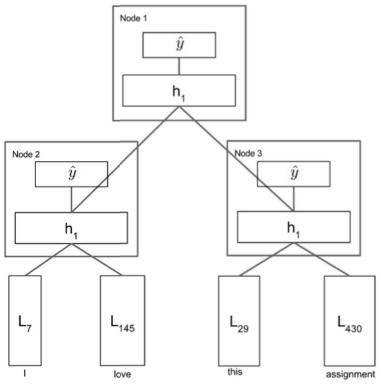


Figure 3: An example standard RNN applied to a parsed sentence "I love this assignment"

### 1.1 *A simple single layer RNN*

Lets walk through the model displayed in Figure 3 above. We first take a sentence parse tree and the sentence word vectors and begin to walk up the tree. The lowest node in the graph is Node 3, so we concatenate $L_{29}$ and $L_{430}$ to form a vector $\in \mathbb{R}^{2d}$ and feed it into our network to compute:

$$h^{(1)} = tanh(W^{(1)} \begin{bmatrix} L_{29} \\ L_{430} \end{bmatrix} + b^{(1)}) \tag{1}$$

Since $W^{(1)} \in \mathbb{R}^{d \times 2d}$ and $b^{(1)} \in \mathbb{R}^d$, $h^{(1)} \in \mathbb{R}^d$. We can now think of $h^{(1)}$ as a point in the same word vector space for the bigram "this assignment", in which we did not need to learn this representation separately, but rather derived it from its constituting word vectors.

We now take $h^{(1)}$ and put it through a softmax layer to get a score over a set of sentiment classes, a discrete set of known classes that represent some meaning. In the case of positive/negative sentiment analysis, we would have 5 classes, class 0 implies strongly negative, class 1 implies negative, class 2 is neutral, class 3 is positive, and finally class 4 is strongly positive.

Now we do the same thing with the "I" and "love" to produce the vector $h^{(1)}$ for the phrase "I love". Again, we compute a score over the semantic classes again for that phrase. Finally, for the most interesting step, we need to merge the two phrases "I love" and "this assignment". Here we are concatenating word phrases, rather than word vectors! We do this in the same manner, concatenating the two $h^{(1)}$ vectors and compute

$$h^{(1)} = \tanh(W^{(1)} \begin{bmatrix} h^{(1)}_{Left} \\ h^{(1)}_{Right} \end{bmatrix} + b^{(1)}) \tag{2}$$

Now we have a vector in the word vector space that represents the full sentence "I love this assignment". Furthermore, we can put this $h^{(1)}$ through the same softmax layer as before, and compute sentiment probabilities for the full sentence. Of course the model will only do this reliably once trained.

Now lets take a step back. First, is it naive to think we can use the same matrix W to concatenate all words together and get a very expressive $h^{(1)}$ and yet again use that same matrix W to concatenate all phrase vectors to get even deeper phrases? These criticisms are valid and we can address them in the following twist on the simple RNN.

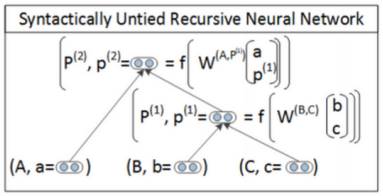



Figure 4: Using different W's for different categories of inputs is more natural than having just one W for all categories

### 1.2 *Syntactically Untied SU-RNN*

As we discussed in the criticisms of the previous section, using the same W to bring together a Noun Phrase and Verb Phrase and to bring together a Prepositional Phrase and another word vector seems intuitively wrong. And maybe we are bluntly merging all of these functionalities into too weak of a model.

What we can do to remedy this shortcoming is to "syntactically untie" the weights of these different tasks. By this we mean, there is no reason to expect the optimal W for one category of inputs to be at all related to the optimal W for another category of inputs. So we let these W's be different and relax this constraint. While this for sure increases our weight matrices to learn, the performance boost we gain is non-trivial.

As in FIgure 4 above, we notice our model is now conditioned upon what the syntactic categories of the inputs are. Note, we deter-

mine what the categories are via a very simple Probabilistic Context Free Grammar (PCFG) which is more or less learned by computing summary statistics over the Penn Tree Bank to learn rules such as "The" is always a DT, etc, etc. No deeper understanding of this part is really necessary, just know itś really simple.

The only major other difference in this model is that we initialize the W's to the identity. This way the default thing to do is to average the two word vectors coming in. Slowly but surely, the model learns which vector is more important and also any rotation or scaling of the vectors that improve performance. We observe in Figure 5 that the trained weight matrices learn actual meaning! For example, the DT-NP rule or Determiner followed by a Noun Phrase such as "The cat" or "A man", puts more emphasis on the Noun Phrase than on the Determiner. (This is obvious because the right diagonals are red meaning higher weights). This is called the notion of soft head words, which is something that Linguists have long observed to be true for sometime, however the model learned this on its own just by looking at data. Pretty cool!

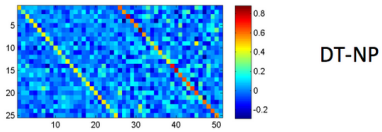


Figure 5: The learnt W weights for DT-NP composition match Linguists theory

The SU-RNN does indeed outperform previously discussed models but perhaps it is still not expressive enough. If we think of modifying words, such as adverbs like "very", any interpolation with this word vector and the following one, is definitely not what the understood nature of "very" is.

As an adverb, it's literal definition is "used for emphasis". How can we have a vector that emphasizes any other vector that is to follow when we are solely performing a linear interpolation? How can we construct a vector that will "scale" any other vector this way? Truth is we can not. We need to have some form of multiplication of word on another word. We uncover two such compositions below that enable this. The first utilizes word matrices and the other utilizes a Quadratic equation over the typical Affine.

### 1.3 *MV-RNN's (Matrix-Vector Recursive Neural Networks)*

We now augment our word representation, to not only include a word vector, but also a word matrix! So the word "very" will have a word vector $v_{very} \in \mathbb{R}^d$ but also $V_{very} \in \mathbb{R}^{d \times d}$. This gives us the expressive ability to not only embed what a word means, but we also learn the way that words "modify" other words. The word matrix enables the latter. To feed two words, $a$ and $b$, into a RNN, we take their word matrices $A$ and $B$, to form our input vector x as the concatenation of vector $Ab$ and $Ba$. For our example of "very", $V_{very}$ could just be the identity times any scalar above one. Which would scale any neighboring word vector by that number! This is the type of expres-

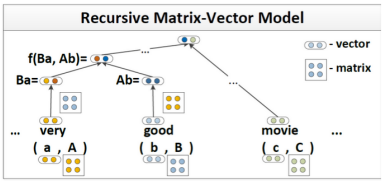


Figure 6: An example MV-RNN

sive ability we desired. While the new word representation explodes our feature space, we can express much better the way words modify each other.

By observing the errors the model makes, we see even the MV-RNN still can not express certain relations. We observe three major classes of mistakes.

First, Negated Positives. When we say something positive but one word turns it negative, the model can not weigh that one word strong enough to flip the sentiment of the entire sentence. Figure 7 shows such an example where the swap of the word "most" to "least" should flip the entire sentiment of the sentence, but the MV-RNN does not capture this successfully.

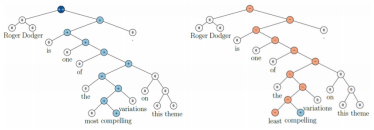


Figure 7: Negated Positives

The second class of mistakes is the Negated Negative case. Where we say something is not bad, or not dull, as in Figure 8. The MV-RNN can not recognize that the word "not" lessens the sentiment from negative to neutral.

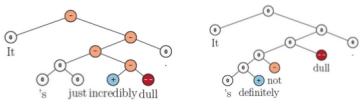


Figure 8: Negated Negatives

The final class of errors we observe is the "X but Y conjunction" displayed in Figure 9. Here the X might be negative BUT if the Y is positive then the model's sentiment output for the sentence should be positive! MV-RNNs struggle with this.

Thus, we must look for an even more expressive composition algorithm that will be able to fully capture these types of high level compositions.

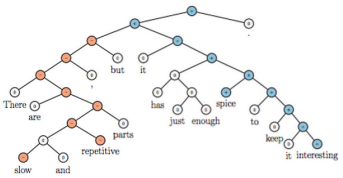


Figure 9: Using a Recursive Neural Net can correctly classify the sentiment of the contrastive conjunction X but Y but the MV-RNN can not

### 1.4 *RNTNs (Recursive Neural Tensor Network)*

The final RNN we will cover here is by far the most successful on the three types of errors we left off with. The Recursive Neural Tensor Network does away with the notion of a word matrix, and furthermore, does away with the traditional affine transformation pre-tanh/sigmoid concept. To compose two word vectors or phrase vectors, we again concatenate them to form a vector $\in \mathbb{R}^{2d}$ but instead of putting it through an affine function then a nonlinear, we put it through a quadratic first, then a nonlinear, such as:

$$h^{(1)} = tanh(x^T V x + Wx) \tag{3}$$

Note that $V$ is a 3rd order tensor in $\in \mathbb{R}^{2d \times 2d \times d}$. We compute $x^T V[i]x \; \forall i \in [1, 2, ...d]$ slices of the tensor outputting a vector $\in \mathbb{R}^d$. We then add $Wx$ and put it through a nonlinear function. The quadratic shows that we can indeed allow for the multiplicative type of interaction between the word vectors without needing to maintain and learn word matrices!

As we see in Figure 11, the RNTN is the only model that is capable of succeeding on these very hard datasets.

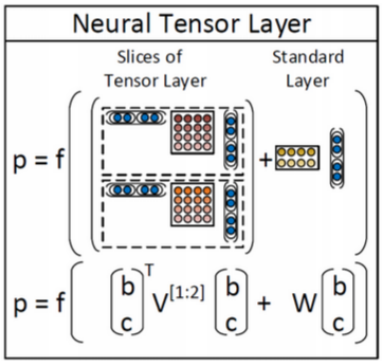



Figure 10: One slice of a RNTN. Note there would be d of these slices.

| Model | Accuracy | |
|---|---|---|
| | Negated Positive | Negated Negative |
| biNB | 19.0 | 27.3 |
| RNN | 33.3 | 45.5 |
| MV-RNN | 52.4 | 54.6 |
| RNTN | **71.4** | **81.8** |

Figure 11: Comparing performance on the Negated Positive and Negated Negative data sets.

We will continue next time with a model that actually outperforms the RNTN in some aspects and it does not require an input parse tree! This model is the Dynamic Convolutional Neural Network, and we will talk about that soon.

## 2 CNNs (Convolutional Neural Networks)

Back to sentence representation, what if you did NOT know the parse tree? The RecNNs we have observed in this set of lecture notes depend on such an initial parsing. What if we are using Recurrent Networks and the context of the phrase is on its right side? If we only applied Softmax on the last time-step, then the last few words would have a disproportionately large impact on the output, e.g.Sentiment Classification.

### 2.1 Why CNNs?

Convolutional Neural Networks take in a sentence of word vectors and first create a phrase vector for all subphrases, not just grammatically correct phrases (as with Recursive Neural Network)! And then, CNNs group them together for the task at hand.

### 2.2 What is Convolution?

Let's start with the 1D case. Consider two 1D vectors, $f$ and $g$ with $f$ being our primary vector and $g$ corresponding to the **filter**. The convolution between $f$ and $g$, evaluated at entry $n$ is represented as $(f * g)[n]$ and is equal to $\sum_{m=-M}^{M} f[n-m]g[m]$.

Figure 12 shows the 2D convolution case. The $9 \times 9$ green matrix represents the primary matrix of concern, $f$. The $3 \times 3$ matrix of red numbers represents the filter $g$ and the convolution currently being evaluated is at position $[2, 2]$. Figure 12 shows the value of the convolution at position $[2, 2]$ as 4 in the second table. Can you complete the second table?

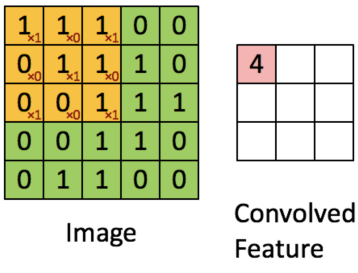



Figure 12: Convolution in the 2D case

### 2.3 A Single-Layer CNN

Consider word-vectors $x_i \in R^k$ and the concatenated word-vectors of a n-word sentence, $x_{1:n} = x_1 \oplus x_2 ... \oplus x_n$. Finally, consider a Convolutional filter $w \in R^{hk}$ i.e. over $h$ words. For $k = 2$, $n = 5$ and $h = 3$, Figure 13 shows the Single-Layer Convolutional layer for NLP. We will get a single value for each possible combination of three consecutive words in the sentence, "the country of my birth". Note, the filter $w$ is itself a vector and we will have $c_i = f(w^T x_{i:i|h-1} + b)$

to give $\mathbf{c} = [c_1, c_2 ... c_{n-h+1}] \in R^{n-h+1}$. For the last two time-steps, i.e. starting with the words "my" or "birth", we don't have enough word-vectors to multiply with the filter (since $h = 3$). If we necessarily need the convolutions associated with the last two word-vectors, a common trick is to pad the sentence with $h-1$ zero-vectors at its right-hand-side as in Figure 14.

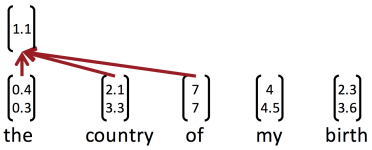



Figure 13: Single-Layer Convolution: one-step

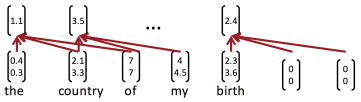



Figure 14: Single-Layer Convolution: all-steps

### 2.4 *Pooling*

Assuming that we don't use zero-padding, we will get a final convolutional output, $\mathbf{c}$ which has $n-h+1$ numbers. Typically, we want to take the outputs of the CNN and feed it as input to further layers like a Feedforward Neural Network or a RecNN. But, all of those need a fixed length input while our CNN output has a length dependent on the length of the sentence, $n$. One clever way to fix this problem is to use max-pooling. The output of the CNN, $\mathbf{c} \in \mathbb{R}^{n-h-1}$ is the input to the max-pooling layer. The output of the max-pooling layer is $\hat{c} = max\{\mathbf{c}\}$, thus $\hat{c} \in \mathbb{R}$.

We could also have used min-pooling because typically we use ReLU as our non-linear activation function and ReLU is bounded on the low side to 0. Hence a min-pool layer might get smothered by ReLU, so we nearly always use max-pooling over min-pooling.

### 2.5 *Multiple-Filters*

In the example above related to Figure 13, we had $h = 2$, meaning we looked only at bi-gram with a single specific combination method i.e. filter. We can use multiple bi-gram filters because each filter will learn to recognize a different kind of bi-gram. Even more generally, we are not restricted to using just bi-grams, we can also have filters using tri-grams, quad-grams and even higher lengths. Each filter has an associated max-pool layer. Thus, our final output from the CNN layers will be a vector having length equal to the number of filters.

### 2.6 *Multiple-Channels*

If we allow gradients to flow into the word-vectors being used here, then the word-vectors might change significantly over training. This is desirable, as it specializes the word-vectors to the specific task at hand (away from say GloVe initialization). But, what about words that appear only in the test set but not in the train set? While other semantically related word vectors which appear in the train set will have moved significantly from their starting point, such words will still be at their initialization point. The neural network will be specialized for inputs which have been updated. Hence, we will get low

performance on sentences with such words (words that are in test but not in train).

One work-around is to maintain two sets of word-vectors, one 'static' (no gradient flow into them) and one 'dynamic', which are updated via SGD. Both are initially the same (GloVe or other initialization). Both sets are simultaneously used as input to the neural network. Thus, the initialized word-vectors will always play a role in the training of the neural network. Giving unseen words present in the test set a better chance of being interpreted correctly.

There are several ways of handling these two channels, most common is to simply average them before using in a CNN. The other method is to double the length of the CNN filters.

### 2.7 *CNN Options*

1. Narrow vs Wide
   Refer to Figure 15. Another way to ask this is should we not (narrow) or should we (wide) zero-pad? If we use Narrow Convolution, we compute a convolution only in those positions where all the components of a filter have a matching input component. This will clearly not be the case at the start and end boundaries of the input, as in the left side network in Figure 15. If we use Wide Convolution, we have an output component corresponding to each alignment of the convolution filter. For this, we will have to pad the input at the start and the end with $h-1$ zeros.
   In the Narrow Convolution case, the output length will be $n-h+1$ and in the Wide Convolution case, the length will be $n+h-1$.

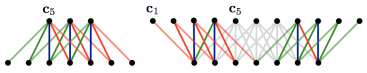


Figure 15: Narrow and Wide Convolution (from Kalchbrenner et al. (2014))

2. $k$-max pooling
   This is a generalization of the max pooling layer. Instead of picking out only the biggest (max) value from its input, the k-max pooling layer picks out the $k$ biggest values. Setting $k=1$ gives the max pooling layer we saw earlier.

## 3 *Constituency Parsing*

Natural Language Understanding requires being able to extract meaning from large text units from the understanding of smaller parts. This extraction requires being able to understand how smaller parts are put together. There are two main techniques to analyze the syntactic structure of sentences: **constituency parsing** and **dependency parsing**. Dependency parsing was covered in the previous lectures (see *lecture notes 4*). By building binary asymmetric relations between a word and its dependents, the structure shows which word

depends on which other words. Now we focus on constituency parsing, that organizes words into nested constituents.

Constituency Parsing is a way to break a piece of text (e.g. one sentence) into sub-phrases. One of the goals of constituency parsing (also known as "phrase structure parsing") is to identify the constituents in the text which would be useful when extracting information from text. By knowing the constituents after parsing the sentence, it is possible to generate similar sentences syntactically correct.

### *3.1 Constituent*

In syntactic analysis, a constituent can be a single word or a phrases as a single unit within a hierarchical structure. A phrase is a sequence of two or more words built around a head lexical item and working as a unit within a sentence. To be a phrase, a group of words should come together to play a specific role in the sentence. In addition, the group of words can be moved together or replaced as a whole, and the sentence should remain fluent and grammatical.

For instance, the following sentence contains the noun phrase: "wonderful CS224N".

We interpret large text units by **semantic composition** of smaller elements. These smaller elements can be changed while keeping a similar meaning, like in the following examples.

- I want to be enrolled in the wonderful CS224N!

We can rewrite the sentence by moving that whole phrase to the front as below.

- The wonderful CS224N I want to be enrolled in!

Or the phrase could be replaced with an constituent of similar function and meaning, like "great CS course in Stanford about NLP and Deep Learning".

- I want to be enrolled in the great CS course in Stanford about NLP and Deep!

For constituency parsing, the basic clause structure is understood as a binary division of the clause into subject (noun phrase NP) and predicate (verb phrase VP), expressed as following rule. The binary division of the clause results in a one-to-one-or-more correspondence. For each element in a sentence, there are one or more nodes in the tree structure.

- S -> NP VP

In fact, the process of parsing illustrates certain similar rules. We deduce the rules by beginning with the sentence symbol S, and applying the phrase structure rules successively, finally applying replacement rules to substitute actual words for the abstract symbols.

Based on the extracted rules, it is possible to generate similar sentences. If the rules are correct, then any sentence produced in this way should be syntactically correct. However, the generated sentences might be syntactically correct but semantically nonsensical, such as the following well-known example:

- Colorless green ideas sleep furiously

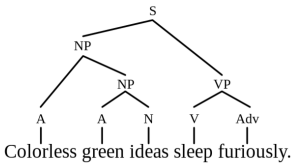



Figure 16: Constituency Parse Tree for 'Colorless green ideas sleep furiously'

### *3.2 Constituency Parse Tree*

Interestingly, in natural language, the constituents are likely to be nested inside one another. Thus a natural representation of these phrases is a tree. Usually we use a consistency parse tree to display the parsing process. The constituency-based parse trees of constituency grammars distinguish between terminal and non-terminal nodes. Non-terminals in the tree are labeled as types of phrases (e.g. Noun Phrase), the terminals are the exact words in the sentence. Take 'John hits the ball' as an example, the syntactic structure of the English sentence is showed as below.

We have a parse tree starting from root S, which represents the whole sentence, and ending in each leaf node, which represents each word in a sentence. We use the following abbreviation:

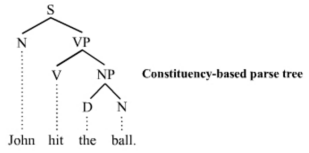



Figure 17: Consistency Parse Tree for 'John hits the ball'

- S stands for sentence, the top-level structure.
- NP stands for noun phrase including the subject of the sentence and the object of the sentence.
- VP stands for verb phrase, which serves as the predicate.
- V stands for verb.
- D stands for determiner, such as the definite article "the"
- N stands for noun

# *CS224n: Deep Learning for NLP*[1]

*Lecture Notes: Part VIII* [2]

*Winter 2017*

**Keyphrases: Coreference Resolution, Dynamic Memory Networks for Question Answering over Text and Images**

[1] Course Instructor: Richard Socher

[2] Authors:

## 1 *Dynamic Memory Networks for Question Answering over Text and Images*

The idea of a QA system is to extract information (sometimes passages, or spans of words) directly from documents, conversations, online searches, etc., that will meet user's information needs. Rather than make the user read through an entire document, QA system prefers to give a short and concise answer. Nowadays, a QA system can combine very easily with other NLP systems like chatbots, and some QA systems even go beyond the search of text documents and can extract information from a collection of pictures.

There are many types of questions, and the simplest of them is factoid question answering. It contains questions that look like "The symbol for mercuric oxide is?" "Which NFL team represented the AFC at Super Bowl 50?". There are of course other types such as mathematical questions ("2+3=?"), logical questions that require extensive reasoning (and no background information). However, we can argue that the information-seeking factoid questions are the most common questions in people's daily life.

In fact, most of the NLP problems can be considered as a question-answering problem, the paradigm is simple: we issue a query, and the machine provides a response. By reading through a document, or a set of instructions, an intelligent system should be able to answer a wide variety of questions. We can ask the POS tags of a sentence, we can ask the system to respond in a different language. So naturally, we would like to design a model that can be used for general QA.

In order to achieve this goal, we face two major obstacles. Many NLP tasks use different architectures, such as TreeLSTM (Tai et al., 2015) for sentiment analysis, Memory Network (Weston et al., 2015) for question answering, and Bi-directional LSTM-CRF (Huang et al., 2015) for part-of-speech tagging. The second problem is full multi-task learning tends to be very difficult, and transfer-learning remains to be a major obstacle for current neural network architectures across artificial intelligence domains (computer vision, reinforcement learning, etc.).

We can tackle the first problem with a shared architecture for NLP: Dynamic Memory Network (DMN), an architecture designed for

general QA tasks. QA is difficult, partially because reading a long paragraph is difficult. Even for humans, we are not able to store a long document in your working memory.

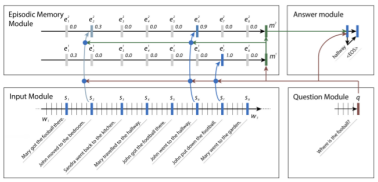



Figure 1: A graphical illustration of the Dynamic Memory Network.

### 1.1 Input Module

Dynamic Memory Network is divided into modules. First we look at input module. The input module takes as input a sequence of $T_I$ words and outputs a sequence of $T_C$ fact representations. If the output is a list of words, we have $T_C = T_I$ and if the output is a list of sentences, we have $T_C$ as the number of sentences and $T_I$ as the number of words in the sentences. We use a simple GRU to read the sentences in, i.e. the hidden state $h_t = \text{GRU}(x_t, h_{t-1})$ where $x_t = L[w_t]$, where $L$ is the embedding matrix and $w_t$ is the word at time $t$. We further improve it by using Bi-GRU as shown in Figure 2.

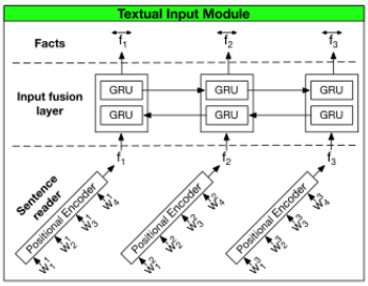



Figure 2: A graphical illustration of the Dynamic Memory Network.

### 1.2 Question Module

We also use a standard GRU to read in the question (using embedding matrix $L$: $q_t = \text{GRU}(L[w_t^Q], q_{t-1})$), but the output of the question module is an encoded representation of the question.

### 1.3 Episodic Memory Module

One of the distinctive features of the dynamic memory network is the episodic memory module which runs over the input sequence multiple times, each time paying attention to a different subset of facts from the input.

It accomplishes this using a Bi-GRU that takes input of the sentence-level representation passed in from the input module, and produces an episodic memory representation.

We denote the episodic memory representation as $m^i$ and the episode representation (output by the attention mechanism) as $e^i$. The episodic memory representation is initialized using $m^0 = q$, and proceeds using the GRU: $m^i = \text{GRU}(e^i, m^{i-1})$. The episode representation is updated using the hidden state outputs from the input module as follows where $g$ is the attention mechanism:

$$h_t^i = g_t^i \text{GRU}(c_t, h_{t-1}^i) + (1 - g_t^i) h_{t-1}^i$$
$$e_i = h_{T_C}^i$$

The attention vector $g$ may be computed in a number of ways, but in the original DMN paper (Kumar et al. 2016), the following formulation was found to work best:

$$g_t^i = G(c_t, m^{i-1}, q)$$

$$G(c, m, q) = \sigma(W^{(2)} \tanh(W^{(1)} z(c, m, q) + b^{(1)}) + b^{(2)})$$
$$z(c, m, q) = [c, m, q, c \circ q, c \circ m, |c - q|, |c - m|, c^T W^{(b)} q, c^T W^{(b)} m]$$

In this way, gates in this module are activated if the sentence is relevant to the question or memory. In the $i$th pass, if the summary is not sufficient to answer the question, we can repeat sequence over input in the $(i + 1)$th pass. For example, consider the question "Where is the football?" and input sequences "John kicked the football" and "John was in the field." In this example, *John* and *football* could be linked in one pass and then *John* and *field* could be linked in the second pass, allowing for the network to perform a transitive inference based on the two pieces of information.

### 1.4 Answer Module

The answer module is a simple GRU decoder that takes in the output of question module, and episodic memory module, and output a word (or in general a computational result). It works as follows:

$$y_t = \text{softmax}(W^{(a)} a_t)$$
$$a_t = \text{GRU}([y_{t-1}, q], a_{t-1})$$

### 1.5 Experiments

Through the experiments we can see DMN is able to outperform MemNN in babl question answering tasks, and it can outperform other architectures for sentiment analysis and part-of-speech tagging. How many episodes are needed in the episodic memory? The answer is that the harder the task is, the more passes are required. Multiple passes also allows the network to truly comprehend the sentence by paying attention to only relevant parts for the final task, instead of reacting to just the information from the word embedding.

The key idea is to modularize the system, and you can allow different types of input by change the input module. For example, if we replace the input module with a convolutional neural network-based module, then this architecture can handle a task called visual question answering (VQA). It is also able to outperform other models in this task.

### 1.6 Summary

The zeal to search for a general architecture that would solve all problems has slightly faded since 2015, but the desire to train on one domain and generalize to other domains has increased. To comprehend more advanced modules for question answering, readers can refer to the dynamic coattention network (DCN).

# Computing Neural Network Gradients

Kevin Clark

## 1 Introduction

The purpose of these notes is to demonstrate how to quickly compute neural network gradients. This will hopefully help you with question 3 of Assignment 2 (if you haven't already done it) and with the midterm (which will have at least one significant gradient computation question). It is **not** meant to provide an intuition for how backpropagation works – for that I recommend going over lecture 5[1] and the cs231 course notes[2] on backpropagation.

## 2 Vectorized Gradients

While it is a good exercise to compute the gradient of a neural network with respect to a single parameter (e.g., a single element in a weight matrix), in practice this tends to be quite slow. Instead, it is more efficient to keep everything in matrix/vector form. The basic building block of vectorized gradients is the *Jacobian Matrix*. Suppose we have a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ that maps a vector of length $n$ to a vector of length $m$: $\boldsymbol{f}(\boldsymbol{x}) = [f_1(x_1, ..., x_n), f_2(x_1, ..., x_n), ..., f_m(x_1, ..., x_n)]$. Then its Jacobian is.

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

That is, $(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial f_i}{\partial x_j}$ (which is just a standard non-vector derivative). The Jacobian matrix will be useful for us because we can apply the chain rule to a vector-valued function just by multiplying Jacobians.

As a little illustration of this, suppose we have a function $\boldsymbol{f}(x) = [f_1(x), f_2(x)]$ taking a scalar to a vector of size 2 and a function $\boldsymbol{g}(\boldsymbol{y}) = [g_1(y_1, y_2), g_2(y_1, y_2)]$ taking a vector of size two to a vector of size two. Now let's compose them to get $\boldsymbol{g}(x) = [g_1(f_1(x), f_2(x)), g_2(f_1(x), f_2(x))]$. Using the regular chain rule, we

[1] http://web.stanford.edu/class/cs224n/lectures/cs224n-2017-lecture5.pdf
[2] http://cs231n.github.io/optimization-2/

can compute the derivative of $\boldsymbol{g}$ as the Jacobian

$$\frac{\partial \boldsymbol{g}}{\partial x} = \begin{bmatrix} \frac{\partial}{\partial x} g_1(f_1(x), f_2(x)) \\ \frac{\partial}{\partial x} g_2(f_1(x), f_2(x)) \end{bmatrix} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1}\frac{\partial f_1}{\partial x} + \frac{\partial g_1}{\partial f_2}\frac{\partial f_2}{\partial x} \\ \frac{\partial g_2}{\partial f_1}\frac{\partial f_1}{\partial x} + \frac{\partial g_2}{\partial f_2}\frac{\partial f_2}{\partial x} \end{bmatrix}$$

And we see this is the same as multiplying the two Jacobians:

$$\frac{\partial \boldsymbol{g}}{\partial x} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{bmatrix}$$

## 3 Useful Identities

This section will now go over how to compute the Jacobian for several simple functions. It will provide some useful identities you can apply when taking neural network gradients.

(1) **Matrix times column vector with respect to the column vector** ($\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)

Suppose $\boldsymbol{W} \in \mathbb{R}^{n \times m}$. Then we can think of $\boldsymbol{z}$ as a function of $\boldsymbol{x}$ taking an $m$-dimensional vector to an $n$-dimensional vector. So its Jacobian will be $n \times m$. Note that

$$z_i = \sum_{k=1}^{m} W_{ik} x_k$$

So an entry $(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij}$ of the Jacobian will be

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} \sum_{k=1}^{m} W_{ik} x_k = \sum_{k=1}^{m} W_{ik} \frac{\partial}{\partial x_j} x_k = W_{ij}$$

because $\frac{\partial}{\partial x_j} x_k = 1$ if $k = j$ and 0 if otherwise. So we see that $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W}}$

(2) **Row vector times matrix with respect to the row vector** ($\boldsymbol{z} = \boldsymbol{x}\boldsymbol{W}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$?)

A computation similar to (1) shows that $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{W}^T}$.

(3) **A vector with itself**
($\boldsymbol{z} = \boldsymbol{x}$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$? )
We have $z_i = x_i$. So

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} x_i = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

So we see that the Jacobian $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a diagonal matrix where the entry at $(i, i)$ is 1. This is just the identity matrix: $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \boldsymbol{I}}$. When applying the chain rule, this term will disappear because a matrix multiplied by the identity matrix does not change.

(4) **An elementwise function applied a vector**
($\boldsymbol{z} = f(\boldsymbol{x})$, what is $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$? )
If $f$ is being applied elementwise, we have $z_i = f(x_i)$. So

$$(\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}})_{ij} = \frac{\partial z_i}{\partial x_j} = \frac{\partial}{\partial x_j} f(x_i) = \begin{cases} f'(x_i) & \text{if } i = j \\ 0 & \text{if otherwise} \end{cases}$$

So we see that the Jacobian $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}}$ is a diagonal matrix where the entry at $(i, i)$ is the derivative of $f$ applied to $x_i$. We can write this as $\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \text{diag}(f'(\boldsymbol{x}))}$. Since multiplication by a diagonal matrix is the same as doing elementwise multiplication by the diagonal, we could also write $\boxed{\circ f'(\boldsymbol{x})}$ when applying the chain rule.

(5) **Matrix times column vector with respect to the matrix**
($\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x}$, $\boldsymbol{\delta} = \frac{\partial J}{\partial \boldsymbol{z}}$ what is $\frac{\partial J}{\partial \boldsymbol{W}} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$?)

This is a bit more complicated than the other identities. The reason for including $\frac{\partial J}{\partial \boldsymbol{z}}$in the above problem formulation will become clear in a moment.

First suppose we have a loss function $J$ (a scalar) and are computing its gradient with respect to a matrix $\boldsymbol{W} \in \mathbb{R}^{n \times m}$. Then we could think of $J$ as a function of $\boldsymbol{W}$ taking $nm$ inputs (the entries of $\boldsymbol{W}$) to a single output ($J$). This means the Jacobian $\frac{\partial J}{\partial \boldsymbol{W}}$ would be a $1 \times nm$ vector. But in practice this is not a very useful way of arranging the gradient. It would be much nicer if the derivatives were in a $n \times m$ matrix like this:

$$\frac{\partial J}{\partial \boldsymbol{W}} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \cdots & \frac{\partial J}{\partial W_{1m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{n1}} & \cdots & \frac{\partial J}{\partial W_{nm}} \end{bmatrix}$$

Since this matrix has the same shape as $\boldsymbol{W}$, we could just subtract it (times the learning rate) from $\boldsymbol{W}$ when doing gradient descent. So (in a slight abuse of notation) let's find this matrix as $\frac{\partial J}{\partial \boldsymbol{W}}$ instead.

This way of arranging the gradients becomes complicated when computing $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$. Unlike $J$, $\boldsymbol{z}$ is a vector. So if we are trying to rearrange the gradients like with $\frac{\partial J}{\partial \boldsymbol{W}}$, $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$ would be an $n \times m \times n$ tensor! Luckily, we can avoid the issue by taking the gradient with respect to a single weight $W_{ij}$ instead.

$\frac{\partial \boldsymbol{z}}{\partial W_{ij}}$ is just a vector, which is much easier to deal with. We have

$$z_k = \sum_{l=1}^{m} W_{kl} x_l$$
$$\frac{\partial z_k}{\partial W_{ij}} = \sum_{l=1}^{m} x_l \frac{\partial}{\partial W_{ij}} W_{kl}$$

Note that $\frac{\partial}{\partial W_{ij}} W_{kl} = 1$ if $i = k$ and $j = l$ and 0 if otherwise. So if $k \neq i$ everything in the sum is zero and the gradient is zero. Otherwise, the only nonzero element of the sum is when $l = j$, so we just get $x_j$. Thus we find $\frac{\partial z_k}{\partial W_{ij}} = x_j$ if $k = i$ and 0 if otherwise. Another way of writing this is

$$\frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ x_j \\ 0 \\ \vdots \\ 0 \end{bmatrix} \leftarrow i\text{th element}$$

Now let's compute $\frac{\partial J}{\partial W_{ij}}$

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial \boldsymbol{z}} \frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial W_{ij}} = \sum_{k=1}^{m} \delta_k \frac{\partial z_k}{\partial W_{ij}} = \delta_i x_j$$

(the only nonzero term in the sum is $\delta_i \frac{\partial z_i}{\partial W_{ij}}$). To get $\frac{\partial J}{\partial \boldsymbol{W}}$ we want a matrix where entry $(i, j)$ is $\delta_i x_j$. This matrix is equal to the outer product $\boxed{\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{\delta}^T \boldsymbol{x}}$

(6) **Row vector time matrix with respect to the matrix** ($\boldsymbol{z} = \boldsymbol{x}\boldsymbol{W}$, $\boldsymbol{\delta} = \frac{\partial J}{\partial \boldsymbol{z}}$ what is $\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{\delta} \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}}$?)

A similar computation to (5) shows that $\boxed{\frac{\partial J}{\partial \boldsymbol{W}} = \boldsymbol{x}^T \boldsymbol{\delta}}$.

(7) **Cross-entropy loss with respect to logits** ($\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{\theta})$, $J = CE(\boldsymbol{y}, \hat{\boldsymbol{y}})$, what is $\frac{\partial J}{\partial \boldsymbol{\theta}}$?)

You computed this in Assignment 1! The gradient is $\boxed{\frac{\partial J}{\partial \boldsymbol{\theta}} = \hat{\boldsymbol{y}} - \boldsymbol{y}}$

(or $(\hat{\boldsymbol{y}} - \boldsymbol{y})^T$ if $\boldsymbol{y}$ is a column vector).

These identities will be enough to let you quickly compute the gradients for many neural networks. However, it's important to know how to compute Jacobians for other functions as well in case they show up. Some examples if you want practice: dot product of two vectors, elementwise product of two vectors, 2-norm of a vector. Feel free to use these identities on the midterm and assignments.

## 4 Example: 1-Layer Neural Network with Embeddings

This section provides an example of computing the gradients of a full neural network. In particular we are going to compute the gradients of the dependency parser you are building in Assignment 2. First let's write out the forward pass of the model.

$$
\begin{aligned}
\boldsymbol{x} &= [\boldsymbol{L}_{w_0}, \boldsymbol{L}_{w_1}, ..., \boldsymbol{L}_{w_{m-1}}] \\
\boldsymbol{z} &= \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}_1 \\
\boldsymbol{h} &= \text{ReLU}(\boldsymbol{z}) \\
\boldsymbol{\theta} &= \boldsymbol{h}\boldsymbol{U} + \boldsymbol{b}_2 \\
\hat{\boldsymbol{y}} &= \text{softmax}(\boldsymbol{\theta}) \\
J &= CE(\boldsymbol{y}, \hat{\boldsymbol{y}})
\end{aligned}
$$

It helps to break up the model into the simplest parts possible, so note that we defined $\boldsymbol{z}$ and $\boldsymbol{\theta}$ to split up the activation functions from the linear transformations in the network's layers. The dimensions of the model's parameters are

$$
\boldsymbol{L} \in \mathbb{R}^{|V| \times d} \qquad \boldsymbol{b}_1 \in \mathbb{R}^{1 \times D_h} \qquad \boldsymbol{W} \in \mathbb{R}^{md \times D_h} \qquad \boldsymbol{b}_2 \in \mathbb{R}^{1 \times N_c} \qquad \boldsymbol{U} \in \mathbb{R}^{D_h \times N_c}
$$

where $|V|$ is the vocabulary size, $d$ is the size of our word vectors, $m$ is the number of features, $D_h$ is the size of our hidden layer, and $N_c$ is the number of classes.

In this example, we will compute all the gradients:

$$
\frac{\partial J}{\partial \boldsymbol{U}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_2} \qquad \frac{\partial J}{\partial \boldsymbol{W}} \qquad \frac{\partial J}{\partial \boldsymbol{b}_1} \qquad \frac{\partial J}{\partial \boldsymbol{L}_{w_i}}
$$

To start with, recall that $\text{ReLU}(x) = \max(x, 0)$. This means

$$
\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if otherwise} \end{cases} = \text{sgn}(\text{ReLU}(x))
$$

where sgn is the signum function. Note that as you did in Assignment 1 with sigmoid, we are able to write the derivative of the activation in terms of the activation itself.

Now let's write out the chain rule for $\frac{\partial J}{\partial \boldsymbol{U}}$ and $\frac{\partial J}{\partial \boldsymbol{b}_2}$:

$$\frac{\partial J}{\partial \boldsymbol{U}} = \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}}$$
$$\frac{\partial J}{\partial \boldsymbol{b}_2} = \frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2}$$

Notice that $\frac{\partial J}{\partial \hat{\boldsymbol{y}}} \frac{\partial \hat{\boldsymbol{y}}}{\partial \boldsymbol{\theta}} = \frac{\partial J}{\partial \boldsymbol{\theta}}$ is present in both gradients. This makes the math a bit cumbersome. Even worse, if we're implementing the model without automatic differentiation, computing $\frac{\partial J}{\partial \boldsymbol{\theta}}$ twice will be inefficient. So it will help us to define some variables to represent the intermediate derivatives:

$$\boldsymbol{\delta}_1 = \frac{\partial J}{\partial \boldsymbol{\theta}} \qquad \boldsymbol{\delta}_2 = \frac{\partial J}{\partial \boldsymbol{z}}$$

These can be thought as the error signals passed down to $\boldsymbol{\theta}$ and $\boldsymbol{z}$ when doing backpropagation. We can compute them as follows:

$$\begin{aligned}
\boldsymbol{\delta}_1 &= \frac{\partial J}{\partial \boldsymbol{\theta}} = \hat{\boldsymbol{y}} - \boldsymbol{y} && \text{this is just identity (7)} \\
\boldsymbol{\delta}_2 &= \frac{\partial J}{\partial \boldsymbol{z}} = \frac{\partial J}{\partial \boldsymbol{\theta}} \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{using the chain rule} \\
&= \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{substituting in } \boldsymbol{\delta}_1 \\
&= \boldsymbol{\delta}_1 \, \boldsymbol{U}^T \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}} && \text{using identity (2)} \\
&= \boldsymbol{\delta}_1 \, \boldsymbol{U}^T \circ \mathrm{ReLU}'(\boldsymbol{z}) && \text{using identity (4)} \\
&= \boldsymbol{\delta}_1 \, \boldsymbol{U}^T \circ \mathrm{sgn}(\boldsymbol{h}) && \text{we computed this earlier}
\end{aligned}$$

A good way of checking our work is by looking at the dimensions of the terms in the derivative:

$$\begin{array}{ccccccc}
\dfrac{\partial J}{\partial \boldsymbol{z}} & = & \boldsymbol{\delta}_1 & & \boldsymbol{U}^T & \circ & \mathrm{sgn}(\boldsymbol{h}) \\
(1 \times D_h) & & (1 \times N_c) & & (N_c \times D_h) & & (D_h)
\end{array}$$

We see that the dimensions of all the terms in the gradient match up (i.e., the number of columns in a term equals the number of rows in the next term). This will always be the case if we computed our gradients correctly.

Now we can use the error terms to compute our gradients:

$$\begin{aligned}
\frac{\partial J}{\partial \boldsymbol{U}} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}} = \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{U}} = \boldsymbol{h}^T \boldsymbol{\delta}_1 && \text{using identity (6)} \\
\frac{\partial J}{\partial \boldsymbol{b}_2} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2} = \boldsymbol{\delta}_1 \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{b}_2} = \boldsymbol{\delta}_1 && \text{using identity (3)} \\
\frac{\partial J}{\partial \boldsymbol{W}} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{\delta}_2 \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{W}} = \boldsymbol{x}^T \boldsymbol{\delta}_2 && \text{using identity (6)} \\
\frac{\partial J}{\partial \boldsymbol{b}_1} &= \frac{\partial J}{\partial \boldsymbol{\theta}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_2 \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}_1} = \boldsymbol{\delta}_2 && \text{using identity (3)} \\
\frac{\partial J}{\partial \boldsymbol{L}_{w_i}} &= \frac{\partial J}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{L}_{w_i}} = \boldsymbol{\delta}_2 \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{L}_{w_i}}
\end{aligned}$$

All that's left is to compute $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{L}_{w_i}}$. It helps to split up $\boldsymbol{W}$ by rows like this:

$$\begin{aligned}
\boldsymbol{xW} &= [\boldsymbol{L}_{w_0}, \boldsymbol{L}_{w_1}, ..., \boldsymbol{L}_{w_{m-1}}]\boldsymbol{W} = [\boldsymbol{L}_{w_0}, \boldsymbol{L}_{w_1}, ..., \boldsymbol{L}_{w_{m-1}}] \begin{bmatrix} \boldsymbol{W}_{0:d} \\ \boldsymbol{W}_{d:2d} \\ \vdots \\ \boldsymbol{W}_{(m-1)d:md} \end{bmatrix} \\
&= \boldsymbol{L}_{w_0}\boldsymbol{W}_{0:d} + \boldsymbol{L}_{w_1}\boldsymbol{W}_{d:2d} + \cdots + \boldsymbol{L}_{w_{m-1}}\boldsymbol{W}_{(m-1)d:md} = \sum_{j=0}^{m-1} \boldsymbol{L}_{w_j}\boldsymbol{W}_{dj:d(j+1)}
\end{aligned}$$

When we compute $\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{L}_{w_i}}$, only the $i$th term in this sum is nonzero, so we get

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{L}_{w_i}} = \frac{\partial}{\partial \boldsymbol{L}_{w_i}} = \boldsymbol{L}_{w_i}\boldsymbol{W}_{di:d(i+1)} = (\boldsymbol{W}_{di:d(i+1)})^T$$

using identity (2).

# *Review of differential calculus theory* [1]

*Winter 2017*

[1] Author: Guillaume Genthial

**Keywords**: Differential, Gradients, partial derivatives, Jacobian, chain-rule

**This note is optional and is aimed at students who wish to have a deeper understanding of differential calculus**. It defines and explains the links between derivatives, gradients, jacobians, etc. First, we go through definitions and examples for $f : \mathbb{R}^n \mapsto \mathbb{R}$. Then we introduce the Jacobian and generalize to higher dimension. Finally, we introduce the chain-rule.

## 1 *Introduction*

We use derivatives all the time, but we forget what they mean. In general, we have in mind that for a function $f : \mathbb{R} \mapsto \mathbb{R}$, we have something like

$$f(x+h) - f(x) \approx f'(x)h$$

Some people use different notation, especially when dealing with higher dimensions, and there usually is a lot of confusion between the following notations

$$f'(x)$$
$$\frac{\mathrm{d}f}{\mathrm{d}x}$$
$$\frac{\partial f}{\partial x}$$
$$\nabla_x f$$

However, these notations refer to different mathematical objects, and the confusion can lead to mistakes. This paper recalls some notions about these objects.

**Scalar-product and dot-product**
Given two vectors $a$ and $b$,

- **scalar-product** $\langle a|b\rangle = \sum_{i=1}^{n} a_i b_i$
- **dot-product** $a^T \cdot b = \langle a|b\rangle = \sum_{i=1}^{n} a_i b_i$

# 2 *Theory for $f : \mathbb{R}^n \mapsto \mathbb{R}$*

## 2.1 *Differential*

**Formal definition**

Let's consider a function $f : \mathbb{R}^n \mapsto \mathbb{R}$ defined on $\mathbb{R}^n$ with the scalar product $\langle \cdot | \cdot \rangle$. We suppose that this function is **differentiable**, which means that for $x \in \mathbb{R}^n$ (fixed) and a small variation $h$ (can change) we can write:

$$f(x+h) = f(x) + \mathrm{d}_x f(h) + o_{h\to 0}(h) \tag{1}$$

**Notation**
$\mathrm{d}_x f$ is a **linear form** $\mathbb{R}^n \mapsto \mathbb{R}$
This is the best **linear approximation** of the function $f$

$\mathrm{d}_x f$ is called the **differential** of $f$ in $x$

$o_{h\to 0}(h)$ (Landau notation) is equivalent to the existence of a function $\epsilon(h)$ such that $\lim_{h\to 0} \epsilon(h) = 0$

and $\mathrm{d}_x f : \mathbb{R}^n \mapsto \mathbb{R}$ is a linear form, which means that $\forall x, y \in \mathbb{R}^n$ , we have $\mathrm{d}_x f(x+y) = \mathrm{d}_x f(x) + \mathrm{d}_x f(y)$.

**Example**

Let $f : \mathbb{R}^2 \mapsto \mathbb{R}$ such that $f(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) = 3x_1 + x_2^2$. Let's pick $\begin{pmatrix} a \\ b \end{pmatrix} \in \mathbb{R}^2$ and $h = \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} \in \mathbb{R}^2$. We have

$$\begin{aligned} f(\begin{pmatrix} a+h_1 \\ b+h_2 \end{pmatrix}) &= 3(a+h_1) + (b+h_2)^2 \\ &= 3a + 3h_1 + b^2 + 2bh_2 + h_2^2 \\ &= 3a + b^2 + 3h_1 + 2bh_2 + h_2^2 \\ &= f(a,b) + 3h_1 + 2bh_2 + o(h) \end{aligned}$$

$h^2 = h \cdot h = o_{h\to 0}(h)$

Then, $\mathrm{d}_{\begin{pmatrix} a \\ b \end{pmatrix}} f(\begin{pmatrix} h_1 \\ h_2 \end{pmatrix}) = 3h_1 + 2bh_2$

## 2.2 *Link with the gradients*

**Formal definition**

It can be shown that for all linear forms $a : \mathbb{R}^n \mapsto \mathbb{R}$, there exists a vector $u_a \in \mathbb{R}^n$ such that $\forall h \in \mathbb{R}^n$

$$a(h) = \langle u_a | h \rangle$$

**Notation** for $x \in \mathbb{R}^n$, the gradient is usually written $\nabla_x f \in \mathbb{R}^n$

The dual of a vector space $E^*$ is isomorphic to $E$
See Riesz representation theorem

In particular, for the **differential** $\mathrm{d}_x f$, we can find a vector $u \in \mathbb{R}^n$ such that

$$\mathrm{d}_x(h) = \langle u | h \rangle$$

.

The gradient has the **same shape** as $x$

We can thus define the **gradient** of $f$ in $x$

$$\nabla_x f := u$$

Then, as a conclusion, we can rewrite equation 2.1

**Gradients** and **differential** of a function are conceptually very different. The **gradient** is a vector, while the **differential** is a function

$$f(x+h) = f(x) + \mathrm{d}_x f(h) + o_{h\to 0}(h) \tag{2}$$
$$= f(x) + \langle \nabla_x f | h \rangle + o_{h\to 0}(h) \tag{3}$$

**Example**

Same example as before, $f : \mathbb{R}^2 \mapsto \mathbb{R}$ such that $f(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) = 3x_1 + x_2^2$. We showed that

$$\mathrm{d}_{\begin{pmatrix} a \\ b \end{pmatrix}} f(\begin{pmatrix} h_1 \\ h_2 \end{pmatrix}) = 3h_1 + 2bh_2$$

We can rewrite this as

$$\mathrm{d}_{\begin{pmatrix} a \\ b \end{pmatrix}} f(\begin{pmatrix} h_1 \\ h_2 \end{pmatrix}) = \langle \begin{pmatrix} 3 \\ 2b \end{pmatrix} | \begin{pmatrix} h_1 \\ h_2 \end{pmatrix} \rangle$$

and thus our gradient is

$$\nabla_{\begin{pmatrix} a \\ b \end{pmatrix}} f = \begin{pmatrix} 3 \\ 2b \end{pmatrix}$$

## 2.3 *Partial derivatives*

**Formal definition**

Now, let's consider an orthonormal basis $(e_1, \dots, e_n)$ of $\mathbb{R}^n$. Let's define the partial derivative

$$\frac{\partial f}{\partial x_i}(x) := \lim_{h\to 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_n)}{h}$$

**Notation**

Partial derivatives are usually written $\frac{\partial f}{\partial x}$ but you may also see $\partial_x f$ or $f'_x$

- $\frac{\partial f}{\partial x_i}$ is a **function** $\mathbb{R}^n \mapsto \mathbb{R}$
- $\frac{\partial f}{\partial x} = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})^T$ is a **function** $\mathbb{R}^n \mapsto \mathbb{R}^n$.
- $\frac{\partial f}{\partial x_i}(x) \in \mathbb{R}$
- $\frac{\partial f}{\partial x}(x) = (\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_n}(x))^T \in \mathbb{R}^n$

Note that the partial derivative $\frac{\partial f}{\partial x_i}(x) \in \mathbb{R}$ and that it is defined *with respect to the i-th component and evaluated in x.*

**Example**

Same example as before, $f : \mathbb{R}^2 \mapsto \mathbb{R}$ such that $f(x_1, x_2) = 3x_1 + x_2^2$. Let's write

Depending on the context, most people omit to write the $(x)$ evaluation and just write $\frac{\partial f}{\partial x} \in \mathbb{R}^n$ instead of $\frac{\partial f}{\partial x}(x)$

$$\begin{aligned}\frac{\partial f}{\partial x_1}\left(\begin{pmatrix}a\\b\end{pmatrix}\right) &= \lim_{h\to 0}\frac{f\left(\begin{pmatrix}a+h\\b\end{pmatrix}\right)-f\left(\begin{pmatrix}a\\b\end{pmatrix}\right)}{h}\\ &= \lim_{h\to 0}\frac{3(a+h)+b^2-(3a+b^2)}{h}\\ &= \lim_{h\to 0}\frac{3h}{h}\\ &= 3\end{aligned}$$

In a similar way, we find that

$$\frac{\partial f}{\partial x_2}\left(\begin{pmatrix}a\\b\end{pmatrix}\right) = 2b$$

## 2.4 *Link with the partial derivatives*

**Formal definition**

It can be shown that

$$\begin{aligned}\nabla_x f &= \sum_{i=1}^{n}\frac{\partial f}{\partial x_i}(x)e_i\\ &= \begin{pmatrix}\frac{\partial f}{\partial x_1}(x)\\ \vdots\\ \frac{\partial f}{\partial x_n}(x)\end{pmatrix}\end{aligned}$$

That's why we usually write

$$\nabla_x f = \frac{\partial f}{\partial x}(x)$$

(**same shape as** $x$)

$e_i$ is a orthonormal basis. For instance, in the canonical basis

$$e_i = (0,\dots,1,\dots 0)$$

with 1 at index $i$

where $\frac{\partial f}{\partial x_i}(x)$ denotes the partial derivative of $f$ with respect to the $i$th component, evaluated in $x$.

**Example**

We showed that

$$\begin{cases}\frac{\partial f}{\partial x_1}\left(\begin{pmatrix}a\\b\end{pmatrix}\right) &= 3\\ \frac{\partial f}{\partial x_2}\left(\begin{pmatrix}a\\b\end{pmatrix}\right) &= 2b\end{cases}$$

and that

$$\nabla_{\begin{pmatrix}a\\b\end{pmatrix}} f = \begin{pmatrix}3\\2b\end{pmatrix}$$

and then we verify that

$$\nabla_{\begin{pmatrix} a \\ b \end{pmatrix}} f = \begin{pmatrix} \frac{\partial f}{\partial x_1}(\begin{pmatrix} a \\ b \end{pmatrix}) \\ \\ \frac{\partial f}{\partial x_2}(\begin{pmatrix} a \\ b \end{pmatrix}) \end{pmatrix}$$

## 3 *Summary*

**Formal definition**

For a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, we have defined the following objects which can be summarized in the following equation

Recall that $a^T \cdot b = \langle a|b\rangle = \sum_{i=1}^n a_i b_i$

$$\begin{aligned} f(x+h) &= f(x) + \mathrm{d}_x f(h) + o_{h\to 0}(h) && \textbf{differential} \\ &= f(x) + \langle \nabla_x f | h \rangle + o_{h\to 0}(h) && \textbf{gradient} \\ &= f(x) + \langle \frac{\partial f}{\partial x}(x) | h \rangle + o_{h\to 0} \\ &= f(x) + \langle \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_n}(x) \end{pmatrix} | h \rangle + o_{h\to 0} && \textbf{partial derivatives} \end{aligned}$$

**Remark**

Let's consider $x : \mathbb{R} \mapsto \mathbb{R}$ such that $x(u) = u$ for all $u$. Then we can easily check that $\mathrm{d}_u x(h) = h$. As this differential does not depend on $u$, we may simply write $\mathrm{d}x$. That's why the following expression has some meaning,

The $\mathrm{d}x$ that we use refers to the differential of $u \mapsto u$, the identity mapping!

$$\mathrm{d}_x f(\cdot) = \frac{\partial f}{\partial x}(x)\mathrm{d}x(\cdot)$$

because

$$\begin{aligned} \mathrm{d}_x f(h) &= \frac{\partial f}{\partial x}(x)\mathrm{d}x(h) \\ &= \frac{\partial f}{\partial x}(x)h \end{aligned}$$

In higher dimension, we write

$$\mathrm{d}_x f = \sum_{i=1}^n \frac{\partial f}{\partial x_i}(x)\mathrm{d}x_i$$

## 4 ***Jacobian**: Generalization to $f : \mathbb{R}^n \mapsto \mathbb{R}^m$*

For a function

$$f : \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \mapsto \begin{pmatrix} f_1(x_1, \dots, x_n) \\ \vdots \\ f_m(x_1, \dots, x_n) \end{pmatrix}$$

We can apply the previous section to each $f_i(x)$ :

$$\begin{aligned} f_i(x+h) &= f_i(x) + \mathrm{d}_x f_i(h) + o_{h\to 0}(h) \\ &= f_i(x) + \langle \nabla_x f_i | h \rangle + o_{h\to 0}(h) \\ &= f_i(x) + \langle \frac{\partial f_i}{\partial x}(x) | h \rangle + o_{h\to 0} \\ &= f_i(x) + \langle (\frac{\partial f_i}{\partial x_1}(x), \dots, \frac{\partial f_i}{\partial x_n}(x))^T | h \rangle + o_{h\to 0} \end{aligned}$$

Putting all this in the same vector yields

$$f \begin{pmatrix} x_1 + h_1 \\ \vdots \\ x_n + h_n \end{pmatrix} = f \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x}(x)^T \cdot h \\ \vdots \\ \frac{\partial f_m}{\partial x}(x)^T \cdot h \end{pmatrix} + o(h)$$

Now, let's define the **Jacobian** matrix as

The **Jacobian** matrix has dimensions $m \times n$ and is a generalization of the gradient

$$J(x) := \begin{pmatrix} \frac{\partial f_1}{\partial x}(x)^T \\ \vdots \\ \frac{\partial f_m}{\partial x}(x)^T \end{pmatrix} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) \dots \frac{\partial f_1}{\partial x_n}(x) \\ \ddots \\ \frac{\partial f_m}{\partial x_1}(x) \dots \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix}$$

Then, we have that

$$\begin{aligned} f \begin{pmatrix} x_1 + h_1 \\ \vdots \\ x_n + h_n \end{pmatrix} &= f \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} + \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) \dots \frac{\partial f_1}{\partial x_n}(x) \\ \ddots \\ \frac{\partial f_m}{\partial x_1}(x) \dots \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix} \cdot h + o(h) \\ &= f(x) + J(x) \cdot h + o(h) \end{aligned}$$

**Example 1 :** $m = 1$

In the case where $m = 1$, the **Jacobian** is a **row vector**
$\frac{\partial f_1}{\partial x_1}(x) \dots \frac{\partial f_1}{\partial x_n}(x)$
Remember that our **gradient** was defined as a column vector with the same elements. We thus have that
$J(x) = \nabla_x f^T$

Let's take our first function $f : \mathbb{R}^2 \mapsto \mathbb{R}$ such that $f(\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) = 3x_1 + x_2^2$. Then, the Jacobian of $f$ is

$$\begin{aligned} \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) & \frac{\partial f}{\partial x_2}(x) \end{pmatrix} &= \begin{pmatrix} 3 & 2x_2 \end{pmatrix} \\ &= \begin{pmatrix} 3 \\ 2x_2 \end{pmatrix}^T \\ &= \nabla_f(x)^T \end{aligned}$$

**Example 2 :** $g : \mathbb{R}^3 \mapsto \mathbb{R}^2$ Let's define

$$g(\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}) = \begin{pmatrix} y_1 + 2y_2 + 3y_3 \\ y_1 y_2 y_3 \end{pmatrix}$$

Then, the Jacobian of $g$ is

$$\begin{aligned} J_g(y) &= \begin{pmatrix} \frac{\partial(y_1+2y_2+3y_3)}{\partial y}(y)^T \\ \frac{\partial(y_1 y_2 y_3)}{\partial y}(y)^T \end{pmatrix} \\ &= \begin{pmatrix} \frac{\partial(y_1+2y_2+3y_3)}{\partial y_1}(y) & \frac{\partial(y_1+2y_2+3y_3)}{\partial y_2}(y) & \frac{\partial(y_1+2y_2+3y_3)}{\partial y_3}(y) \\ \frac{\partial(y_1 y_2 y_3)}{\partial y_1}(y) & \frac{\partial(y_1 y_2 y_3)}{\partial y_2}(y) & \frac{\partial(y_1 y_2 y_3)}{\partial y_3}(y) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 \\ y_2 y_3 & y_1 y_3 & y_1 y_2 \end{pmatrix} \end{aligned}$$

## 5 *Generalization to* $f : \mathbb{R}^{n \times p} \mapsto \mathbb{R}$

If a function takes as input a matrix $A \in \mathbb{R}^{n \times p}$, we can transform this matrix into a vector $a \in \mathbb{R}^{np}$, such that

$$A[i, j] = a[i + nj]$$

Then, we end up with a function $\tilde{f} : \mathbb{R}^{np} \mapsto \mathbb{R}$. We can apply the results from 3 and we obtain for $x, h \in \mathbb{R}^{np}$ corresponding to $X, h \in \mathbb{R}^{n \times p}$,

$$\tilde{f}(x + h) = f(x) + \langle \nabla_x f | h \rangle + o(h)$$

where $\nabla_x f = \begin{pmatrix} \frac{\partial f}{\partial x_1}(x) \\ \vdots \\ \frac{\partial f}{\partial x_{np}(x)} \end{pmatrix}$.

Now, we would like to give some meaning to the following equation

$$f(X + H) = f(X) + \langle \nabla_X f | H \rangle + o(H)$$

The gradient of $f$ wrt to a matrix $X$ is a matrix of same shape as $X$ and defined by
$\nabla_X f_{ij} = \frac{\partial f}{\partial X_{ij}}(X)$

Now, you can check that if you define

$$\nabla_X f_{ij} = \frac{\partial f}{\partial X_{ij}}(X)$$

that these two terms are equivalent

$$\langle \nabla_x f | h \rangle = \langle \nabla_X f | H \rangle$$
$$\sum_{i=1}^{np} \frac{\partial f}{\partial x_i}(x) h_i = \sum_{i,j} \frac{\partial f}{\partial X_{ij}}(X) H_{ij}$$

## 6 Generalization to $f : \mathbb{R}^{n \times p} \mapsto \mathbb{R}^m$

Let's generalize the generalization of the previous section

Applying the same idea as before, we can write

$$f(x+h) = f(x) + J(x) \cdot h + o(h)$$

where $J$ has dimension $m \times n \times p$ and is defined as

$$J_{ijk}(x) = \frac{\partial f_i}{\partial X_{jk}}(x)$$

Writing the 2d-dot product $\delta = J(x) \cdot h \in \mathbb{R}^m$ means that the $i$-th component of $\delta$ is

You can apply the same idea to any dimensions!

$$\delta_i = \sum_{j=1}^{n} \sum_{k=1}^{p} \frac{\partial f_i}{\partial X_{jk}}(x) h_{jk}$$

## 7 Chain-rule

**Formal definition**

Now let's consider $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ and $g : \mathbb{R}^p \mapsto \mathbb{R}^n$. We want to compute the **differential** of the composition $h = f \circ g$ such that $h : x \mapsto u = g(x) \mapsto f(g(x)) = f(u)$, or

$$\mathrm{d}_x (f \circ g)$$

.

It can be shown that the differential is the composition of the differentials

$$\mathrm{d}_x (f \circ g) = \mathrm{d}_{g(x)} f \circ \mathrm{d}_x g$$

Where $\circ$ is the composition operator. Here, $\mathrm{d}_{g(x)} f$ and $\mathrm{d}_x g$ are linear transformations (see section 4). Then, the resulting differential is also a linear transformation and the **jacobian** is just the dot product between the jacobians. In other words,

The **chain-rule** is just writing the resulting **jacobian** as a dot product of **jacobians**. Order of the dot product is very important!

$$J_h(x) = J_f(g(x)) \cdot J_g(x)$$

where $\cdot$ is the dot-product. This dot-product between two matrices can also be written component-wise:

$$J_h(x)_{ij} = \sum_{k=1}^{n} J_f(g(x))_{ik} \cdot J_g(x)_{kj}$$

**Example**

Let's keep our example function $f : (\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}) \mapsto 3x_1 + x_2^2$ and our function $g : (\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}) = \begin{pmatrix} y_1 + 2y_2 + 3y_3 \\ y_1 y_2 y_3 \end{pmatrix}$.

The composition of $f$ and $g$ is $h = f \circ g : \mathbb{R}^3 \mapsto \mathbb{R}$

$$\begin{aligned} h(\begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}) &= f(\begin{pmatrix} y_1 + 2y_2 + 3y_3 \\ y_1 y_2 y_3 \end{pmatrix}) \\ &= 3(y_1 + 2y_2 + 3y_3) + (y_1 y_2 y_3)^2 \end{aligned}$$

We can compute the three components of the gradient of $h$ with the partial derivatives

$$\begin{aligned} \frac{\partial h}{\partial y_1}(y) &= 3 + 2y_1 y_2^2 y_3^2 \\ \frac{\partial h}{\partial y_2}(y) &= 6 + 2y_2 y_1^2 y_3^2 \\ \frac{\partial h}{\partial y_3}(y) &= 9 + 2y_3 y_1^2 y_2^2 \end{aligned}$$

And then our gradient is

$$\nabla_y h = \begin{pmatrix} 3 + 2y_1 y_2^2 y_3^2 \\ 6 + 2y_2 y_1^2 y_3^2 \\ 9 + 2y_3 y_1^2 y_2^2 \end{pmatrix}$$

In this process, we did not use our previous calculation, and that's a shame. Let's use the chain-rule to make use of it. With examples 2.2 and 4, we had

For a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, the Jacobian is the transpose of the gradient

$$\nabla_x f^T = J_f(x)$$

$$\begin{aligned} J_f(x) &= \nabla_x f^T \\ &= \begin{pmatrix} 3 & 2x_2 \end{pmatrix} \end{aligned}$$

We also need the jacobian of $g$, which we computed in 4

$$J_g(y) = \begin{pmatrix} 1 & 2 & 3 \\ y_2 y_3 & y_1 y_3 & y_1 y_2 \end{pmatrix}$$

Applying the chain rule, we obtain that the **jacobian** of $h$ is the product $J_f \cdot J_g$ (**in this order**). Recall that for a function $\mathbb{R}^n \mapsto \mathbb{R}$, the jacobian is formally the transpose of the gradient. Then,

$$\begin{aligned} J_h(y) &= J_f(g(y)) \cdot J_g(y) \\ &= \nabla^T_{g(y)} f \cdot J_g(y) \\ &= \begin{pmatrix} 3 & 2y_1y_2y_3 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ y_2y_3 & y_1y_3 & y_1y_2 \end{pmatrix} \\ &= \begin{pmatrix} 3 + 2y_1y_2^2y_3^2 & 6 + 2y_2y_1^2y_3^2 & 9 + 2y_3y_1^2y_2^2 \end{pmatrix} \end{aligned}$$

and taking the transpose we find the same gradient that we computed before!

**Important remark**

- The gradient is only defined for function with values in $\mathbb{R}$.
- Note that the chain rule gives us a way to compute the **Jacobian** and not the **gradient**. However, we showed that in the case of a function $f : \mathbb{R}^n \mapsto \mathbb{R}$, the **jacobian** and the **gradient** are directly identifiable, because $\nabla_x J^T = J(x)$. Thus, if we want to compute the gradient of a function by using the chain-rule, the best way to do it is to compute the Jacobian.
- As the gradient must have the same shape as the variable against which we derive, and
  - we know that the Jacobian is the transpose of the gradient
  - and the Jacobian is the dot product of Jacobians

  an efficient way of computing the gradient is to find the ordering of jacobian (or the transpose of the jacobian) that yield correct shapes!
- the notation $\frac{\partial \cdot}{\partial \cdot}$ is often ambiguous and can refer to either the gradient or the Jacobian.

# *CS224n: Natural Language Processing with Deep Learning*[1]

*Lecture Notes: TensorFlow*[2]

*Winter 2017*

[1] Course Instructors: Christopher Manning, Richard Socher

[2] Authors: Zhedi Liu, Jon Gauthier, Bharath Ramsundar, Chip Huyen

**Keyphrases: TensorFlow**

**Code Demo: `https://github.com/nishithbsk/tensorflow_tutorials`**

## 1 Introduction

TensorFlow is an open source software library for numerical computation using data flow graphs. It was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research.

Check the official tutorial `https://www.tensorflow.org/get_started/`

Nodes in TensorFlow's data flow graph represent mathematical operations, while the edges represent the multidimensional data arrays (tensors) communicated between them. The advantage of the flexible architecture is that it allows users to build complex models step by step and makes gradient calculations simple. TensorFlow programs use a tensor data structure to represent all data – only tensors are passed between operations in the computation graph. You can think of a TensorFlow tensor as an n-dimensional array or list. A tensor has a static type, a rank, and a shape.

## 2 Concepts

### 2.1 Variables, Placeholders, Mathematical Operations

Let's use

$$h = ReLU(Wx + b)$$

where *ReLU* (Rectified Linear Unit) is defined as $f(x) = max(0, x)$ as an example to take a closer look at TensorFlow's data flow graph, shown in Figure 1. There are three types of nodes in a flow graph: variables, placeholders and mathematical operations.

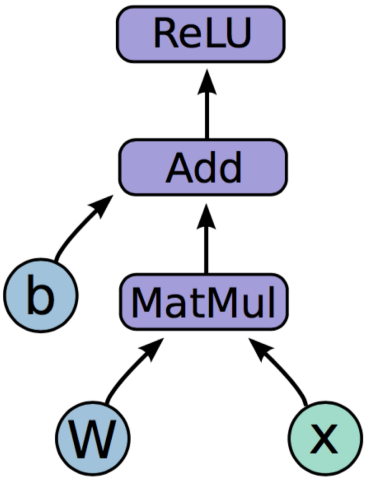



Figure 1: An Illustration of a TensorFlow Flow Graph

Variables are stateful nodes that maintain state across executions of the graph. By stateful, we mean that variables retain their current values over multiple executions, and it's easy to restore those saved values. Variables can be saved to disk during and after training. Typically, variables are parameters in a neural network. In our example, weights $W$ and bias $b$ are variables.

Placeholders are nodes whose values are fed in at execution time. The rationale behind having placeholders is that we want to be able

to build flow graphs without having to load external data, as we only want to pass in them at run time. Placeholders, unlike variables, require initialization. In order to initialize a placeholder, type and shape of data have to be passed in as arguments. Input data and labels are some examples that need to be initialized as placeholders. In our example, placeholder is $x$. See the code snippet below for initializing an input placeholder that has type tf.float32 and shape (batch_size, n_features), and a labels placeholder that has type tf.int32 and shape (batch_size, n_classes).

```
## Example code snippet
input_placeholder = tf.placeholder(tf.float32,
          shape=(batch_size, n_features))
labels_placeholder = tf.placeholder(tf.int32, s
          hape=(batch_size, n_classes))
```

Mathematical operations, as the name suggests, represent mathematical operations in a flow graph. In our example, `MatMul` (multiply two matrix values), `Add` (add element-wise with broadcasting) and `ReLU` (activate with element-wise rectified linear function) are mathematical operations.

Now we are ready to see our flow graph in code. Let's assume our input $x$ has shape $(N, Dx)$, $W$ has shape $(Dx, N)$ and type tf.float32, $b$ has shape $(N, 1)$ and we will initialize $W \sim \text{Uniform}(-1, 1)$ and $b = 0$. Then the code snippet below shows us how to build our flow graph for $h = ReLU(Wx + b)$.

```
## Example code snippet
import tensorflow as tf

b = tf.Variable(tf.zeros((N,)))
W = tf.Variable(tf.random_uniform((Dx, N), -1, 1))
x = tf.placeholder(tf.float32, (N, Dx))
h = tf.nn.relu(tf.matmul(x, W) + b)
```

The key thing to remember about symbolic programming language is that, up to what what we have written here, no data is actually being computed. $x$ is just a placeholder for our input data. A flow graph merely defines a function. We cannot do `print(h)` and gets its value as it only represents a node in the graph.

## 2.2 *Fetch, Fetch*

Now that we've defined a graph, the next steps are to deploy this graph with a session and run the session to get our outputs. A session is an environment that supports the execution of all operations to a particular execution context (e.g. CPU, GPU). A session can be easily built by doing `sess = tf.Session()`. In order for a session to

run, two arguments have to be fed: fetches and feeds. We use feeds and fetches to get data into and out of arbitrary operations.

Fetches represent a list of graph nodes and return the outputs of these nodes. We could fetch a single node or multiple tensors. See the code snippet below for an example of fetching two tensors: `mul` and `intermed`.

```
## Example code snippet
import tensorflow as tf

input1 = tf.constant([3.0])
input2 = tf.constant([2.0])
input3 = tf.constant([5.0])
intermed = tf.add(input2, input3)
mul = tf.mul(input1, intermed)

with tf.Session() as sess:
  result = sess.run([mul, intermed])
  print(result)

# output:
# [array([ 21.], dtype=float32), array([ 7.], dtype=float32)]
```

A feed, supplied as an argument to a `run()` call, temporarily replaces the output of an operation with a tensor value. The feed is only used for the `run` call to which it is passed. Essentially, feeds are dictionaries mapping placeholders to their values. Nodes that depend on placeholders cannot run unless their values are fed. See the code snippet below for an example of feeding a `feed_dict`.

```
## Example code snippet
import tensorflow as tf

input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.mul(input1, input2)

with tf.Session() as sess:
  print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

# output:
# [array([ 14.], dtype=float32)]
```

Before moving on to how to train a model, let's see a slightly more complicated example combining fetch and feed. In this example, we have a placeholder $x$ that requires initialization. We have two variables $W$ and $b$. It should be noted that when we launch a graph, all variables have to be explicitly initialized before one can run Ops that use their value. A variable can be initialized by running its initializer op, restoring the variable from a save file, or simply running an assign Op that assigns a value to the variable. In fact, the variable initializer op is just an assign Op that assigns the

variable's initial value to the variable itself. An example usage is `sess.run(w.initializer)` where $w$ is a variable in the graph. The more common initialization pattern is to use the convenience function `tf.initialize_all_variables()` to add an Op to the graph that initializes all the variables, as illustrated in the code snippet below.

```
## Example code snippet
import numpy as np
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100),
                -1, 1))

x = tf.placeholder(tf.float32, (100, 784))
h = tf.nn.relu(tf.matmul(x, W) + b)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
# {x: np.random.random(100, 784)} is a feed
# that assigns np.random.random(100, 784) to placeholder x
sess.run(h, {x: np.random.random(100, 784)})
```

## 2.3 *How to Train a Model in TensorFlow*

*1. Define a Loss*

The first thing to do in order to train a model is to build a loss node. See the code snippet below for an example of defining a cross-entropy loss. We build the loss node using labels and prediction. Note that we use `tf.reduce_sum` to compute the sum of elements across dimensions of a tensor. For our example, `axis=1` is used to perform a row-wise sum.

```
## Example code snippet
import tensorflow as tf

prediction = tf.nn.softmax(...)  #Output of neural network
label = tf.placeholder(tf.float32, [100, 10])

cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)

# More examples of using tf.reduce_sum
# 'x' is [[1, 1, 1]
#         [1, 1, 1]]
# tf.reduce_sum(x) ==> 6
# tf.reduce_sum(x, 0) ==> [2, 2, 2]
# tf.reduce_sum(x, 1) ==> [3, 3]
# tf.reduce_sum(x, 1, keep_dims=True) ==> [[3], [3]]
# tf.reduce_sum(x, [0, 1]) ==> 6
```

*2. Compute Gradients*

The next thing we have to do is to compute gradients. TensorFlow nodes have attached operations; therefore gradients with respect

to parameters are automatically computed with backpropagation. All we need to do is creating an optimizer object and calling the `minimize` function on previously defined loss. See code snippet below for an example of using a `GradientDescentOptimizer` optimizer where `cross_entropy` is the same as we introduced in the previous code snippet. Evaluating the minimization operation, `train_step` at runtime will automatically compute and apply gradients to all variables in the graph.

```
## Example code snippet
import tensorflow as tf

lr = 0.5 # learning rate
optimizer = tf.train.GradientDescentOptimizer(lr)
train_step = optimizer.minimize(cross_entropy)
```

3. *Train Model*

Now we are ready to train a model. This can simply be done by creating an iterating training schedule that feeds in data, labels and applies gradients to the variables, as shown in the code snippet below.

```
## Example code snippet
import tensorflow as tf

sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(1000):
  batch_x, batch_label = data.next_batch()
  sess.run(train_step, feed_dict={x: batch_x, label: batch_label}
```

## 2.4 *Variable Sharing*

One last important concept is variable sharing. When building complex models, we often need to share large sets of variables and might want to initialize all of them in one place. This can be done by using `tf.variable_scope()` and `tf.get_variable()`.

Imagine we are building a neural nets with two layers, if we use `tf.Variable`, we would have two sets of weights and two sets of biases. Let's assume that these variables are initialized in `define_variables()`. The problem arises when we want to use this model for two tasks that share the same parameters. We would have to call `define_variables(inputs)` twice, resulting in two sets of variables, 4 variables in each one, for a total of 8 variables. A common try to share variables is to create them in a separate piece of code and pass them to functions that use them, say by using a dictionary. I.e. the `define_variables` now takes two arguments, `inputs` and `variables_dict`. While convenient, cre-

ating a `variables_dict`, outside of the code, breaks encapsulation: 1) the code that builds the graph must document the names, types, and shapes of variables to create, and 2) When the code changes, the callers may have to create more, or less, or different variables. One way to address the problem is to use classes to create a model, where the classes take care of managing the variables they need. For a lighter solution, not involving classes, TensorFlow provides a Variable Scope mechanism that allows to easily share named variables while constructing a graph.

Variable Scope mechanism in TensorFlow consists of two main functions: `tf.get_variable(<name>, <shape>, <initializer>)` creates or returns a variable with a given name instead of a direct call to `tf.Variable`; `tf.variable_scope(<scope_name>)` manages namespaces for names passed to `tf.get_variable()`. `tf.get_variable` does one of two things depending on the scope it is called in. Let's set `v = tf.get_variable(name, shape, dtype, initializer)`.

Case 1: the scope is set for creating new variables, i.e. `tf.get_variable_scope(name, reuse=False)`.In this case, $v$ will be a newly created `tf.Variable` with the provided shape and data type. The full name of the created variable will be set to the current variable scope name + the provided name and a check will be performed to ensure that no variable with this full name exists yet. If a variable with this full name already exists, the function will raise a `ValueError`. If a new variable is created, it will be initialized to the value `initializer(shape)`. For example,

```
## Example code snippet
import tensorflow as tf

with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
assert v.name == "foo/v:0"
```

Case 2: the scope is set for reusing variables, i.e. `tf.get_variable_scope(name, reuse=True)`. In this case, the call will search for an already existing variable with name equal to the current variable scope name + the provided name. If no such variable exists, a `ValueError` will be raised. If the variable is found, it will be returned. If a variable already exists but `reuse=False`, program will crash. For example:

```
## Example code snippet
import tensorflow as tf

with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=False):
  v1 = tf.get_variable("v")      # CRASH foo/v:0 already exists!
```