DCMMC

A NOTE IN MACHINE LEARNING

Table of Contents

List of Figures

List of Tables

About this Note

Note that most of symbols in this note are vector, matrix, or tensor. Strictly speaking, we should write them as bold to differ from scalars. But for simplification, most bolds of them are ignored in this note.

Also,$\times$in superscript will leads into overflow in this latex source code. Therefore, all$\times$are replaced by $*$ in superscripts.

When there is no possibility for confusion, we write the propability $Pr(W = w)$ where W is the random variable and w is the specific value to the shorthands $P(w)$.

TODO(DCMMC)...

1 | NLP with DL

Natural Language Processing with Deep Learning – Stanford
CS224n Winter 2019

Learning Objectives:

- Word Vector
- Calculus Review
- RNN & Language Model
- Seq2Seq & Attention
- ConvNet for NLP
- Transformer

1.1 Word Vector

Arguably the most simple word vector, i.e., **one-hot vector**: an $\mathbb{R}^{|V|*1}$ vector with one 1 and the rest 0s. Note that these one-hot vectors are **orthogonal** (i.e., no similarity/relastionship) and V is a very big vocabulary ($\sim 500k$ words for english).

Another idea: **distributional representation** in modern statistical NLP. A word's meaning is given by the words that frequently appear close-by. Using some N-dim ($N \ll |V|$) space is sufficient to encode all semantics of our language into a dense vector. Once we get the word embedding matrix where each column is a word vector, we can query the word vector from one-hot representation by treating it as **lookup table** instead of using matrix product.

Dependencies: Machine Learning Basic

In traditional NLP (before 2013), words are regarded as discrete symbols (**localist** representation) and cannot capture similarity. One-hot vector is an example.

To evaluate word vectors, there are two fold: *intrinsic* (directly used, e.g. word analogies/similarity) and *extrinsic* (indirectly used in real task, e.g. Q&A). Word vector analygies for $a : b :: c : d$ is calculated by cosine similarity as example shown in Fig. 1.1:

$$d = \arg\max_i \frac{(x_b - x_a + x_c)^\top x_i}{\|x_b - x_a + x_c\|} \tag{1.1}$$

If we have hundreds of millions of words, it's okay to start the vectors *randomly*. If there is a *small* ($\leq 100,000$) training data set, it's best to just treat the pre-trained word vectors as *fixed*. In the other hand, if there is a large dataset, then we can gain by **fine tuning** of the word vectors.

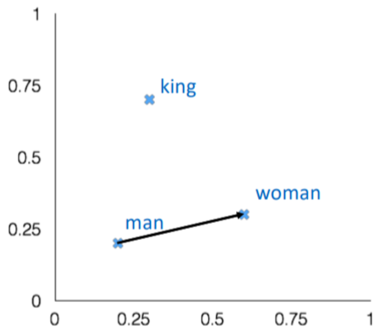


Figure 1.1: An example of word analogy of man:woman :: king:?

1.1.1 Word2vec

Two families of models: **Skip-gram** and **Continuous Bag of Words**.

Idea of **Skip-gram** (predicting context words by a given center word) in Word2vec[1]:

- a large corpus of text T with a vocabulary V

[1] Mikolov et al. 2013

- every word is represented by a vector $w \in \mathbb{R}^d$ and start off as a random vector
- use the (cosine) similarity of the word vectors for c (center word) and o (context/outside word) to calculate the probability of o given c: $p(w_o|w_c)$
- adjusting the word vectors to maximize the probability

The conditional probability is calculated by the **softmax** (normalize to probability distribution) of **cosine** similarity (review dot product: $\boldsymbol{a} \cdot \boldsymbol{b} = |\boldsymbol{a}||\boldsymbol{b}| \cos\langle \boldsymbol{a}, \boldsymbol{b}\rangle$). Note that the visualization of word vectos utilizes 2D projection (e.g. PCA) that will loss huge information.

Why we use two vectors per word? Make it simpler to calculate the gradient of loss function. Because the center word would be one of the choices for the context word and thus squared terms are imported. Average both vectors at the end is the final word vector.

$$p(w_o|w_c) = \frac{\exp(u_o^\top v_c)}{\sum_{w \in V}(u_w^\top v_c)} \tag{1.2}$$

where v_c denotes the center word vector of w when w is used as a center word in the formula, and u_w denotes the context word vector of w as the similar way. A demo of the window size and conditional probability is shown in Fig. 1.2.

The objective function (a.k.a loss or cost function) is given by the (average) negative log likelihood (abbr. **NLL**). The parameters of the model are adjusted by minimizing the loss function $J(\theta)$ or maximizing the likelihood. This is, give a high probability estimate to those words that occur in the context and low probability to those don't typically occur in the context.

$$\begin{aligned}
\arg\max_\theta L(\theta) &= \prod_{c=1}^{T} p(w_{c-m}, \cdots, w_{c-1}, w_{c+1}, \cdots, w_{c+m}|w_c;\theta) \\
&= \prod_{c=1}^{T} \prod_{\substack{-m \le j \le m \\ o=j+c \\ o \ne c}} p(w_o|w_c;\theta) \\
&\Downarrow \\
\arg\min_\theta -\frac{1}{T}\log L(\theta) &= -\frac{1}{T}\sum_{c=1}^{T} \sum_{\substack{-m \le j \le m \\ o=j+c \\ o \ne c}} \log p(w_o|w_c;\theta) \\
&= -\frac{1}{T}\sum_{c=1}^{T} \sum_{\substack{-m \le j \le m \\ o=j+c \\ o \ne c}} \left(u_o^\top v_c - \log \sum_{w \in V} \exp(u_w^\top v_c) \right)
\end{aligned} \tag{1.3}$$

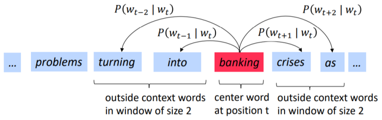


Figure 1.2: A demo of the window size and $p(w_o|w_c)$

where m is the window size, $\theta \in \mathbb{R}^{2d|V|}$ represents all model parameters. And we assume that $p(\cdot|w_c)$ are **i.i.d**.

The properties of log and arg max (arg min) used in Eq. 1.3 are VERY useful. $\exp(\cdot)$ ensures anything positive.

We use **gradient descent** (i.e. averaged gradient of all samples/windows) to optimize the loss function. Note that stochastic (one sample/window with noisy estimates of the gradients) or mini-batch (a subset of samples/windows with size powered of 2 such as 64) gradient descent methods are useful to prevent overfitting and train for large dataset. Calculating the gradient of the loss function is trivial:

$$\begin{aligned}\frac{\partial J}{\partial v_c} &= -\frac{1}{T}\sum_{c=1}^{T}\sum_{\substack{-m\leq j\leq m\\ o=j+c\\ o\neq c}}\left(u_o - \sum_{x\in V}\frac{\exp(u_x^\top v_c)u_x}{\sum_w \exp(u_w^\top v_c)}\right)\\ &= -\frac{1}{T}\sum_{c=1}^{T}\sum_{\substack{-m\leq j\leq m\\ o=j+c\\ o\neq c}}\left(u_o - \sum_{x\in V}p(w_x|w_c)\cdot u_x\right)\end{aligned} \tag{1.4}$$

$$\frac{\partial J}{\partial u_o} = -\frac{1}{T}\sum_{c=1}^{T}\sum_{\substack{-m\leq j\leq m\\ o=j+c\\ o\neq c}}(v_c - p(w_o|w_c)) \tag{1.5}$$

Iteratively update equation (naïve version) is given by:

$$\theta^{new} = \theta^{old} - \alpha\nabla_\theta J(\theta) \tag{1.6}$$

where α is the learning size (step size) such as 10^{-3}.

Note that the summation over $|V|$ ($\sum_{x\in V}$) is very expensive to compute! For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples! **negative sampling**: train binary logistic regression instead. $p(D = 1|w_o, w_c)$ denotes the probability when (w_o, w_c) came from the same window pf the corpus data, and $p(D = 0|w_o, \tilde{w}_o)$ is the probability given $(w_o, \tilde{w}_o)$ did not come from the same window (i.e. noisy/invalid pair). Randomly sample a bunch of noise words from the **unigram distribution** raised to the power of 3/4: $p(w) = {}^{U(w)^{3/4}}/_{Z}$, where $U(w)$ is the counts for every unique words (i.e. unigram) and Z is the nomalization term.

To avoid high frequence effect of words such as **of** and **the**, one simple way is just lop off the first biggest component in the word vector. The unigram with power of 3/4 in word2vec is also a trick to handle the effect, where it decrease how often you sample very common words and increase how often you sample rare words.

The objective function is also come from NLL:

$$J(\theta) = -\frac{1}{T}\sum_{c=1}^{T}\sum_{\substack{-m\leq j\leq m\\ o=j+c\\ o\neq c}}\left(\log\sigma\left(u_o^\top v_c\right) + \sum_{j\sim p(w)}\left[\log\sigma\left(-u_j^\top v_c\right)\right]\right) \quad (1.7)$$

where **sigmoid** function is $\sigma(x) = \frac{1}{1+e^{-x}}$ which can be seen as the 1D (binary) version of softmax and used to output the probability, and k is the number of negative samples such as 5 and 15. Note that according to the symmetric property of sigmoid function we get: $P(D=0|\tilde{w}_j, w_c) = 1 - P(D=1|\tilde{w}_j, w_c) = \sigma\left(-u_j^\top v_c\right)$.

Continuous Bag of Words (CBOW): predict center word from (bag of) context words. Similar to Skip-gram, the objective function is formulated as:

Although word2vec model is fairly simple and clean, there are actually many tricks which aren't particularly theoretical.

$$\begin{aligned} J &= -\frac{1}{T}\sum_{c=1}^{T}\log P(w_c|w_{c-m},\cdots,w_{c-1},w_{c+1},\cdots,w_{c+m}) && (1.8)\\ &= -\frac{1}{T}\sum_{c=1}^{T}\log p(v_c|\hat{u}) && (1.9)\\ &= -\frac{1}{T}\sum_{c=1}^{T}\log \underset{c}{\text{softmax}}(v_c^\top \hat{u}) && (1.10)\\ &= -\frac{1}{T}\sum_{c=1}^{T}(v_c^\top \hat{u} - \log\sum_{j=1}^{|V|}\exp(v_j^\top \hat{u})) && (1.11)\end{aligned}$$

where $\hat{u} = \frac{1}{2m}\sum_{\substack{-m\leq j\leq m\\ o=j+c\\ o\neq c}} u_o$

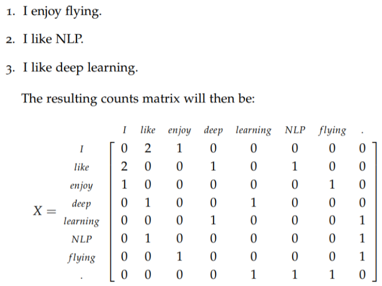
Although word2vec can capture complex patterns beyond word similarity, it has inefficient usage of statistics (i.e. rely on sampling rather than directly use counts of words).

1.1.2 HW1

A simple intro to co-occurrence matrix, SVD, cosine similarity, and some applications (e.g. word analogy) of word2vec.

1.1.3 GloVe

Co-occurrence matrix $X \in \mathbb{R}^{|V|*|V|}$ with window size k. Fig. 1.3 shows an example. Note that such matrix is extremely sparse and very high dimensional, and the dimensions of the matrix change very often as new words are added very frequently and corpus changes in size. We can perform SVD on X to reduce the dimensionality to 25 $\sim$ 1000-dim. In addition, there are some hacks to X that transform the raw



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

The resulting counts matrix will then be:

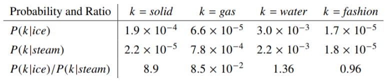
$$X = \begin{array}{c|cccccccc} & I & like & enjoy & deep & learning & NLP & flying & . \\ \hline I & 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ like & 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ enjoy & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ deep & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ learning & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ NLP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

Figure 1.3: An example of co-occurrence matrix with window size of 1

count introduced by [2]: (1) set upper bound (e.g. 100) or just ignore them all for the counts of too frequent words, (2) ramped windows that count closer words more. (3) use Pearson correlations instead of counts. Note that they made some interesting observation in their word vector that the verb (e.g. swim) and the corresponding doer (e.g. swimmer) pairs are roughly *linear components* (e.g. $\boldsymbol{v}_{swimmer} - \boldsymbol{v}_{swim} = k(\boldsymbol{v}_{driver} - \boldsymbol{v}_{drive})$).

TODO(DCMMC)... SVD

Although the aforementioned conventional method has disproportionate importance given to large counts and mainly only capture word similarity, it enjoys the fast training and efficient usage of statistics. GloVe (**Glo**bal **Ve**ctor) [3] combines the advantages from both of this conventional method (global count matrix factorization) and the DL-based methods (local context window methods) such as word2vec. It captures global corpus statistics directly.



Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 1.4: An example of the conditional probabilities and their ratio in GloVe paper.

Some notations: X_{ij} tabulate the number of times word j occurs in the context of word i, $X_i = \sum_k X_i k$ is the number of times any word appears in the context of word i i.e., the nomalization denominator. $P_{ij} = P(j|i) = {}^{X_{ij}}/_{X_i}$ is the probability that word j appear in the context of word i. The crucial insight is that the *ratios* of co-occurrence probabilities as shown in Fig. 1.4 to encode meaning components. We'd like to leverage the word vectors $w_i, w_j, \tilde{w}_k$ to represent such ratio: $F(w_i, w_j, \tilde{w}_k) = {}^{P_{ik}}/_{Pjk}$, where $\tilde{w}$ is a seperate *context* word vector for various *probe words* k, instead of the word vector w (similar to center word vector in skip-gram).

We can select a unique choice of F by enforcing a few desiderata (i.e. restrictions). To fit the demand of the *linear components* and the output *scalar* value, in addition to the *homomorphism* between the groups $(\mathbb{R}, -)$ and $(\mathbb{R}^+, \div)$ (i.e., $F(i,j) = {}^{P_{ik}}/_{Pjk} = {}^1/_{F(j,i)} = {}^{P_{jk}}/_{Pik}$), we can derivate that $F(w_i, w_j, \tilde{w}_k) = F\left((w_i - w_j)^\top \tilde{w}_k\right) = {}^{F(w_i^\top \tilde{w}_k)}/_{F(w_j^\top \tilde{w}_k)} = {}^{P_{ik}}/_{P_{jk}}$. Therefore, $F = \exp, w_i^\top \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i)$. Note that the symmetry property of co-occurrence: $X_{ik} = X_{ki}$. We add two biases to restore the symmetry: $w_i^\top \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik})$, where we can analogy that $b_i + \tilde{b}_j = \log X_i$.

More details, the relationship to the "global skip-gram" and the complexity refer to the original GloVe paper [4].

To handle the ill-defined log function when its argument be 0 (its common that $X_{ij} = 0$), the authors use the factorized log: $\log(X_{ik}) \to \log(1 + X_{ik})$.

$$w_i \cdot w_j = \log P(i|j) \tag{1.12}$$

$$w_x \cdot (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)} \tag{1.13}$$

Therefore, the ratios of co-occurrence probabilities is the **log-bilinear model with vector differences**. The final objective

[2] Rohde et al. 2005

[3] Pennington et al. 2014

[4] Pennington et al. 2014

function is *weighted* **least squares** (MSE) for this regression problem.

$$J = \sum_{i,j=1}^{V} f(X_{ij}) \left(w_i^\top \tilde{w}_j + b_i + \tilde{b}_j - logX_{ij} \right) \tag{1.14}$$

where weighted function (is also a hyperparamter) is:

$$f(x) = \begin{cases} \left(\frac{x}{x_{max}} \right)^{\alpha} & \text{if } x < x_{max} \\ 1 & \text{otherwise} \end{cases} \tag{1.15}$$

where $x_{max} = 100, \alpha = 3/4$ (*empirical* value).

1.1.4 Word sense ambiguity

Because most words have lots of meanings. One crude way [5] is to cluster word windows around words, retrain with each word assigned to multiple different clusters bank_1, bank_2, etc. Another method [6] is weighted sum of different senses of a word reside in a linear superposition, e.g.:

$$v_{\text{pike}} = \alpha_1 v_{\text{pike}_1} + \alpha_2 v_{\text{pike}_2} + \alpha_3 v_{\text{pike}_3} \tag{1.16}$$

where $\alpha_i = \frac{f_i}{\sum_{j=1}^{3} f_j}$ for frequency f.

The result is counterintuitive very well, because of the idea from *sparse* coding you can actually separate out the senses.

1.2 Math Backgrounds

For **multi-class classification** problem, **NLL** (negative likelihood loss) is the objective function of **Maximum Likelihood Estimate** (abbr, MLE):

$$J(\boldsymbol{\theta}) = -\sum_i \log p(y = y_i^{true} | \boldsymbol{x}_i; \boldsymbol{\theta}) \tag{1.17}$$

cross entropy (distance measure) between (discrete) distribution p and q is more convenient way:

$$H(p, q) = -\sum_{c=1}^{C} p(c) \log q(c) \tag{1.18}$$

However, in the multi-class (with single label) setting, the p(c) is the **ground truth distribution** which has the *one-hot* style

[5] Huang et al. 2012

[6] Arora et al. 2018

(**empirical distribution**), i.e. $p = [0, \cdots, 0, 1, 0, \cdots, 0]$ where 1 at the right class and 0 everywhere else. Therefore, the **cross entropy** in the multi-class classification is *equal* to the NLL.

A simple k-class model example is **dense layer** with *softmax*:

$$p(y|\boldsymbol{x}; \boldsymbol{\theta}) = softmax(\boldsymbol{W}_2 f(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b})) \tag{1.19}$$

where $\boldsymbol{\theta} = [\boldsymbol{W}_1, \boldsymbol{b}, \boldsymbol{W}_2]^\top$ are the parameters, $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{W}_1 \in \mathbb{R}^{n*m}, \boldsymbol{b} \in \mathbb{R}^n, \boldsymbol{W}_2 \in \mathbb{R}^{k*n}$, $f(\cdot)$ is a kind of simple activate (non-linear) function to provide non-linearity, such as $ReLU(x) = max(0, x)$. The visualization of neural network refer to [7].

The **Jacobian Matrix** (generalization of the gradient) of function $\boldsymbol{f}(\boldsymbol{x}) : \mathbb{R}^n \to \mathbb{R}^m$ is a $m \times n$ matrix: $\left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial f_i}{x_j}$.

Supposed that we have a function $\boldsymbol{g}(\boldsymbol{f}(x)), \boldsymbol{f} : \mathbb{R} \to \mathbb{R}^2, \boldsymbol{g} : \mathbb{R}^2 \to \mathbb{R}^2$, we can compute the partial derivative of $\boldsymbol{g}$ w.r.t x by **chain rule**:

$$\frac{\partial \boldsymbol{g}}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1}\frac{\partial f_1}{x} + \frac{\partial g_1}{\partial f_2}\frac{\partial f_2}{x} \\ \frac{\partial g_2}{\partial f_1}\frac{\partial f_1}{x} + \frac{\partial g_2}{\partial f_2}\frac{\partial f_2}{x} \end{bmatrix} \tag{1.20}$$

It is the same as multiplying the two Jacobians:

$\frac{dg_1}{d\boldsymbol{y}} = \frac{\partial g_1}{y_1} + \frac{\partial g_2}{y_2}$ is the relationship of the full differential and the partial differential.

$$\frac{\partial \boldsymbol{g}}{\partial x} = \frac{\partial \boldsymbol{g}}{\partial \boldsymbol{f}}\frac{\partial \boldsymbol{f}}{\partial x} = \begin{bmatrix} \frac{\partial g_1}{\partial f_1} & \frac{\partial g_1}{\partial f_2} \\ \frac{\partial g_2}{\partial f_1} & \frac{\partial g_2}{\partial f_2} \end{bmatrix} \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_2}{\partial x} \end{bmatrix} \tag{1.21}$$

There are some useful identities:

- $\frac{\partial \boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{I}$
- $\frac{\partial \boldsymbol{W}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{W}, \frac{\partial \boldsymbol{u}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{u}^\top$
- $\frac{\partial \boldsymbol{x}^\top \boldsymbol{W}}{\partial \boldsymbol{x}} = \boldsymbol{W}^\top$
- For elemenwise function $\boldsymbol{f}(\boldsymbol{x})$: $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} = \texttt{diag}(\boldsymbol{f}'(\boldsymbol{x}))$
- $\frac{\partial \boldsymbol{\theta}^\top (\boldsymbol{W} \cdot \boldsymbol{h})}{\partial \boldsymbol{W}} = \boldsymbol{\theta}\boldsymbol{h}^\top$ where $\boldsymbol{\theta} \in \mathbb{R}^{D_\theta * 1}, \boldsymbol{W} \in \mathbb{R}^{D_\theta * D_h}, \boldsymbol{h} \in \mathbb{R}^{D_h * 1}$
- For cross entropy loss: $J(\boldsymbol{h}) = -\boldsymbol{y}^\top \log(\hat{\boldsymbol{y}}) = -\boldsymbol{y}^\top \log \texttt{softmax}(\boldsymbol{h})$ ($\boldsymbol{y}$ is one-hot vector) is: $\frac{\partial J}{\partial \boldsymbol{h}} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^\top$

We can use **backward propagation** (reversed of the *topological sort*) and *re-use* intermediate nodes to reduce complexity in the *computation graph*.

Other machine learning basic concepts are: **regularization** (e.g. L2) to prevent **overfitting**, vectorization to parallelization, (non-linear) **activation function** (e.g. sigmoid, tanh, (leaky) ReLU), parameter initialization (e.g. Xavier), **Optimizer** (e.g. RMSprop, Adam), learning rate.

Compared with activations such as sigmoid and tanh, ReLU does not *saturate* even for larger values. Note that ReLU is not *differentiable* at 0, where we can use *sub-derivatives* in implementation with a certain value such as 0 or 1.

[7] ConvNetJS: https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

1.2.1 Data Preprocessing

- **Mean Subtraction (Shifting)**: Shifting all data so that they have zero mean as shown in Fig. 1.5. Formally, $\boldsymbol{x}^{(i)} \leftarrow \boldsymbol{x}^{(i)} - \mathbb{E}[\boldsymbol{x}^{(i)}]$ for sample i, where $\mathbb{E}$ indicates mean.

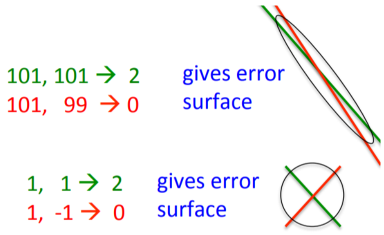


Figure 1.5: An exmaple of mean subtration.

- **Normalization (Scaling)**: Scale every input feature dimension to have similar ranges of magnitudes, as shown in Fig. 1.6. This is useful since input features are often measured in different "units", but we often want to initially consider all features as equally important. Formally, $x_j^{(i)} \leftarrow \frac{x_j^{(i)}}{\sigma_i(x_j^{(i)})}$ for feature j in sample i where $\sigma(\cdot)$ is the standard variance over $x_j^{(0)}, \cdots, x_j^{(N)}$.

- **Whitening**: Whitening converts the data to a have an identity covariance matrix - that is, features become uncorrelated and have a variance of 1. In the specific, we can divide the principal components achieved from PCA by the square roots of their eigenvalues (singular value).

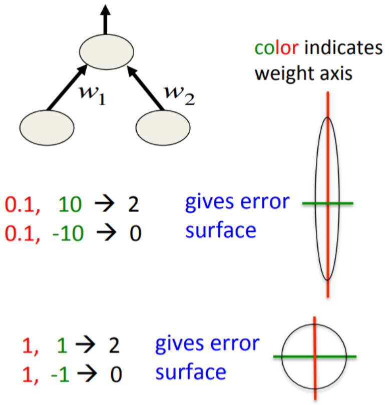


Figure 1.6: An example of normalization.

1.2.2 Parameter Initialization

If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights (e.g. different channels for learning different features in the same convolutional layer), they will always get exactly the same gradient. So we should initialize the weights to small random values.

A good starting strategy is to initialize the weights to small random numbers of **normal distribution** with the mean around 0. Xavier et al. [8] suggest that for sigmoid and tanh activation units, it's better for the weight matrix $W \in \mathbb{R}^{n^{(l+1)} * n^{(l)}}$ to be initialized randomly with a **uniform distribution**:

$$W \sim U\left[-\sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}, \sqrt{\frac{6}{n^{(l)} + n^{(l+1)}}}\right] \qquad (1.22)$$

where $n^{(l)}, n^{(l+1)}$ are also called **fan-in** and **fan-out**. Besides, bias units are initialized to 0.

1.2.3 Optimizer

To avoid a diverging loss (too large learning step) or unconverging (too small learning step), there are some learning strategies.

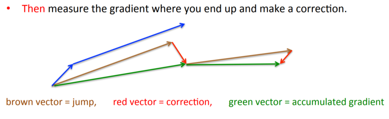
Annealing: start off with a high learning rate to approach a minimum quickly, after several iterations, the learning rate is reduced in some way under a more fine-grained scope.

- Exponential decay: $\alpha(t) = \alpha_0 e^{-kt}$ where α_0 is initial learning rate.

[8] Glorot and Bengio 2010

- Decrease over time: $\alpha(t) = {}^{(\alpha_0 \tau)}/_{\max(t,\tau)}$ where τ denotes the time at which the learning rate should start reducing.

  Momentum (a picture of it can be seen in Fig.1.7) based methods:



- First make a big jump in the direction of the previous accumulated gradient.
- Then measure the gradient where you end up and make a correction.

brown vector = jump,
red vector = correction,
green vector = accumulated gradient
blue vectors = standard momentum

Figure 1.7: A picture of momentum.

- AdaGrad: $\boldsymbol{m} \leftarrow \boldsymbol{m} + (\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))^2, \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \left(\sqrt{\boldsymbol{m}} + \epsilon\right)^{-1}$ where $\odot, (\cdot)^{-1}, (\cdot)^2, \sqrt{\cdot}$ are all element-wise operators, and ϵ is a very small value such as 10^{-8} to prevent **arithmetic underflow**. It leads to that parameters that have not been updated much in the past are likelier to have higher learning rates now.

- RMSprop: $\boldsymbol{m} \leftarrow \beta \boldsymbol{m} + (1-\beta)(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))^2, \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \odot \left(\sqrt{\boldsymbol{m}} + \epsilon\right)^{-1}$ where β is the decay rate with default value 0.9. Unlike AdaGrad, its updates do not become monotonically smaller.

- Adam[9]: $\boldsymbol{m} \leftarrow \beta_1 \boldsymbol{m} + (1-\beta_1) \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \boldsymbol{v} \leftarrow \beta_2 \boldsymbol{v} + (1-\beta_2)(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))^2, \hat{\boldsymbol{m}} = \boldsymbol{m}/(1-\beta_1^t), \hat{\boldsymbol{v}} = \boldsymbol{v}/(1-\beta_2^t), \boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \hat{\boldsymbol{m}} / \left(\sqrt{\hat{\boldsymbol{v}}} + \epsilon\right)^{-1}$, where $/$ is also a element-wise operator, hyperparameters $\beta_1 = 0.9, \beta_2 = 0.999 \in [0,1)$. $\hat{\boldsymbol{m}}, \hat{\boldsymbol{v}}$ are the bias-corrected $\boldsymbol{m}, \boldsymbol{v}$, and they indicate a rolling average of the gradients and a rolling average of the magnitudes of the gradients, respectively. In adition, $\boldsymbol{m}, \boldsymbol{v}$ are all initialized to $\mathbf{0}$ Adam is like a combination of RMSProp and momentum.

For the implementation of momentum such as RMSprop, there is a interesting small trick: use $\boldsymbol{m} = \boldsymbol{m} - (1-\beta)(\boldsymbol{m} - (\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))^2)$ instead of $\boldsymbol{m} = \beta \boldsymbol{m} + (1-\beta)(\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}))^2$. In such way, we need calculate only one multiplication, compared with original two multiplications.

1.2.4 Regularization

1. Dropout

During training, **dropout** [10] randomly disables units in the hidden layer by a mask vector drawn from Bernoulli distribution where each entry is 0 with probability p_{drop} and 1 with probability $(1-p_{\text{drop}})$:

$$\text{Dropout Layer: } d_i \sim \text{Bernoulli}(1-p_{\text{drop}}), \hat{\boldsymbol{h}}^{(t)} = \frac{1}{1-p_{\text{drop}}} \boldsymbol{d} \odot \boldsymbol{h}^{(t)} \tag{1.23}$$

where $\odot$ is element-wise product, $\boldsymbol{d} \in \{0,1\}^{D_h}, \boldsymbol{h}^{(t)} \in \mathbb{R}^{D_h}$.

If the expected output of a neuron during testing if far different as it was during training, the magnitude of the outputs could be radically different, and the behavior of the network is no longer well-defined. Therefore, all the parameters should divided by retain rate $1-p$ (blue part in above formula), so that $\mathbb{E}_{p_{\text{drop}}}[\boldsymbol{h}_{\text{drop}}]_i = h_i$. If we do not such correction in training, we should multiply $1-p$ to all related parameters.

If we use dropout in testing, the result is *unstable* (vary from every testing) because of the dropout is drawn from Bernoulli distribution. Therefore, we should apply dropout only during training but not during evaluation or testing.

[9] Kingma and Ba 2014

[10] Srivastava et al. 2014

2. Batch Normalization

Although **batch nomalization** [11] is like nomalization used in data preprocessing with N be the mini-batch size instead of the dataset size, it is inserted between hidden layers to normalize the output of last layer. It leads to achieve the fixed distributions of inputs that would remove the ill effects of the internal covariate shift. *Internal Covariate Shift* is defined as the change in the distribution of network activations due to the change in network parameters during training.

The batch normalization in training is defined as follows:

$$\text{BN}(h_i) = \gamma \frac{h_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \tag{1.24}$$

where γ, β are **trainable parameters**, $\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}^2$ are the mean and variance over the mini-batch as the way of normalization for data preprocessing. Since the mean subtraction will *ignore* the learned bias which may useful to the model. The trainable γ and β are used to corrected them and make the BN layer trainable. Them ensure that the batch normalization inserted in the network can represent the identity transform.

However, when testing, we cannot use mini-batch in most time. We instead feed one sample into the model. Therefore, we leverage m training mini-batches to perform **unbiased estimates** of them:

$$\mathbb{E}[h_i] \leftarrow \mathbb{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$

$$\texttt{Var}[h_i] \leftarrow \frac{m}{m-1}\mathbb{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

Note that in many implementations, the above estimation is replaced with the way like the momentum used in RMSprop. More details refer to the source code, e.g. Keras.

1.2.5 Practice: Named Entity Recognition

To find and classify words as entities (e.g. location, or organization) in text. One simple idea is that train softmax classifier to classify a center word by taking *concatenation* of word vectors surrounding it in a window (*word window*) [12]. To perform NER of localtion, we need (unnormalized) score for each window, and make *true window*s (i.e. location in the center) score larger and other *corrupt window*s score lower. The model is formulated as:

$$s = \boldsymbol{W}_2 f(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}) \tag{1.25}$$

The objective function (*max-margin loss*) is:

$$J = max(0, s_c - (s - 1)) \tag{1.26}$$

[11] Ioffe and Szegedy 2015

[12] Collobert and Weston 2008

where s and s_c is the score of true window and corrupt window. It ensure each window with an NER location at its center should have a score +1 higher than any window without a location at its center.

1.2.6 HW2

Gradient calculation and implementation of word2vec.

1. Written: Understanding word2vec

$$(a)\ \hat{y}_o = P(O=o|C=c)$$

$$(b)\ \frac{\partial J}{\partial \boldsymbol{v}_c} = (\hat{\boldsymbol{y}} - \boldsymbol{y})^\top \boldsymbol{U}^\top$$

$$(c)\ \frac{\partial J}{\partial \boldsymbol{U}} = \boldsymbol{v}_c(\hat{\boldsymbol{y}} - \boldsymbol{y})^\top$$

$$(d)\ \sigma(\boldsymbol{x}) = \frac{1}{1+\exp(-\boldsymbol{x})}, \frac{d\sigma(\boldsymbol{x})}{\boldsymbol{x}} = \mathtt{diag}(\sigma(x_i)(1-\sigma(x_i)))$$

$$(e)\ \frac{\partial J}{\partial \boldsymbol{v}_c} = \sum_k \sigma(\boldsymbol{u}_k^\top \boldsymbol{v}_c)\boldsymbol{u}_k^\top - (1-\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c))\boldsymbol{u}_o^\top$$

$$\frac{\partial J}{\partial \boldsymbol{u}_o} = (\sigma(\boldsymbol{u}_o^\top \boldsymbol{v}_c) - 1)\boldsymbol{v}_c^\top$$

$$\frac{\partial J}{\partial \boldsymbol{u}_k} = \sigma(\boldsymbol{u}_k^\top \boldsymbol{v}_c)\boldsymbol{v}_c^\top$$

$$(f)\ (i)\frac{\partial J}{\partial \boldsymbol{U}} = \sum_o \boldsymbol{v}_c(\hat{\boldsymbol{y}}_o - \boldsymbol{y}_o)^\top$$

$$(ii)\frac{\partial J}{\partial \boldsymbol{v}_c} = \sum_o (\hat{\boldsymbol{y}}_o - \boldsymbol{y}_o)^\top \boldsymbol{U}^\top$$

$$(iii)\frac{\partial J}{\partial \boldsymbol{v}_w} = \boldsymbol{0}$$

2 Coding: Implementing word2vec

Note that $\boldsymbol{U}, \boldsymbol{V}$ in the handout are the matrices whose i-th column is the n-dimensional embedded vector for word w_i. However, in the codes of HW2, all the centerWordVectors and outsideVectors are as rows.

Use shape convention to check the result.

1.3 Dependency Parser

Two views of linguistic structure: (1) constituency (i.e., phrase structure grammar, or context-free grammar) (2) Dependency structure. Dependence parse trees (single root with optional fake root, acyclic) use binary asymmetric relations which depicted as typed arrows going from *head* to *dependent*. Note that the natural language is ambiguity.

Basic transition-based dependency parser [13] with stack σ = [ROOT], buffer $\beta = w_1, \cdots, w_n$, set of dependency arcs $A = \emptyset$,

[13] Nivre 2003

and a set of actions (*transitions*) based on the above 3-tuple:

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

where $r(w_j, w_i)$ denotes w_i is the dependency of w_j (e.g. nsubj(ate → I)), $|$ and $\cup$ stand for concatenate. The finish state is: $\sigma = [w], \beta = \emptyset$. How to select (search) the best choice among the exponential size of different possible parse trees is the problem. In 1960s, they use *dynamic programming algorithms* ($\mathcal{O}(n^3)$). In paper [14], the authors predict each action by a discriminative classifier (e.g. SVM classifier) which is more efficient but the accuracy is fractionally below the state-of-the-art.

1.3.1 Neural Dependency Parsing

Compared with traditional sparse feature-based discriminative dependency parsers, the work by [15] utilizes **feedforward neural network model** with simple **dense layers** and the softmax layer to predict each transition. The input features with embedding dimension d are:

1. $x^w \in \mathbb{R}^{d*N_w}$: 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, $N_w = 18$.

2. $x^t \in \mathbb{R}^{d*N_t}$: The corresponding POS (Part-of-speech, e.g. noun, verb, adjective) tags for S_{word}, $N_t = 18$.

3. $x^l \in \mathbb{R}^{d*N_l}$: The corresponding arc labels of words, excluding those 6 words on the stack/buffer, $N_l = 12$.

The predicted class is the one of transitions (i.e. shift, left/right arc reduction): $p = \texttt{softmax}(\boldsymbol{W}_2 f(\boldsymbol{W}_1^w \boldsymbol{x}^w + \boldsymbol{W}_1^t \boldsymbol{x}^t + \boldsymbol{W}_1^l \boldsymbol{x}^l + \boldsymbol{b}_1) + \boldsymbol{b}_2)$, where $f(\cdot)$ is the activation function (e.g. ReLU, or x^3). The number of class is 3 when untyped reductions or $T * 2 + 1$ when typed reductions (e.g. left-arc reduction with type *nsubj*).

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.

1.3.2 HW3

1. Machine Learning & Neural Networks

(a) Adam

[14] Nivre 2003

[15] Chen and Manning 2014

(i) Because $\beta = 0.9$, most of the final gradients ($\boldsymbol{m}$) come from the past (90%). Even if current gradients are varying much from previous, it only occupy $1 - \beta_1 = 0.1$ of the final gradients.

(ii) Parameters that have not been updated much in the past are likelier to have higher learning rates.

(b) Dropout

(i) If the expected output of a neuron during testing if far different as it was during training, the magnitude of the outputs could be radically different, and the behavior of the network is no longer well-defined. Thus, all the parameters should divided by retain rate $1 - p$, so that $\mathbb{E}_{p_{\text{drop}}}[\boldsymbol{h}_{\text{drop}}]_i = h_i$.

(ii) If we use dropout in testing, the result is unstable because of the dropout is drawn from Bernoulli distribution.

2. Neural Transition-Based Dependency Parsing

(a) The remaindered configuration is:

Stack	Buffer	New dependency	Transition
[ROOT, parsed, this]	[sentence, correctly]		SHIFT
[ROOT, parsed, this, sentence]	[correctly]		SHIFT
[ROOT, parsed, sentence]	[correctly]	sentence → this	LEFT-ARC
[ROOT, parsed]	[correctly]	parsed → sentence	RIGHT-ARC
[ROOT, parsed, correctly]	[]		SHIFT
[ROOT, parsed]	[]	parsed → correctly	RIGHT-ARC
[ROOT]	[]	ROOT → parsed	RIGHT-ARC

(b) $2n$

Note that in the source code, the restriction that the version of PyTorch must be 1.0.0 is meaningless and thus I remove it.

1.4 Language Modeling and Recurrent Neural Networks

Language Modeling: given a sequence of words $\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(t)}$, compute the probability distribution of the next word at $\boldsymbol{x}^{(t+1)}$:

$$P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(t)}) \tag{1.27}$$

The joint probability of a text is:

$$P(\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(T)}) = \prod_{t=1}^{T} P(\boldsymbol{x}^{(t)}|\boldsymbol{x}^{(t-1)}, \cdots, \boldsymbol{x}^{(1)}) \tag{1.28}$$

n-gram is a chunk of n consecutive words: unigram, bigram, trigram, 4-gram, ... n-gram language model is based on a simplifying assumption: $\boldsymbol{x}^{(t+1)}$ depends only on the preceding $n-1$ words with i.i.d.:

$$\begin{aligned} P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\cdots,\boldsymbol{x}^{(1)}) &= P(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\cdots,\boldsymbol{x}^{(t-n+2)}) \\ &= \frac{P(\boldsymbol{x}^{(t+1)},\boldsymbol{x}^{(t)},\cdots,\boldsymbol{x}^{(t-n+2)})}{P(\boldsymbol{x}^{(t)},\cdots,\boldsymbol{x}^{(t-n+2)})} \end{aligned}$$

where the n-gram and (n-1)-gram probabilities are calculated by *counting*. There are some *sparsity problems* with the above n-gram models such as the numerator or denominator is zero. Some tricks such as *smoothing* (add small δ to the count) and *backoff* (e.g. 4-gram backoff to 3-gram) are proposed to solve them.

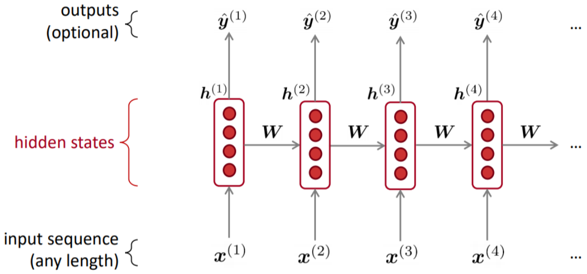


Figure 1.8: Principle of RNN

To process *variable* length **sequential input** such as text, **Recurrent Neural Network** (RNN) is introduced. As the principle of RNN shown in Fig. 1.8: *repeat* (i.e. **unfold** or unroll) the same RNN cell for each time-step but with different input and previous **hidden state**. A vanilla RNN for language modeling is:

Note that for n-gram, increasing n makes sparsity problems worse. Typically $n \leq 5$.

$$\begin{aligned} \boldsymbol{h}^{(t)} &= \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x\boldsymbol{x}^{(t)} + \boldsymbol{b}_1\right) \\ \hat{\boldsymbol{y}} &= P(\boldsymbol{x}^{(t)}|\boldsymbol{x}^{(t-1)},\cdots,\boldsymbol{x}^{(1)}) \\ &= \texttt{softmax}(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2) \end{aligned}$$

where $\sigma(\cdot)$ is the activation function, and $\boldsymbol{h}^{(0)}$ is the initial (random or zero) hidden state. The gradient w.r.t. the weight matrix is the *sum* of the gradients w.r.t each time it appears using **back-propagation through time** (BPTT, just as same as normal back-prop). And the **evaluation metric** for language modeling is *perlexity* which is equal to the exponential of the cross-entropy losses:

$$\begin{aligned} \text{perplexity} &= \prod_{t=1}^{T}\left(\frac{1}{P_{LM}(\boldsymbol{x}^{(t+1)}|\boldsymbol{x}^{(t)},\cdots,\boldsymbol{x}^{(1)})}\right)^{1/T} \\ &= \exp\left(\frac{1}{T}\sum_{t=1}^{T} -\log\hat{\boldsymbol{y}}^{(t)}\right) \qquad (1.29) \end{aligned}$$

There are some other applications of RNN: part-of-speech tagging, named entity recognition, sentence classification, text generator, encoder module, etc. The final feature can be the final hidden state or elemen-wise max/mean of all hidden states. Using chain rule, we get $\frac{\partial J^{(n)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial J^{(n)}}{\partial \boldsymbol{h}^{(n)}} \times \prod_{i=2}^{n} \frac{\partial \boldsymbol{h}^{(n)}}{\partial \boldsymbol{h}^{(n-1)}} = \frac{\partial J^{(n)}}{\partial \boldsymbol{h}^{(n)}} \times \prod_{i=2}^{n} \sigma' \circ \boldsymbol{W}_h$. For a large n and small $\boldsymbol{W}_h$, it's easy to encounter the vanishing gradient problem. In overall, the *vanilla* RNN has these disadvantages: (1) recurrent computation is slow (2) hard to access long-term information (**long-term dependencies**) due to *gradient vanish* and *gradient explosion*.

We can formalize the above vanishing intuitions according to [16]. Let $\boldsymbol{W}_h$ have the **eigenvalues** $\lambda_1, \cdots, \lambda_n$ such that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$ and the corresponding (left) eigenvectors $\boldsymbol{q}_1^\top, \cdots, \boldsymbol{q}_n^\top$ which have unit norms: $\boldsymbol{q}_i^\top \boldsymbol{W}_h = \lambda_i \boldsymbol{q}_i$. We can rewrite the gradients $\frac{J^{(n)}}{\boldsymbol{h}^{(n)}} = \sum_{i=1}^{N} c_i \boldsymbol{q}_i^\top$ where $c_i = 0$ for $i < j$ and $c_j \neq 0$. Thus, the overall gradient is:

$$\begin{aligned}
\frac{\partial J^{(n)}}{\partial \boldsymbol{h}^{(1)}} &= \frac{\partial J^{(n)}}{\partial \boldsymbol{h}^{(n)}} \times \prod_{i=2}^{n} \sigma' \circ \boldsymbol{W}_h \\
&= \sum_{i=1}^{N} c_i \boldsymbol{q}_i^\top (\texttt{diag}(\sigma'))^{n-1} (\boldsymbol{W}_h)^{n-1} \\
&= \sum_{i=1}^{N} c_i \boldsymbol{q}_i^\top (\boldsymbol{W}_h)^{n-1} (\texttt{diag}(\sigma'))^{n-1} \\
&= c_j \lambda_j^{n-1} \boldsymbol{q}_j^\top (\texttt{diag}(\sigma'))^{n-1} + \lambda_j^{n-1} \sum_{i=j+1}^{N} c_i \left(\frac{\lambda_i}{\lambda_j}\right)^{n-1} \boldsymbol{q}_i^\top (\texttt{diag}(\sigma'))^{n-1} \\
&\approx c_j \lambda_j^{n-1} (\sigma')^{n-1} \boldsymbol{q}_j^\top
\end{aligned}$$

where $\frac{\lambda_i}{\lambda_j} < 1$, and for large n we have $\lim_{n\to\infty} \left(\frac{\lambda_i}{\lambda_j}\right)^{n-1} = 0$. Therefore, if $\forall j, \sigma' < \frac{1}{\lambda_j}$ then we get vanishing gradient. Note that, $\sup \text{sigmoid}' = \frac{1}{4}, \sup ReLU' = 1$. Thus, the largest eigenvalue $\lambda_1 < \frac{1}{\sigma'}$ will lead to vanishing.

For avoid gradient explosion, one simple method is *gradient clipping*: $\hat{\boldsymbol{g}} \leftarrow \frac{threshold}{\|\boldsymbol{g}\|} \boldsymbol{g}$ if $\|\boldsymbol{g}\| \geq threshold$. As for fixing vanishing gradient, many RNN variants are introduced such as **Long Short-Term Memory** (LSTM) [17] and **Gated Recurrent Unit** (GRU) [18]. LSTM uses two *separated* memories: *hidden state* $\boldsymbol{h}^{(t)}$ for *short-term* information and *cell state* $\boldsymbol{c}^{(t)}$ for *long-term* information. There are three *gates* performed in each LSTM *cell*:

Forget gate: $\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f)$

Input gate: $\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i)$

Output gate: $\boldsymbol{o}^{(t)} = \sigma(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o)$

New cell content: $\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$

[16] Pascanu et al. 2013

[17] Hochreiter and Schmidhuber 1997

[18] Cho et al. 2014

$$\text{Cell state: } \boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$
$$\text{Hidden state: } \boldsymbol{h}^{(t)} = \boldsymbol{o}^{(t)} \circ \tanh \boldsymbol{c}^{(t)}$$

where σ is sigmoid function, $\circ$ is element-wise product, and tanh used in hidden state (the last formula) is to provide non-linearity and normalize $\boldsymbol{c}^{(t)}$ to $(0, 1)$. The structure of LSTM is shown in Fig.1.9 made by colah's blog.

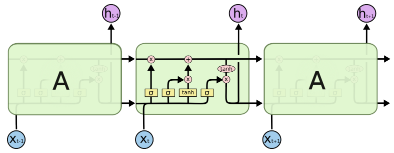


Figure 1.9: The repeating module in an LSTM contains four interacting layers.

Those three gates (forget, input, output) enable the abilities of erase, read and write for LSTM. Each element of the gates are between 1 (open) and 0 (close). While The LSTM architecture makes it *easier* for the RNN to preserve long-distance dependencies, it does not *guarantee* that there is no vanishing/exploding gradient.

In the other hand, GRU combines input and forget gate into *update* gate, and add new *reset* gate to select useful part of previous hidden state to compute new state content. While there is no conclusive evidence that GRU consistently performs better than LSTM or vice versa, GRU is computed more efficient due to fewer parameters.

$$\text{Update gate: } \boldsymbol{u}^{(t)} = \sigma(\boldsymbol{W}_u \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_u \boldsymbol{x}^{(t)} + \boldsymbol{b}_u)$$
$$\text{Reset gate: } \boldsymbol{r}^{(t)} = \sigma(\boldsymbol{W}_r \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_r \boldsymbol{x}^{(t)} + \boldsymbol{b}_r)$$
$$\text{New hidden state content: } \tilde{\boldsymbol{h}}^{(t)} = \tanh(\boldsymbol{W}_h(\boldsymbol{r}^{(t)} \circ \boldsymbol{h}^{(t-1)}) + \boldsymbol{U}_h \boldsymbol{x}^{(t)} + \boldsymbol{b}_h)$$
$$\text{Hidden state: } \boldsymbol{h}^{(t)} = (1 - \boldsymbol{u}^{(t)}) \circ \boldsymbol{h}^{(t-1)} + \boldsymbol{u}^{(t)} \circ \tilde{\boldsymbol{h}}^{(t)}$$

The vanishing gradient problem appears not only in RNNs, but also for most all other neural networks inlcuding MLP (dense layers) and CNNs, especially for deep ones. One solution is add more *direct connections* between future apart layers to allow gradients flow more easier. For example, **Residual connections** (aka. ResNet [19] or skip-connections) is shown in Fig.1.10 where an identity skips two layers. Another example is **Dense connections** (aka. DenseNet [20]) which directly connect every layers to every layers where the output of each layer will **concatenate** the input as presented in Fig.1.11. **Highway connections** is inspired from the gates of LSTM and similar to residual connections, where the identity connect and the transformation layer is controlled by a dynamic gate.

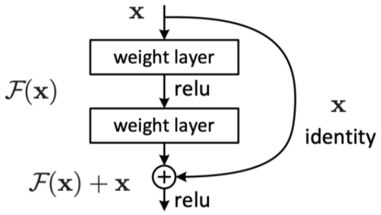


Figure 1.10: Residual connections.

Apart from the above RNNs, there are other important RNN architectures: **Bidirectional RNNs** and **Multi-layer RNNs** (aka. stacked RNNs). The definition of bidirectional RNNs is given by:

$$\text{Forward RNN: } \overrightarrow{\boldsymbol{h}}^{(t)} = \overrightarrow{\text{RNN}}_{FW}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$$
$$\text{Backward RNN: } \overleftarrow{\boldsymbol{h}}^{(t)} = \overleftarrow{\text{RNN}}_{BW}(\overleftarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$$
$$\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$$

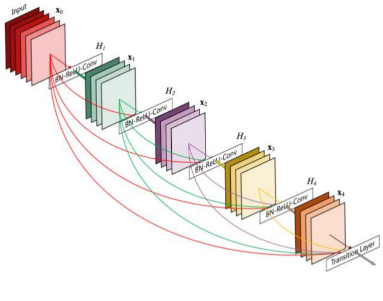


Figure 1.11: Dense Net.

While LSTM bacame the dominant approach between 2013 to 2016,

[19] He et al. 2016

[20] Huang et al. 2017

Transormer is the state-of-the-art now in 2019. More descriptions refer to Section 1.6.

1.5 Seq2Seq and Attention

Pre-neural machine translation: (1) Rule-based bilingual dictionary in 1950s (2) Statistical machine translation from 1990s to 2010s. More formally, statistical machine translation from *source language* x to *target language* y is given by:

$$\begin{aligned}\arg\max_y P(y|x) &= \arg\max_y \frac{P(x,y)}{P(x)} \\ &= \arg\max_y \frac{P(x|y)P(y)}{P(x)} \\ &= \arg\max_y P(x|y)P(y)\end{aligned}$$

where $P(y|x)$ is broke down to two components according to **Baye's rule**: $P(x|y)$ is the translation model which learn from parallel (bilingual) data to model the fidelity of words and phrases whether x is well- or ill-formed, $P(y)$ is the language model which learn from monolingual data of target language to model the fluency of the whole sentence regardless of their connection to the French.

In practice, we further consider **alignment** (word-level correspondence) because there are one-to-many, many-to-one, many-to-many, and even no couterpart apart from one-to-one reflection relations. One example of one-to-many (entarté) and no counterpart (a) is shown in Fig.1.12. More examples refer to the original paper [21]. Therefore, we add alignment to the model: $P(x, a|y)$.

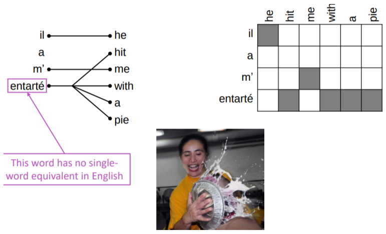


Figure 1.12: Alignment from french to english translation.

The core idea of Seq2Seq model is using two RNN to construct an *encode-decode* architecture. At test time, first we feed source sentence (with embedding) into the encoder RNN, then we use the last hidden state of the encode RNN as the initial state of the decoder RNN as a conditional language model. The output word at position t is given by:

$$w_t = \texttt{softmax}(RNN(h^{(t-1)}, W_e \arg\max \texttt{softmax}(h^{(t-1)})))$$

where W_e is the embedding table of the target language and the first input x_1 = <START> is a special token and repeatedly output until output <END>. Note that there are two different embedding lookup table for source and target language. When traning, we need provide parallel dataset whose samples consist of bilingual sentences. As for traning, the diagram is represented in Fig.1.13.

However, the aforementioned (greedy) decoding has no way to undo decisions. This is, if one of the output words are wrong, all the follow-up outputs are also wrong. **Beam search** decoding is utilized to fix

[21] Brown et al. 1993

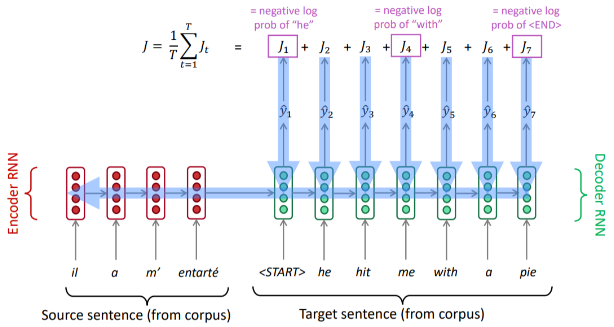

= negative log prob of "he"
= negative log prob of "with"
= negative log prob of <END>
$J = \frac{1}{T}\sum_{t=1}^{T} J_t = \boxed{J_1} + J_2 + J_3 + \boxed{J_4} + J_5 + J_6 + \boxed{J_7}$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_4$
$\hat{y}_5$
$\hat{y}_6$
$\hat{y}_7$
Encoder RNN
Decoder RNN
il
a
m'
entarté
<START>
he
hit
me
with
a
pie
Source sentence (from corpus)
Target sentence (from corpus)

Figure 1.13: Training phase for NMT.

this problem: on each step of decoder, keep track of the k (in practice around 5 to 10) most probable partial translation (*hypotheses*, path, or branch). The target is the path with the largest cumulative log probabilities with shortest one (i.e. average). Apart from all paths reach <END> as the stop sign, we can set some pre-defined cutoff for maximum number of timesteps or finished paths.

Although NMT is much simple and less human engineering effort while achieves better performance, NMT is diffcult to control (i.e. specify rules or guidelines) and less interpretable which leads to hard to debug.

The popular evaluation metric for MT is **BLEU** (Bilingual Evaluation Understudy). Its calculation is based on n-gram precision plus a penality for too-short translations. Note that BLEU is useful but imperfect because there are many valid translations which has low n-gram overlap with the ground truth translation. NMT outperforms SMT quickly, but there are still many difficulties remain:

- Out-of-vocabulary words.
- Domain mismatch between train and test data.
- Maintaining context over longer text (long-term dependencies).
- Low-resource language pairs (few-shot learning).
- Using common sense is still hard.

We notice that only the last hidden state from encoder RNN to represent the whole source sentence may be the information bottleneck. Thus we introduce Seq2Seq with **attention**: on each step of the decoder, we use the attention distribution (like soft version of alignment) for each hidden states of encoder RNN to take a weighted sum of the encoder hidden states as the current decoder hidden state. Note that sometimes the input of decoder RNN will concatenate the previous translated word vector and the previous attention output. The diagram can be seen in Fig.1.14. In addition, attention in here helps with

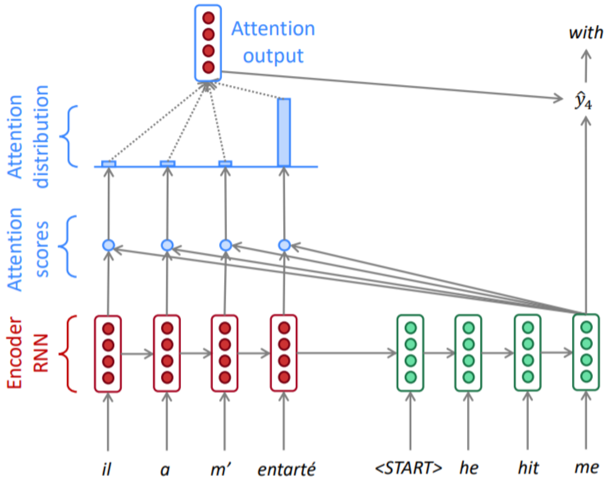


Figure 1.14: Seq2Seq with attention.

vanishing gradient problem by providing shortcut to faraway states, and also provides some interpretability. More formally, the attention for Seq2Seq is:

$$\begin{aligned}
&\boldsymbol{e}^{(t)} = [\boldsymbol{s}_t^\top \boldsymbol{h}_1, \cdots, \boldsymbol{s}_t^\top \boldsymbol{h}_n] \in \mathbb{R}^n \\
&\text{Attention distribution: } \boldsymbol{\alpha}^{(t)} = \texttt{softmax}(\boldsymbol{e}^{(t)}) \\
&\text{Attention ouput: } \boldsymbol{a}^{(t)} = \sum_{i=1}^{n} \alpha_i^t \boldsymbol{h}_i \in \mathbb{R}^h \\
&\text{Attention decoder hidden state: } [\boldsymbol{a}^{(t)}; \boldsymbol{s}_t] \in \mathbb{R}^{2h}
\end{aligned} \tag{1.30}$$

where encoder hidden states $\boldsymbol{h}_1, \cdots, \boldsymbol{h}_n \in \mathbb{R}^h$, and decoder hidden state at timestep t is $\boldsymbol{s}_t \in \mathbb{R}^h$.

More general definition of attention: given a set of vector *values*, and a vector *query*, attention is to compute a weighted sum of the values, dependent on the query. We can found that attention is a way to obtain a *fixed-size* representation of an arbitrary set of representations (e.g. sequential features). There are many ways to obtain query vector: dot product $e_i = \boldsymbol{s}^\top \boldsymbol{h}_i$, multiplicative $e_i = \boldsymbol{s}^\top \boldsymbol{W} \boldsymbol{h}_i$, additive $e_i = \boldsymbol{v}^\top \tanh(\boldsymbol{W}_1 \boldsymbol{h}_i + \boldsymbol{W}_2 \boldsymbol{s})$, where $\boldsymbol{v}, \boldsymbol{W}, \boldsymbol{W}_1, \boldsymbol{W}_2$ are trainable parameters.

1.6 Contextual Word Representations and Pretraining

2 | Bibliography

Arora, S., Li, Y., Liang, Y., Ma, T., and Risteski, A. (2018). Linear algebraic structure of word senses, with applications to polysemy. *Transactions of the Association for Computational Linguistics*, 6(0):483–495.

Brown, P. F., Della Pietra, S. A., Della Pietra, V. J., and Mercer, R. L. (1993). The mathematics of statistical machine translation: Parameter estimation. *Computational Linguistics*, 19(2):263–311.

Chen, D. and Manning, C. (2014). A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 740–750, Doha, Qatar. Association for Computational Linguistics.

Cho, K., van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Collobert, R. and Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning*, ICML 08, page 160167, New York, NY, USA. Association for Computing Machinery.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Huang, E., Socher, R., Manning, C., and Ng, A. (2012). Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 873–882, Jeju Island, Korea. Association for Computational Linguistics.

Huang, G., Liu, Z., v. d. Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2261–2269.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML15, page 448456. JMLR.org.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

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

Nivre, J. (2003). An efficient algorithm for projective dependency parsing. In *Proceedings of the Eighth International Conference on Parsing Technologies*, pages 149–160, Nancy, France.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML13, page III1310III1318. JMLR.org.

Pennington, J., Socher, R., and Manning, C. (2014). Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar. Association for Computational Linguistics.

Rohde, D. L., Gonnerman, L. M., and Plaut, D. C. (2005). An improved model of semantic similarity based on lexical co-occurrence.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958.