Encoding Sentences With Recurrent Neural Network

Author: *Bowen Zheng*

Supervisor: *Dr David Barber*

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Master of Research

of

University College London.

Department of Computer Science

University College London

November 16, 2016

I, Bowen Zheng, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

Learning the vector representations for sentences is a tough problem, although there are some models can learn such representations, it is still not clear about which architectures are preferred and what objectives can result in most useful representations. This thesis presents a comparison between several recurrent neural network based encoders have exactly same architecture but are trained with different objectives.

We first describe how we pretrain recent proposed sequence to sequence model on different objectives to obtain sentence encoders. Next we introduce metrics that are used to evaluate these encoders. Finally, we discuss our evaluation results and figure out a few points that can affect the performance of these encoders.

Acknowledgements

Firstly, I would like to thank my supervisor, Dr David Barber, for his patience and continuous support during this year. He makes me realise the basis of doing research is to be precise. Furthermore, I would like to thank all members of my study group, without your help it would be really difficult for me to develop knowledge about machine learning from knowing nothing. And I also would like to thank UCL Media Future group to share their computational resource with me. Finally, thank for those encouragements from my parents and Miss Kejing when I feel struggling this year.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Building a human like agent that can process natural language information is a long term and illustrated goal in Natural Language Processing (NLP). One fundamental problem for NLP researchers is to find certain machine interpretable representations, that can describe full information of language data. In today's NLP research, there are many established methods can acquire representations for words from unlabelled text data [2, 3, 4]. These methods have been well studied and analysed[5], the high quality representations can be used as additional features to improve the performance of language processing system[6]. In contrast, much less efforts are made to develop methods that can learn the representations of phrases or sentences. This is because little consensus about which model architectures or training objectives can results in useful representations for input sentences[7].

Recurrent Neural Networks (RNNs)[8] are neural networks which are well-suited for NLP tasks. An RNN encode a sequence into high dimensional vector (known as the hidden representation or hidden state) that incorporates new inputs using a nonlinear function[9]. However, it is very difficult for standard RNNs to store information in hidden state for long time[10]. Gated RNNs[11, 12] are extension of standard RNNs with "memory units" which has been shown capability to store and access information over very long time period. In various NLP tasks RNNs have outperformed traditional models[13]. The recent proposed RNN based sequence to sequence (seq2seq) model[1, 14] allows researchers to deal with variable length

of inputs and outputs, the application of seq2seq model in machine translation has achieved remarkable results[1, 14]. It is shown that seq2seq model can be trained to reconstruct input sentences as its outputs[15]. This allows us to pretrain an RNN sentence encoder in an unsupervised way.

We are typically interested in how different objectives can affect the pretrain sentence encoder and learnt representations. Therefore we pretrain seq2seq model with different objectives to obtain several sentence encoders, and evaluate these encodes on a range of tasks.

The structure of remaining contents are organised as follow: **Chapter 2** reviews standard n-gram language model and background knowledge about RNN. It also provides some brief introduction about neural language models. **Chapter 3** describes the seq2seq model in detail and training objective functions, as well as evaluation metrics. **Chapter 4** discusses the datasets and training strategies that are used to train our models. **Chapter 5** shows the analysis on evaluation metrics and experiments. **Chapter 6** gives a final conclusion about this thesis.

Chapter 2

Background

In this chapter we provide necessary background knowledge for understanding the content of this thesis.

2.1 N-gram Language Models

In NLP, the **language models** are developed to compute the probability of a given sentence[16]. For example, a candidate sentence

"*The cat sits on the mat.*"

A language model should assign a probability $p(\text{The cat sits on the mat})$ to this sentence. We can treat the example sentence as a sequence of words, then $p(\text{The cat sits on the mat})$ becomes the joint probability of the terms in this sentence[17], and it is possible to factorise this joint probability by applying the chain rule of probability. In our example, we have 6 words $\{w_1 = \text{the}, w_2 = \text{cat}, w_3 = \text{sits}, w_4 = \text{on}, w_5, \text{the}, w_6 = \text{mat}\}$,before factorisation we add another two special tokens $w_0 =< \text{s} >$,$w_7 =< \backslash\text{s} >$ to indicate the begin and end of sentence. Now the probability assigned to this modified sentence becomes

$$p(w_{0:7}) = p(w_0)\prod_{i=7}^{7} p(w_i|w_{1:i-1}) \tag{2.1}$$

$$p(w_0) = 1 \tag{2.2}$$

We can simplify the conditional distribution terms by assuming that each word only depends on $n-1$ previous words so that

$$p(w_i|w_{1:i-1}) \approx p(w_i|w_{i-n+1:i-1}) \tag{2.3}$$

This yields to the wildly used **n-gram** language models in NLP[16], if we choose $n = 2$, then we call this 2-gram model and so on. If a training corpus (a collection of text documents) is available, we can compute the value of conditional distribution $p(w_i|w_{i-n+1:i-1})$ based on co-occurrence counts. For example, if we want to train a 2-gram model for the given corpus

"$< s >$ The cat sits on the mat $< \backslash s >$"

"$< s >$ The cat sits on the desk $< \backslash s >$"

We can work out the conditional probabilities for instance $p(\text{cat}|\text{the})$ as

$$p(\text{cat}|\text{the}) = \frac{C(\text{the cat})}{C(\text{the})} = \frac{2}{4} = 0.5 \tag{2.4}$$

Where $C(\cdot)$ is the counts of token's occurrence in given corpus.

Vocabulary (V)	Occurrence	**2-grams**	Occurrence
$< s >$	2	$< s >$ the	2
the	4	the cat	2
cat	2	cat sits	2
sits	2	sits on	2
on	2	on the	2
mat	1	the mat	1
desk	1	the desk	1
$< \backslash s >$	2	mat $< \backslash s >$	1
-	-	desk $< \backslash s >$	1

Table 2.1: Occurrence for 2-gram and words of the example corpus

Having a language model like n-gram that can assign the probability to sentences is essential for many NLP tasks. For example in **spelling correction**, a language model would assign a low probability to "Their are two apples on the table" because the word "there" is mistyped as "their", this can help spelling checking model to detect errors[16].

	$< s >$	the	cat	sits	on	mat	desk	$< \backslash s >$
$< s >$	0	0	0	0	0	0	0	0
the	0.5	0	0	0	0.5	0	0	0
cat	0	1	0	0	0	0	0	0
sits	0	0	1	0	0	0	0	0
on	0	0	0	1	0	0	0	0
mat	0	1	0	0	0	0	0	0
desk	0	1	0	0	0	0	0	0
$< \backslash s >$	0	0	0	0	0	0.5	0.5	0

Table 2.2: 2-gram conditional probabilities for given example

The performance of n-gram increases with the training corpus size and n-gram size[16, 13]. However at same time, we have to spend huge amount of memory to store $p(w_i|w_{i-n+1:i-1})$ produce by n-gram if n and size of vocabulary $|\mathbf{V}|$ is to large, because the storage array requires $|\mathbf{V}|^n$ memory to store all possible $p(w_i|w_{i-n+1:i-1})$, and usually this array is very sparse (as shown in Table 2.2). On the other hand, n-gram models only encode co-occurrence probabilities between words and grams. To capture additional information between words, for example, the semantic similarity we need to build a massive semantic network[18] which can be very expensive to create and maintain. Therefore we desire a model that can learn richer information from training corpus and requires cheaper storage when doing large scale language modelling.

Artificial neural networks are a family of machine learning methods, which have shown its ability to learn the dense and informative representations of input data[19]. Thus many researchers have applied neural networks to learn representations for text data and solve NLP task, many of these applications have achieved remarkable results[13]. Before we go further, we first provide some background knowledge about neural networks.

2.2 Feedforward Neural Network

A feedforward neural network defines a mapping function $f(\mathbf{x}, \theta)$ and tries to learn the value of parameters θ result in the best approximation of the actual mapping

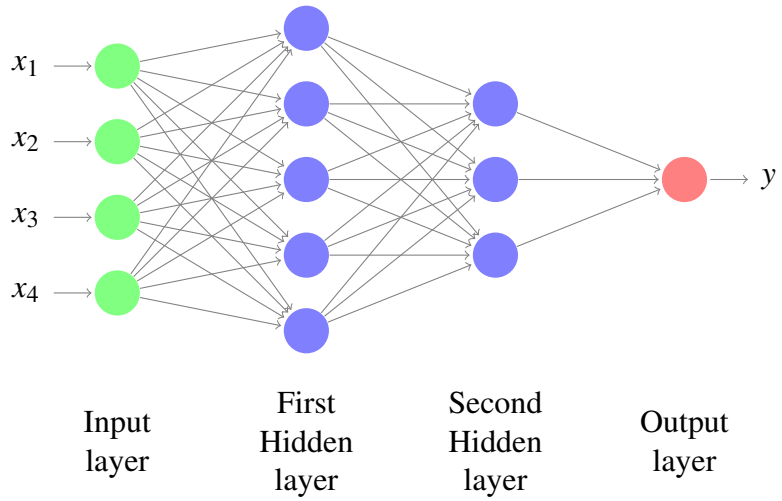


Figure 2.1: An example of deep feedforward network with 2 hidden layers

$\mathbf{y} = f^*(\mathbf{x})$ between $\mathbf{y}$ and $\mathbf{x}$. For example, a chain like function $\mathbf{y}' = f^3(f^2(f^1(\mathbf{x})))$. We usually call f^1 the **first hidden layer** of the network, f^2 the **second hidden layer** and so on. And we refer the final function f^3 as the **output layer** of neural network. The number of layers gives the **depth** of the neural network. This is the terminology that the name **"deep learning"** arises[20].

2.2.1 Set up A Neural Network

Training a supervised neural network $f(\mathbf{x}, \theta)$ requires to define a loss function $L(\mathbf{y}, f(\mathbf{x}, \theta))$, where $\mathbf{y}$ is actual label of input $\mathbf{x}$. In most cases, we follow the principle of maximum likelihood to train neural networks; that is to maximise the defined conditional distribution $p_{model}(\mathbf{y}|\mathbf{x})$. Usually, we use the cross entropy between the training data and model's predictions as the loss function[20].

$$L(\mathbf{y}, f(\mathbf{x}, \theta)) = -\mathbb{E}_{\hat{p}_{data}} \log p_{model}(\mathbf{y}|\mathbf{x}) \tag{2.5}$$

Where the neural network $f(\mathbf{x}, \theta)$ is trained to approach the conditional distribution $p_{model}(\mathbf{y}|\mathbf{x})$. For $f(\mathbf{x}, \theta)$ to be a valid probability distribution, the output of $f(\mathbf{x}, \theta)$ must within $[0, 1]$.

In many tasks, the neural network is required to make a binary prediction of vari-

able $y \in \{0,1\}$. In this case, a convenient choice of the output layer for the neural network is the sigmoid function. For example, we can define a very simple neural network with sigmoid output layer as

$$f(\theta, \mathbf{x}) = \sigma(\theta^T \mathbf{x}) = \frac{1}{1+\exp(-\theta^T \mathbf{x})} \tag{2.6}$$

Where the first hidden layer is the inner product of θ and $\mathbf{x}$, and the output layer is the sigmoid function. Our cross entropy loss function is

$$L(y, f(\mathbf{x}, \theta)) = y \log f(\mathbf{x}, \theta) + (1-y)\log(1 - f(\mathbf{x}, \theta)) \tag{2.7}$$

And we usually make our prediction $\hat{y}$ by

$$y' = \begin{cases} 0 & \text{if } f(\theta, \mathbf{x}) > 0.5 \\ 1 & \text{if } f(\theta, \mathbf{x}) \leq 0.5 \end{cases} \tag{2.8}$$

When solving the multiple classes classification problems, for example, a three types classification problem, the actual label $\mathbf{y}$ is usually a one-hot vector, which can be

$$\mathbf{y} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{2.9}$$

To perform this task the output layer is required to assign a probability to each class, and the sum of assigned probabilities must equal to 1.

$$f(\mathbf{x}, \mathbf{W}) = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} \tag{2.10}$$

$$\sum_{i=1}^{3} p_i = 1 \tag{2.11}$$

A common choice of output layer for multiple classes classification problems is softmax function, the softmax function assigns a probability to each element as

$$p_i = \frac{\exp\{\mathbf{w}_i^T \mathbf{x}\}}{\sum_{i=1}^{3} \exp\{\mathbf{w}_i^T \mathbf{x}\}} \tag{2.12}$$

Where we define

$$\mathbf{W}\mathbf{x} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \end{bmatrix} \mathbf{x} \tag{2.13}$$

The loss function in this case is

$$L(\mathbf{y}, f(\mathbf{x}, \mathbf{W})) = \sum_{i=1}^{3} y_i \log p_i \tag{2.14}$$

In our example, because we know in advance that all the elements of $\mathbf{y}$ is 0 except the first one, so the Equation 2.14 can be reduced to

$$L(\mathbf{y}, f(\mathbf{x}, \mathbf{W})) = y_1 \log p_1 \tag{2.15}$$

If the output we obtain from $f(\mathbf{x}, \mathbf{W})$ is

$$f(\mathbf{x}, \mathbf{W}) = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \end{bmatrix} = \begin{bmatrix} 0.3 \\ 0.5 \\ 0.2 \end{bmatrix} \tag{2.16}$$

We can make our prediction $\mathbf{y}'$ by convert the element with maximum probability p_i to 1, and others to 0.

$$\mathbf{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \tag{2.17}$$

2.2.2 Gradient Based Learning

Once we have a training set $D = \{\mathbf{x}_{1:n}, \mathbf{y}_{1:n}\}$, the neural network $f(\mathbf{x}, \theta))$ and loss function $L(\mathbf{y}, f(\mathbf{x}, \theta))$, we are facing the problem of learning optimal parameters θ^*

to minimise loss function.

$$\theta^* = \arg\min_{\theta} L(\mathbf{y}, f(\mathbf{x}, \theta)) \tag{2.18}$$

The neural networks are trained to learn optimal parameters θ^* by iterative gradient based algorithms. For example, the **Gradient Descent** (GD)

Algorithm 1: GD for finding θ^*

initialise γ
randomly initialise $\theta \leftarrow \theta_0$
while *Stop criteria not meet* **do**
    $\theta \leftarrow \theta - \frac{\gamma}{|D|} \sum_{i=1}^{|D|} \bigtriangledown_\theta L(\mathbf{y}_i, f(\mathbf{x}_i, \theta))$
end
$\theta^* \leftarrow \theta$

The γ is known as learning rate, which controls the speed of gradients update. $|D|$ is the size of the training dataset. The stop criteria can vary with different training strategies, one simple choice of stop criteria is to limit the maximum number of iterations.

Algorithm 2: Basic SGD algorithm for finding θ^*

initialize γ
initialize $\theta \leftarrow \theta_0$
while *Stop criteria not meet* **do**
    Sample a minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_s, \mathbf{y}_s)\}$ from D
    $\mathbf{g} \leftarrow \frac{1}{s} \sum_{i=1}^{s} \bigtriangledown_\theta L(\mathbf{y}_i, f(\mathbf{x}_i, \theta))$
    $\theta \leftarrow \theta - \gamma \mathbf{g}$
end
$\theta^* \leftarrow \theta$

Apply GD has to iterate through the whole training dataset for single step gradients update when the size of training dataset is large GD can be very computationally expensive. Unlike GD, **Stochastic Gradient Descent** (SGD) computes the approximated gradients based on a random minibatch (a subset) sampled from training data D, and it terms out that SGD tends to work better than GD on a large dataset[20].

There are many extended versions of SGD algorithms which can result in better and faster convergence, readers who are interested in can refer to [21, 22].

In above discussion, we only introduce different gradient based algorithms for parameters update. To apply these algorithms, we need a mechanism that can compute gradients. Here we briefly introduce the **back propagation** algorithm[23] which can efficiently compute exact gradients for any differentiable functions. The back propagation algorithm is based on the chain rule of calculus, let x be a scalar real number, and suppose we have $y = g(x)$ and $z = f(y)$. Then the chain rule gives that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{2.19}$$

Now consider the following function

$$f(x_1, x_2) = \cos(\exp((x_1 x_2)) \tag{2.20}$$

Figure 2.2a gives a graph illustrate of this function. Refer to Figure 2.2a, to calculate the derivative of $f(\cdot)$ with respected to x_1 and x_2, we need to evaluate

$$\frac{df}{dx_1} = \frac{df_3}{df_2}\frac{df_2}{df_1}\frac{df_1}{dx_1} \tag{2.21}$$

and

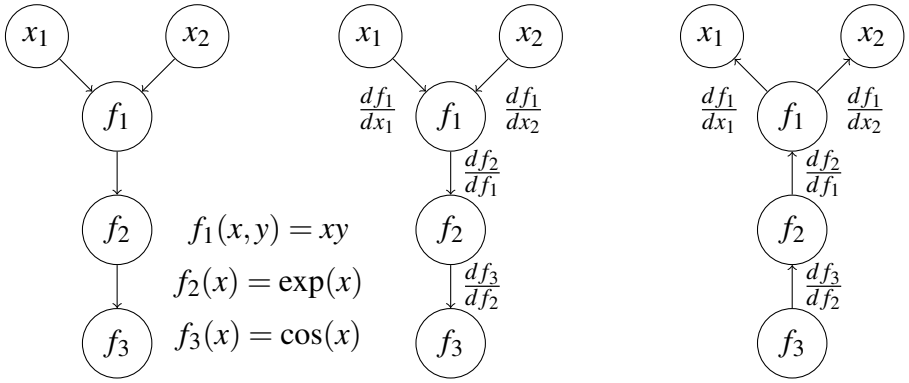
$$\frac{df}{dx_2} = \frac{df_3}{df_2}\frac{df_2}{df_1}\frac{df_1}{dx_2} \tag{2.22}$$

We could compute $\frac{df}{dx_1}$ and $\frac{df}{dx_2}$ by forward propagating (Figure 2.2b) or backward propagating (Figure 2.2c) the intermediate derivatives. We can notice that there is a shared path between $\frac{df}{dx_1}$ and $\frac{df}{dx_2}$ which is

$$\frac{df_3}{df_1} = \frac{df_3}{df_2}\frac{df_2}{df_1} \tag{2.23}$$

By doing the backward gradient propagation, we do not need to recompute $\frac{df_3}{df_1}$,

while forward propagation has to. This makes the forward propagation requires 2 more evaluations to work out $\frac{df}{dx_1}$ and $\frac{df}{dx_2}$ compares with back propagation in our example. Research suggests that the complexity of back propagation is approximately equal to the complexity of evaluating $f(\cdot)$, while the complexity of calculating derivatives forward is n times greater[24], where n is the numbers of inputs of $f(\cdot)$.



(a) The computational graph for evaluating $f(\cdot)$

(b) The forward propagation of derivatives

(c) The backward propagation of derivatives

Figure 2.2: An diagram illustration of computing the value of $f(x_1, x_2)$ and its derivatives, we can see that, to calculate the result of $f(\cdot)$, we need 3 evaluation steps (Figure 2.2a). And for the gradients $\frac{df}{dx_1}$ and $\frac{df}{dx_2}$, we need 6 evaluations for forward propagation (Figure 2.2b) and 4 evaluations for backward propagation (Figure 2.2c).

The above example only shows the basic idea of back propagation algorithm. There are many deep learning packages for example Theano[25] and Tensorflow[26], implemented back-propagation algorithm to compute gradients for training neural networks.

2.3 Regularisation

When training a neural network, we usually incur the problem of **overfitting**. There are many regularisation methods explicitly designed to reduce overfitting during training time. Here we introduce two methods we will use in this thesis.

L^2 **regularisation** which also known as weight decay, is the simplest and most commonly used regularisation method. Assume that θ stands for the parameters of

the neural network, and the loss function is L_θ. To perform the l^2 regularisation, we add l^2 norm of θ and loss function, then minimise them together with respect to θ

$$J = L_\theta + \frac{\alpha}{2}||\theta||_2^2 \tag{2.24}$$

Where α controls the importance of regularisation term.

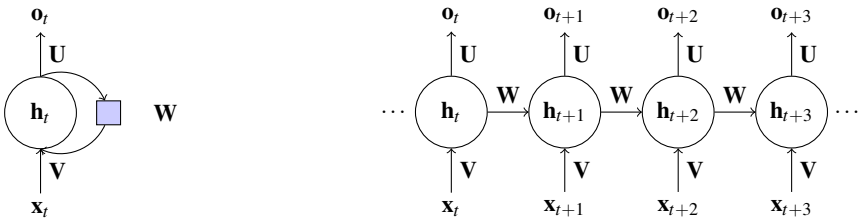
Algorithm 3: SGD with early stopping

early stop trigger p
updating step n
initialize γ
integer counter $j \leftarrow 0$
optimal parameters $\theta^* \leftarrow \mathbf{0}$
initialize $v \leftarrow \infty$
initialize $\theta \leftarrow \theta_0$
while $j < p$ **do**
    for 0 *to* n **do**
        Sample a minibatch $\{(\mathbf{x}_1, \mathbf{y}_1), ..., (\mathbf{x}_s, \mathbf{y}_s)\}$ from D
        $\mathbf{g} \leftarrow \frac{1}{s}\sum_{i=1}^{s} \bigtriangledown_\theta L(\mathbf{y}_i, f(\mathbf{x}_i, \theta))$
        $\theta \leftarrow \theta - \gamma\mathbf{g}$
    end
    Get validation set V
    $v' \leftarrow \frac{1}{|V|}\sum_{(\mathbf{x}_i, \mathbf{y}_i \in V)} L(\mathbf{y}_i, f(\mathbf{x}_i, \theta))$
    $j++$
    if $v' < v$ **then**
        $j \leftarrow 0$
        $v \leftarrow v'$
        $\theta^* \leftarrow \theta$
    end
end
return θ^*

When training a neural network, it is common to use a validation set to estimate generalisation error during training. If the neural network starts to overfit, we usually observe that training error decreases steadily over time, while the error on validation set starts to rise again. This means that we can obtain θ with better generalisation error by storing θ at the point with the lowest validation error, and hope this θ can also perform well on testing set. This strategy is known as **early stopping**.

When applying the early stopping, our stop criteria depends on validation loss. In the given example (Algorithm 3), we use an integer counter j to record the number of times that validation loss increases. j is always set to 0 if the validation loss is decreasing. When j is greater or equal to our maximum number of allowance p, we halt the algorithm and return θ[20].

2.4 Recurrent Neural Networks



(a) A folded RNN update graph (b) An unfolded update graph of RNN

Figure 2.3: A simple diagram example of RNN architecture, the parameters $\mathbf{U}, \mathbf{V}, \mathbf{W}$ are shared across the model. At each time step, the hidden state is computed based on the current input and previous hidden state

Recurrent neural networks (RNNs) are neural networks that can process sequential data. An important property of RNNs is that the parameters are shared across the model[8]. This makes RNN possible to take input sequences with arbitrary length and generalise across them. Here we give an example of a simple RNN, which contains a single and self-connected hidden layer.

$$\mathbf{h}_t = \tanh(\mathbf{V}\mathbf{x}_t + \mathbf{W}\mathbf{h}_{t-1}) \tag{2.25}$$

$$\mathbf{o}_t = \text{softmax}(\mathbf{U}\mathbf{h}_t) \tag{2.26}$$

A convenient way to visualise RNN is to consider the unfolded parameter update graph of the network along with the input sequence, Figure 2.3b shows part of an unfolded RNN[20]. We can see that, each hidden state $\mathbf{h}_t$ can capture the information from past $\mathbf{h}_{t-1}$ and present input $\mathbf{x}_t$. So we can consider the last hidden as a "summary" of input sequence.

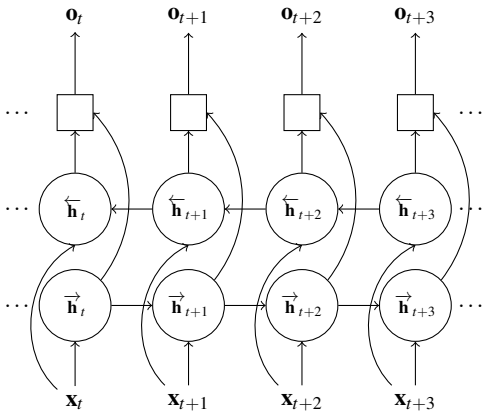


Figure 2.4: A digram illustration of Bidirectional RNN, in this diagram the square box represents any transformation that can map forward and backward information into output.

In some situations, we may also want $\mathbf{h}_t$ to have information about future. The Bidirectional RNN[27] is proposed to address this need. Acorrding to Figure 2.4, the bottom RNN encodes past information into hidden state $\overrightarrow{\mathbf{h}}_t$, while the top RNN encodes future information hidden state $\overleftarrow{\mathbf{h}}_t$. The output at each time step is computed based on hidden states of both RNNs.

2.4.1 Back Propagation Through Time

A common approach to determine the derivatives of RNN is Backpropagation Through Time (BPTT)[28]. Similar to standard back propagation, BPTT consists of a repeated application of chain rule. The difference is that to obtain correct gradients update of hidden layer parameters, we have to propagate the gradient all the way back to initial time step, as shown in Figure 2.5. To be more specific, assume that we have an RNN based on Equation 2.25 and Equation 2.26, the loss at time step t is $L_t(\mathbf{y}_t, \mathbf{o}_t)$ and we wish to update our parameter just base on $L_t(\mathbf{y}_t, \mathbf{o}_t)$. From

Equation 2.25 we can notice that, there is a recursive dependency between each hidden states and $\mathbf{W}, \mathbf{V}$. So when doing the gradients update, we have to work out gradients at each time step, and sum all corresponding gradients to obtain correct update, here we only show the update of $\mathbf{W}$ as example

$$\frac{\partial L_t}{\partial \mathbf{W}} = \sum_{i=1}^{t} \frac{\partial L_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_t} \Big(\prod_{j=i+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}} \Big) \frac{\partial \mathbf{o}_i}{\partial \mathbf{W}} \tag{2.27}$$

From Equation 2.27 we can find that the hidden state Jacobian matrices are accumulating through time

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_i} = \prod_{j=i+1}^{t} \frac{\partial \mathbf{h}_j}{\partial \mathbf{h}_{j-1}}$$

With t goes large ($t \to \infty$), $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_i}$ can quickly shrink to zero or explode to infinity[29]. This can disrupt gradient update and results in the problem of learning long-term dependencies in RNN[10].

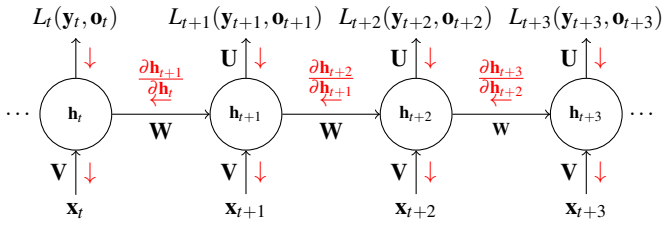


Figure 2.5: A diagram illustration of BPTT on RNN, we can see that the partial derivatives between hidden states are accumulating back through time. As shown in this diagram, because the parameters of RNN are shared across, therefore in order to compute correct gradients update, we have to back propagate derivatives all the way back to initial time step.

2.4.2 Gated RNNs

Using the gated RNNs is one effective way that can reduce the difficulty of learning long-term dependencies. Gated RNNs produce paths through time that have derivatives neither vanish nor explode. An example of gated RNNs is Gated Recurrent Units (GRU)[12], in GRU the update equations for hidden unit $\mathbf{h}_t$ are

$$\mathbf{h}_t = \mathbf{u}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{u}_t) \odot \tilde{\mathbf{h}}_t \tag{2.28}$$

Where $\mathbf{r}_t$ stands for "reset" gate and $\mathbf{u}_t$ for "update" gate, $\tilde{\mathbf{h}}_t$ is the candidate hidden state

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}\mathbf{x}_t + \mathbf{U}\left[\mathbf{r}_t \odot \mathbf{h}_{t-1}\right] + b_h) \tag{2.29}$$

$$\mathbf{u}_t = \sigma(\mathbf{W}_u\mathbf{x}_t + \mathbf{U}_u\mathbf{h}_{t-1} + b_u) \tag{2.30}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r\mathbf{x}_t + \mathbf{U}_r\mathbf{h}_{t-1} + b_r) \tag{2.31}$$

The reset gate can control information from past used to compute next target state, while update gate can "copy" the past or "ignore" it by replace it new target state.

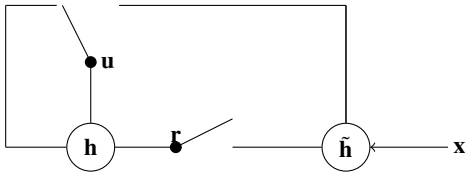


Figure 2.6: A simplified diagram illustration of GRU unit the update of $\mathbf{h}$,$\tilde{\mathbf{h}}$ is controlled by $\mathbf{r}$ and $\mathbf{u}$.

2.5 Autoenocder

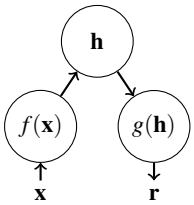


Figure 2.7: The diagram shows the architecture of autoencoder, $f(\cdot)$ encodes $\mathbf{x}$ into hidden representation $\mathbf{h}$, the $g(\cdot)$ decodes $\mathbf{h}$ to $\mathbf{r}$. $\mathbf{r}$ can be consider as the reconstruction of $\mathbf{x}$.

An autoencoder[30] is a neural network that attempts to "reconstruct" its input as its output. The autoencoder can be separated into two parts, an encoder $\mathbf{h} = f(\mathbf{x})$ and a decoder $\mathbf{r} = g(\mathbf{h})$, where $\mathbf{h}$ is usually considered as the hidden representation of input $\mathbf{x}$, and $\mathbf{r}$ is the reconstruction of $\mathbf{x}$.

Autoencoder offers us a way to perform unsupervised learning on neural network. To be more specific, we can first train the autoencoder on unlabelled data to acquire

some information about inputs. This is known as **unsupervised pretraining**. Then we take the encoder of autoencoder to perform some supervised learning tasks. We can view this as a form of parameter initialization for supervised learning tasks[31].

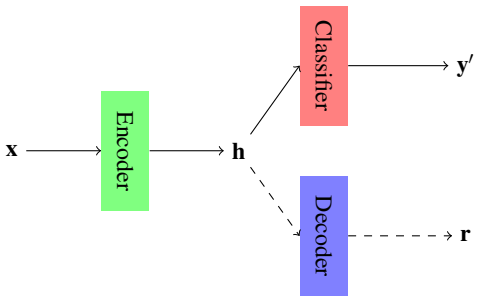


Figure 2.8: Unsupervised pretaining by using autoencoder, the encoder is first pretrained as part of autoencoder (dashed line), and then tuned with a classifier (solid line).

2.6 Neural Language Models

Neural language models (**NLMs**) are members of neural networks for language modelling. NLMs gained attentions after Bengio et al [32] used a feedforward neural network to learn the representations for words in 2003. However, until recently, it was not possible to train an NLM efficiently on big amounts of data, because of hardware limitations, in Bengio et al[32] also reported that they spend three weeks in training their model on forty CPUs for only five epochs. With the recent advance in hardware development, researchers can train their neural network much more efficient on GPU.

An NLM usually has following components in their model architecture[33, 6]

1. An embedding layer that allows NLM to perform index lookup and find corresponded feature vectors for the given input words. The technique used to acquire feature vectors will be presented soon. We will introduce how to create these indexes and embedding layer in our experiments later in section 4.1.2 and 4.1.3.

2. An encoding neural network which takes the look-upped feature vectors as inputs and encodes them into intermediate hidden representations. Then the intermediate hidden representations are passed to an output classifier and perform a certain task.

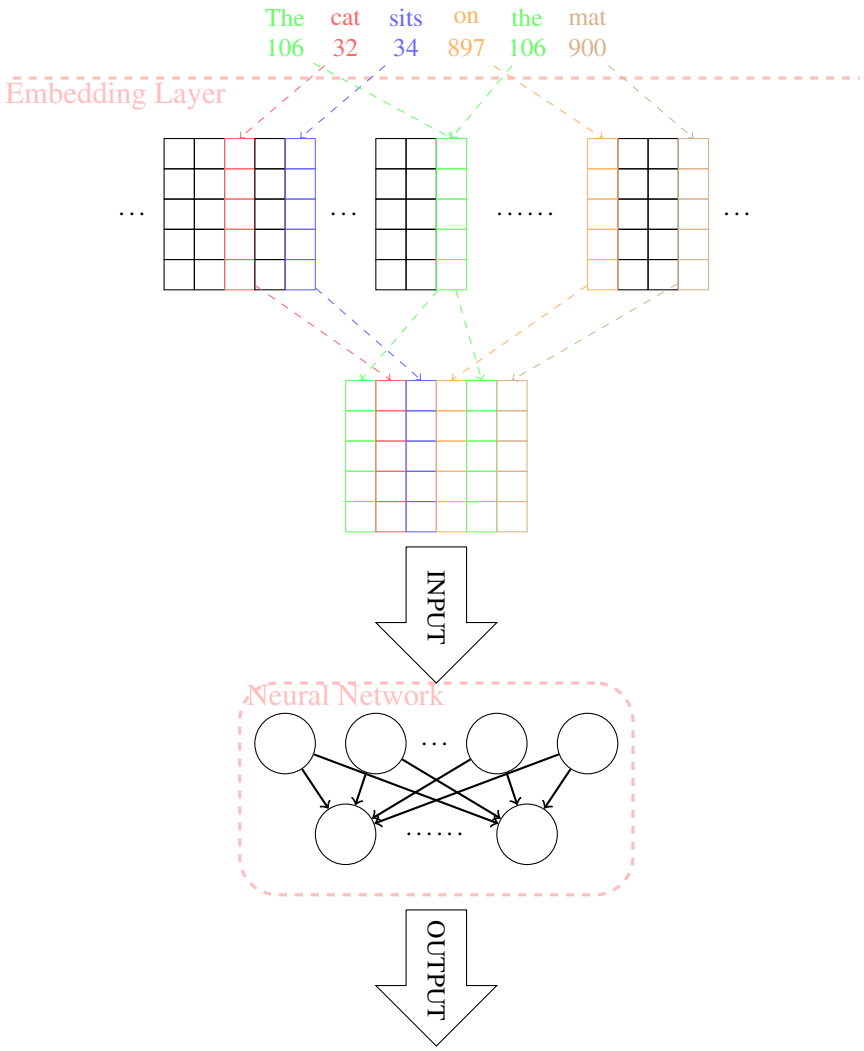


Figure 2.9: Architecture of NLM's ,the word feature vectors are first retrieved from embedding layer by index, then the feature vectors are fed into neural network to perform certain task.

2.6.1 Word Level Neural Language Model

Learning the vector representations for words from unlabelled data is one of the most successful applications of NLMs. There are some different word level NLMs have been proposed over the years[2, 4, 34]. Here we only briefly introduce **Con-**

tinuous Bags of Words (CBOW) proposed by Mikolov et al[3].

The CBOW works as follow, given a small corpus " There is a king lives in castle", We first produce a central word and context pair $(\text{king}, \{\text{ther is a lives in castle}\})$ and then give a one-hot vector to each word according to their orders in window

$$\text{there} = \mathbf{v}_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{is} = \mathbf{v}_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \dots \quad \text{castle} = \mathbf{v}_7 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.32}$$

We randomly initialise a matrix $\mathbf{W} \in \mathbb{R}^{7\times n}$ with in range $[-1,1]$, each word can obtain their corresponded word vector $\mathbf{w}_i$ by

$$\mathbf{w}_i = \mathbf{v}_i^T \mathbf{W} \tag{2.33}$$

Now we can set up a feedforward neural network takes $\mathbf{C} = \{\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, \mathbf{w}_5, \mathbf{w}_6, \mathbf{w}_7\}$ as inputs, the first layer calculates the average $\mathbf{z}$ over $\mathbf{C}$

$$\mathbf{z} = \frac{1}{6} \sum_{w_i \in \mathbf{C}} \mathbf{w}_i \tag{2.34}$$

The second layer of the neural network covert $\mathbf{z}$ into a seven dimensional vector $\mathbf{u}$

$$\mathbf{u} = \mathbf{z}^T \mathbf{U} \tag{2.35}$$

The output layer is a softmax function computes the conditional probability

$$p(\mathbf{w}_i|\mathbf{C}) = \frac{\exp(u_i)}{\sum_{j=1}^{7} \exp(u_j)} \tag{2.36}$$

Where $\mathbf{w}_i \in \{\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3, \mathbf{w}_4, \mathbf{w}_5, \mathbf{w}_6, \mathbf{w}_7\}$, u_i, u_j are elements of $\mathbf{u}$. Our target output is the central word "king", therefore the loss function is defined as

$$L = -\sum_{i=1}^{7} v_4^i p(\mathbf{w}_i|\mathbf{C}) \tag{2.37}$$

Where v_4^i is ith element of $\mathbf{v}_4$.

CBOW is incorporated with word2vec tool[1], which we will use to acquire word feature vectors for our experiments later.

2.6.2 Sentence Level Neural Language Model

Unlike word level NLMs, it is still not clear about which architectures or objectives can help us to learn the most useful sentences representations. Nevertheless, there are still a few NLMs that can learn representations for sentences. For example

- **Recursive Autoencoder (RAE)**[36] is an autoencoder constructed based on the syntactic tree of a given input sentence. Each node on the RAE encodes their children's output and provides the encoded vector to their parent's node. We take the output of root node as the final representation of the sentence. RAE can acquire more syntactical information about the sentence, as it is built from syntax tree. But, to obtain high-quality syntax trees tremendous amount of pre-processing work is usually required.

- **Sequence to Sequence (seq2seq) Model**[14, 1] consists of an encoding and a decoding RNN. The encoding RNN tries to encode intermediate representations from input data, while the decoding RNN produces sequences conditioned on the intermediate representations. Many applications of seq2seq model have observed that the learnt internal representations of seq2seq model can reflect some semantic meaning of input sentences.

- **Skip Through Vectors**[37] can be consider as an extension of seq2seq model, given the consecutive sentences S_{i-1}, S_i, S_{i+1}, the Skip Through model en-

[1] https://code.google.com/archive/p/word2vec/

codes S_i into an intermediate representation, and then predict S_{i-1}, S_{i+1} as outputs. Training the Skip through vectors model must be able to access contextual information from consecutive sentences.

We can see that above models are developed based on different concepts. A systematic comparison between these models has been made by Hill et al[7].

In this thesis, our primal interest is to study how different objectives can affect the sentence representations learnt by NLMs. We do our experiments on seq2seq model mainly because of its ability to acquire representations for sentences without additional syntactic and contextual information. In next chapter, We will discuss seq2seq model and how we set up this model to learn sentence representations in detail.

Chapter 3

Model and Experiment settings

In this chapter we will describe the seq2seq model[14, 1] in detail, we also introduce different objective functions for training seq2seq model and evaluation metrics we will use to carry out our analysis.

3.1 Sequence to Sequence Model

Seq2seq model[14, 1] emerges as an effective paradigm for dealing with variable length inputs and outputs. The model consists of an encoding and a decoding RNNs, the encoding RNN processes input sequence $\{\mathbf{x_{1:n}}\}$ to emit a fixed size context vector $\mathbf{c}$. Then the decoding RNN generate a output sequence conditional on the context vector $\mathbf{c}$[20].

3.1.1 Encoding RNN

We first feed input sentence $\mathbf{x_{1:n}}$ to a bidirectional RNN[27] with GRU[12] to obtain forward information $\overrightarrow{\mathbf{h}}_{1:n}$ and backward information $\overleftarrow{\mathbf{h}}_{1:n}$:

$$\overrightarrow{\mathbf{h}}_i = \begin{cases} (1-\overrightarrow{\mathbf{z}}_i)\odot\overrightarrow{\mathbf{h}}_{i-1} + \overrightarrow{\mathbf{z}}_i\odot\overrightarrow{\mathbf{h}}'_i & \text{if i } >0 \\ 0 & otherwise \end{cases} \tag{3.1}$$

Where

$$\overrightarrow{\mathbf{h_i}}' = \tanh(\overrightarrow{\mathbf{W}}\mathbf{x}_i + \overrightarrow{\mathbf{U}}\left[\overrightarrow{\mathbf{r}}_i\odot\overrightarrow{\mathbf{h}}_{i-1}\right]) \tag{3.2}$$

$$\overrightarrow{\mathbf{z}}_i = \sigma(\overrightarrow{\mathbf{W}}_z\mathbf{x_i} + \overrightarrow{\mathbf{U}}_z\overrightarrow{\mathbf{h}}_{i-1}) \tag{3.3}$$

$$\overrightarrow{\mathbf{r}}_i = \sigma(\overrightarrow{\mathbf{W}}_r \mathbf{x_i} + \overrightarrow{\mathbf{U}}_r \overrightarrow{\mathbf{h}}_{i-1}) \tag{3.4}$$

$\overrightarrow{\mathbf{W}}, \overrightarrow{\mathbf{W}}_z, \overrightarrow{\mathbf{W}}_r$ and $\overrightarrow{\mathbf{U}}, \overrightarrow{\mathbf{U}}_z, \overrightarrow{\mathbf{U}}_r$ are weight matrices, $\sigma(\cdot)$ is a sigmoid function. Note that the backward hidden states are computed similarly, with different sets of weight matrices. The outputs of bidirectional RNN is the concatenate of forward and backward information.

$$\mathbf{h_i} = \begin{bmatrix} \overrightarrow{\mathbf{h}}_i \\ \overleftarrow{\mathbf{h}}_i \end{bmatrix} \tag{3.5}$$

The $\mathbf{h_{1:n}}$ are encoded by another RNN with GRU, and we take the final output of this RNN as the fixed context representation $\mathbf{c}$ of input sentence.

3.1.2 Decoding RNN

The decoding RNN computes outputs $\mathbf{s}_{1:m}$ as following

$$\mathbf{s_j} = \begin{cases} (1 - \mathbf{z_j}) \odot \mathbf{s_{j-1}} + \mathbf{z_j} \odot \bar{\mathbf{s}}_\mathbf{j} & \text{j} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.6}$$

Where $\mathbf{s}_j$ is calculated conditional on $\mathbf{c}$

$$\bar{\mathbf{s}}_\mathbf{j} = \tanh(\mathbf{W x_{j-1}} + \mathbf{r_j} \odot \left[\mathbf{U s_{j-1}} + \mathbf{V c}\right]) \tag{3.7}$$

$$\mathbf{z_j} = \sigma(\mathbf{W_z x_{j-1}} + \mathbf{U_z s_{j-1}} + \mathbf{V_z c}) \tag{3.8}$$

$$\mathbf{r_j} = \sigma(\mathbf{W_r x_{j-1}} + \mathbf{U_r s_{j-1}} + \mathbf{V_r c}) \tag{3.9}$$

Then with the target output sequence $\mathbf{y}_{1:m}$, we can compute the output conditional probability[1]

$$p_\theta(\mathbf{y}_{1:m} | \mathbf{x_{1:n}}) = \prod_{j=1}^{m} p_\theta(\mathbf{y}_j | \mathbf{y}_{j-1}, \mathbf{c}) \tag{3.10}$$

$$p_\theta(\mathbf{y}_j | \mathbf{y}_{j-1}, \mathbf{c}) = \frac{\exp(\mathbf{y}_j^T \mathbf{W_p e_j})}{\sum_{\mathbf{y}_k \in \mathbf{V}} \exp(\mathbf{y}_k^T \mathbf{W_p e_i})} \tag{3.11}$$

$$\mathbf{e_i} = \mathbf{O_e s_{i-1}} + \mathbf{O_y y_{i-1}} + \mathbf{O_c c} \tag{3.12}$$

Where θ indicate all the parameters of our seq2seq model, $\mathbf{V}$ is feature vectors of full vocabulary set. The negative log likelihood is considered as the loss function of seq2seq model

$$L_{rec} = -\sum_{j=1}^{m} \log p_\theta(\mathbf{y}_j|\mathbf{y}_{j-1}, \mathbf{c}) \tag{3.13}$$

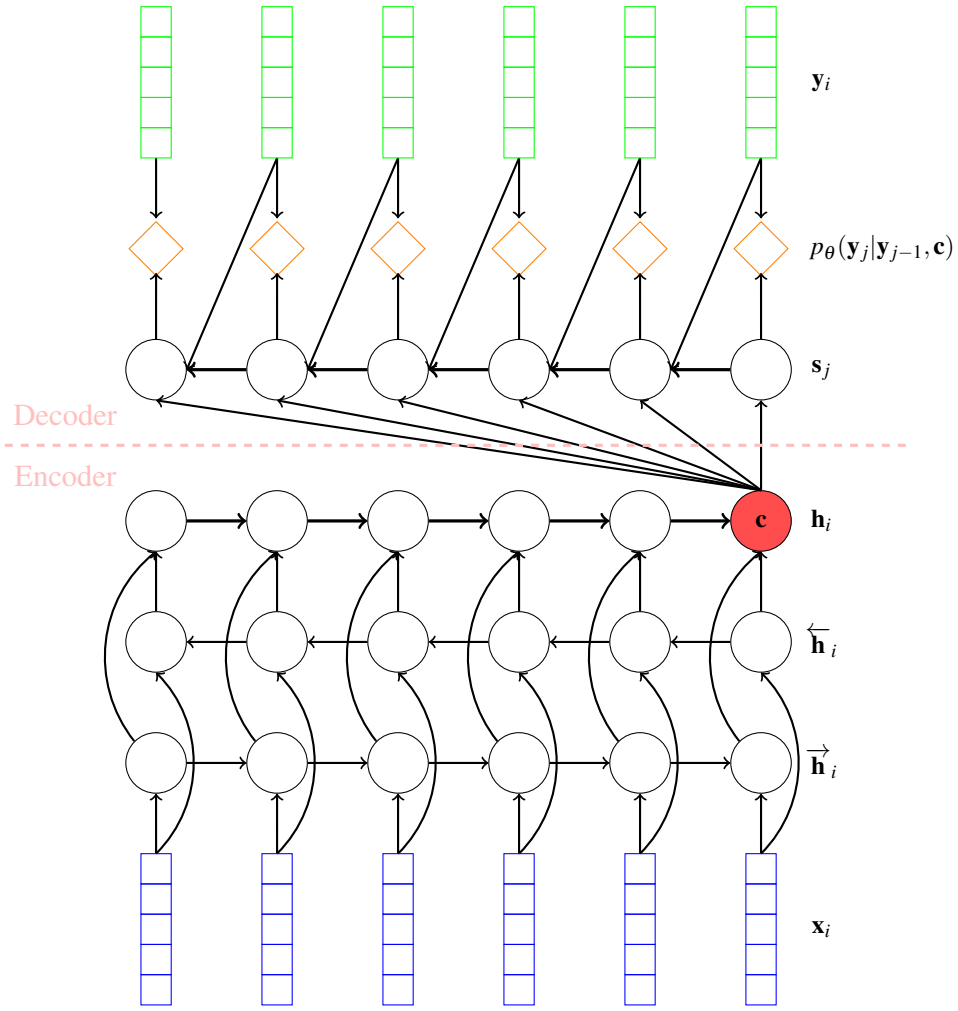


Figure 3.1: Seq2seq model with bidirectional RNN encoder [1], the input word feature vectors are first encoded by a bidirectional RNN into fixed context vector $\mathbf{c}$,then the decoder calculate the probability of decoding sequence conditional on this context vector $\mathbf{c}$.

3.2 Loss Estimation

Training the seq2seq on large vocabulary can incur the problem of intractable computation[38]. Let us consider the gradient of Equation 3.13. For convenient

we let $f_\theta(\mathbf{y},\mathbf{e}) = \mathbf{y}^T\mathbf{W_p}\mathbf{e}$,the gradient can be composed as follow:

$$g_\theta = \bigtriangledown_\theta f_\theta(\mathbf{y_j},\mathbf{e_j}) - \sum_{\mathbf{y_k}\in\mathbf{V}} \frac{\exp(f_\theta(\mathbf{y_k},\mathbf{e_j}))}{Z} \bigtriangledown_\theta f_\theta(\mathbf{y_k},\mathbf{e_j}) \tag{3.14}$$

Where $Z = \sum_{\mathbf{y_k}\in\mathbf{V}} \exp(f_\theta(\mathbf{y_k},\mathbf{e_j}))$. When the size of $\mathbf{V}$ becomes large, the normaliser Z quickly becomes intractable[39]. A common way to address this issue is to approximate this term by importance sampling[40]:

$$Z = \sum_{\mathbf{y_k}\in\mathbf{V}} q(\mathbf{y_k}) \frac{\exp(f_\theta(\mathbf{y_k},\mathbf{e_j}))}{q(\mathbf{y_k})} \approx \frac{1}{S}\sum_{s=1}^{S} \frac{\exp(f_\theta(\mathbf{y_s},\mathbf{e_j}))}{q(\mathbf{y_s})} \tag{3.15}$$

Where $\mathbf{y}_s$ is the sample drawn from the proposed importance distribution $q(\mathbf{y})$. We can predefine a small subset of vocabulary $\mathbf{V}_s \subset \mathbf{V}$ with size s, and assume the distribution $q(\mathbf{y})$ has the form[41]:

$$q(\mathbf{y}) = \begin{cases} \frac{1}{S} & \text{if } \mathbf{y} \in \mathbf{V}_s \\ 0 & \text{otherwise} \end{cases} \tag{3.16}$$

Under this assumption we can compute an approximated normalizer:

$$Z \approx \sum_{s=1}^{S} \exp(f_\theta(\mathbf{y_s},\mathbf{e_j})) \tag{3.17}$$

By using this approach, we can estimate the value of both gradient g_θ and likelihood function $p_\theta(\mathbf{y_j}|\mathbf{y_{j-1}},\mathbf{c})$ with a smaller subset of $\mathbf{V_s}$. Which can reduce the computational cost during training.

3.3 Finding the Prediction

Unlike training time the target output sequences are available to model, at testing time seq2seq model must be able to predict the appropriate output sequences[14], this can be done via

$$\hat{\mathbf{y}}_{1:m} = \arg\max_{\mathbf{y_{1:m}}\in\mathbf{V}} p(\mathbf{y_{1:m}}|\mathbf{x_{1:n}}) = \arg\max_{\mathbf{y_{1:m}}\in\mathbf{V}} \prod_{t=1}^{m} p(\mathbf{y_t}|\mathbf{y_{t-1}},\mathbf{c}) \tag{3.18}$$

Viterbi algorithm can solve this problem with $O(m \times |\mathbf{V}|^2)$ complexity[42]

$$\alpha(\mathbf{y}_0) = 1 \tag{3.19}$$

$$\alpha(\mathbf{y}_t) = \max_{\mathbf{y}_{t-1} \in \mathbf{V}} \alpha(\mathbf{y}_{t-1}) p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{c}) \tag{3.20}$$

But with the size of vocabularies grows, the computational cost for decoding will become very expensive. A much efficient approach is to apply Beam search with Viterbi decoder[43].

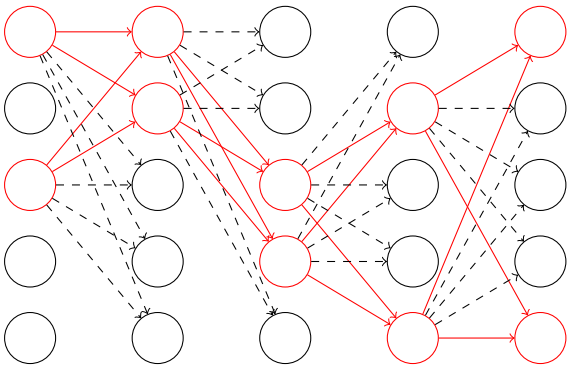


Figure 3.2: Viterbi algorithm with Beam search with $k = 2$. Only top 2 paths are kept in candidate set are explored (indicate by red color), other paths are discarded (indicate by dashed line).

Beam search is a heuristic search algorithm, which explores the search space by expanding a set of most promising candidates with size k[44]. In our application is to rank all alternative candidate words at each time point and store top k words in candidate set $\mathbf{V}_k$ for next evaluation. This changes Equation 3.20 the original Viterbi algorithm to

$$\alpha(\mathbf{y}_t) \approx \max_{\mathbf{y}_{t-1} \in \mathbf{V}_k} \alpha(\mathbf{y}_{t-1}) p(\mathbf{y}_t | \mathbf{y}_{t-1}, \mathbf{c}) \tag{3.21}$$

Figure 3.2 shows an example of Viterbi algorithm with Beam search applied.

The Viterbi Beam decoding can solve the problem with $O(m \times k \times |\mathbf{V}|)$ complexity. And the most probable sequence can be compute by backtracking. However beam search may not be able to find the best output sequences, as the vocabularies are not

fully searched at each time steps. So there is always a trade off between the beam search efficiency and accuracy. For the sequence to sequence model, Sutskever et al [14] suggested a beam size of 2 suffices to provide most of benefits of beam search.

3.4 Objective Functions

Here we introduce different objective functions used to train our seq2seq models.

3.4.1 Unsupervised Pretrain

We feed seq2seq model with English sentences and ask it to reconstruct input sentences as its output. This objective function is proposed by Dai et al [15].

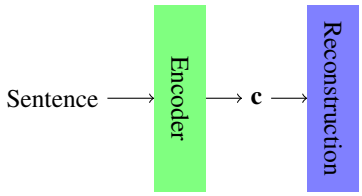


Figure 3.3: Unsupervised pretrain of seq2seq model. The seq2seq model is asked to reconstruction the input sentence as its output.

3.4.2 Semi Supervised Pretrain

In language recognition, some pairs of sentences $(\mathbf{x_{1:n}}, \bar{\mathbf{x}}_{\mathbf{1:m}})$ with different expression might have similar meaning. This means, their learnt internal fixed context representation $(\mathbf{c}, \bar{\mathbf{c}})$ suppose to be close under some measurements.

We provide a limited set of sentence's pairs with similar meaning to sequential autoencoder, and we define a constrained loss L' as

$$L' = (\mathbf{c} - \bar{\mathbf{c}})^T (\mathbf{c} - \bar{\mathbf{c}}) \tag{3.22}$$

Therefore the overall loss L of this setting is

$$L = \alpha L_{rec} + L' \tag{3.23}$$

Where αL_{rec} is the penalised reconstruction loss. Zhang et al [45] try the similar idea with hinge loss on REA in machine translation, their observations show that the additional constraint can help model to learn closer representations for sentences with similar meaning.

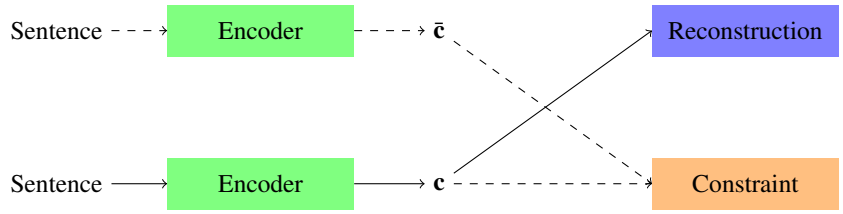


Figure 3.4: Semi supervised pretrain, the model is pretrained with constraint (indicate by dashed line).

3.4.3 Multiple Tasks Joint Training

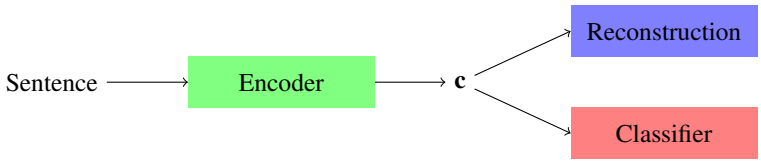


Figure 3.5: Joint training, seq2seq model is joint trained with another supervised task.

We joint train seq2seq model with another supervised task. In this thesis the candidate supervised task includes sentiment analysis, machine translation and sentence entailment. The training objective function is

$$L(\mathbf{X}, Y) = L_c + \alpha L_{rec} \tag{3.24}$$

Where αL_{rec} is the penalised reconstruction loss, L_c is the classification loss of the supervised task. Dong et al [46] and Luong et al[47] show that, multi-task training allows information shares across tasks, which will benefit the testing time performance.

3.5 Intrinsic Evaluation

In NLP, the intrinsic evaluation metric is one which measures the quality of a model independent of any application[16]. The intrinsic evaluation is easy to carry out and

usually used as a quick check of language models.

3.5.1 Euclidean Distance

We use Euclidean distance as one of the intrinsic metrics, to be more specific, we calculate the Euclidean distance between representations of some selected sentence pairs, to see whether the Euclidean distance is smaller for pairs with similar meaning compares with those pairs have opposite meaning.

3.5.2 Perplexity

We also compute the perplexity[48] of each experimental models, to evaluate how well they can assign probabilities to a sample prediction. The perplexity is calculate by

$$\text{perplexity}(\mathbf{y}_{1:m}) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{p(\mathbf{y}_i|\mathbf{y}_{i-1}, \mathbf{c})}} \tag{3.25}$$

When calculate perplexity we clip the value of $p(\mathbf{y}_i|\mathbf{y}_{i-1}, \mathbf{c})$ to be in between $[0.01, 1]$, so the the perplexity is within $[1, 100]$. Usually better models have lower perplexity.

3.6 Extrinsic Evaluation

Extrinsic evaluation is the evaluation on real tasks. To carry out this evaluation, we tune our pretrained RNN encoder on sentence entailment, sentiment analysis and machine translation. The testing time performance is taken as the extrinsic evaluation metrics.

3.6.1 Binary Sentiment Analysis

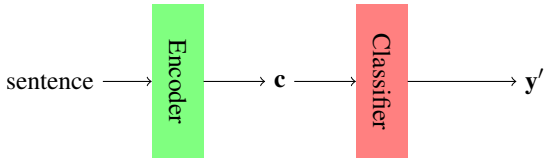


Figure 3.6: Sentiment analysis model, the sentence is first encoded into fixed context vector $\mathbf{c}$, and then pass to classifier to give predicted result $\mathbf{y}'$.

We pass the sentence representation $\mathbf{c}$ encoded by RNN encoder to a simple sigmoid

binary classifier,

$$p = \sigma(\mathbf{w}^T \mathbf{c} + b) \tag{3.26}$$

We use cross entropy loss as objective function

$$L_c = ylog(p) + (1-y)log(1-p) \tag{3.27}$$

At the testing time the prediction is given by

$$y' = \begin{cases} 0 & p > 0.5 \\ 1 & p \leq 0.5 \end{cases} \tag{3.28}$$

Where 1 indicate positive sentiment, 0 indicate negative sentiment.

3.6.2 Sentence Entailment

In sentence entailment task, the model has to figure out whether the input sentence pair $\mathbf{X} = \{\mathbf{x^p_{1:n}}, \mathbf{x^h_{1:m}}\}$ is entailed, natural or contradicted[49], these three relation are represent by different three dimensional one-hot vectors

$$\text{Entailment} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad \text{Natural} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \qquad \text{Contradict} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

The output layer of sentence entailment task is designed as follow. After the RNN sentence encoder encodes the context vector $\mathbf{c_p}$ and $\mathbf{c_h}$ from the input pair, we construct a intermediate information vector $\mathbf{c_{ph}}$ by following the suggestion from[50]

$$\mathbf{c_{ph}} = \begin{bmatrix} \mathbf{c_p} \\ \mathbf{c_p} - \mathbf{c_h} \\ \mathbf{c_p} \odot \mathbf{c_h} \\ \mathbf{c_h} \end{bmatrix} \tag{3.29}$$

Then the vector $\mathbf{c_{ph}}$ is then passed to softmax output layer to give a three dimensional output $\mathbf{p}$

$$\mathbf{p} = \text{softmax}(\mathbf{W}\mathbf{c_{ph}} + b) \tag{3.30}$$

The cross entropy loss is used as objective function

$$L_c = -\sum_{i=1}^{3} y_i log(p_i) \tag{3.31}$$

At testing time the prediction is worked out by convert $\mathbf{p}$ into a one-hot binary vector as discussed in **section 2.2.1**.

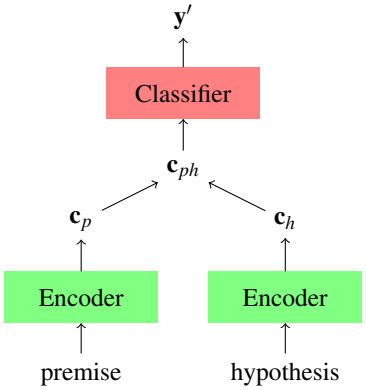


Figure 3.7: Sentence entailment model, a shared encoder (green) encodes premise and hypothesis into fixed context vector $\mathbf{c}_p$ and $\mathbf{c}_h$, and then produce the intermediate vector $\mathbf{c}_{ph}$, before give prediction $\mathbf{y}'$.

3.6.3 Machine Translation

We use the standard seq2seq model to solve this task, but note that the encoder is initialised with our pretrained RNN encoder. At testing time, the performance of machine translation is measured by bilingual evaluation understudy (BLEU) score[51], one of the common and cheap evaluation techniques used in machine translation.

The BLEU score is calculated as follow

$$w_n = \frac{1}{N}$$

$$p_n = \frac{\text{numbers of matched n-gram}}{\text{total numbers of n-gram in reference}}$$

$$\text{BLEU Score} = \exp\{\min(1-\frac{r}{c},0)\}(\prod_{n=1}^{N} p_n)^{w_n} \tag{3.32}$$

Where N is the size of n-gram up to, p_n is known as n-gram precision between candidate and reference sentence, r, c are the length of reference and candidate sentences, w_n, p_n is the weight and precision of nth gram. Here we show an example of BLEU score calculation with $N = 4$.

Figure 3.8: An example of BLEU score n-gram match

Metrics	Candidate One	Candidate Two
1-gram precision (p_1)	3/6	6/6
2-gram precision (p_2)	1/5	4/5
3-gram precision (p_3)	0/4	2/4
4-gram precision (p_4)	0/3	1/3
BLUE Score	0	0.51

Table 3.1: BLEU Score example

Bibliography

[1] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[2] T Mikolov and J Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 2013.

[3] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[4] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[5] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014.

[6] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[7] Felix Hill, Kyunghyun Cho, and Anna Korhonen. Learning distributed representations of sentences from unlabelled data. *arXiv preprint arXiv:1602.03483*, 2016.

[8] Jeffrey L Elman. Distributed representations, simple recurrent networks, and grammatical structure. *Machine learning*, 7(2-3):195–225, 1991.

[9] Ilya Sutskever. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

[10] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[12] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[13] Tomáš Mikolov. Statistical language models based on neural networks.

[14] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[15] Andrew M Dai and Quoc V Le. Semi-supervised sequence learning. In *Advances in Neural Information Processing Systems*, pages 3079–3087, 2015.

[16] Dan Jurafsky. *Speech & language processing*. Pearson Education India, 2000.

[17] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[18] Fritz Lehmann. Semantic networks. *Computers & Mathematics with Applications*, 23(2-5):1–50, 1992.

[19] Yoshua Bengio. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

[20] Ian Goodfellow Yoshua Bengio and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2016.

[21] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[22] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning.

[23] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

[24] Uwe Naumann. *The art of differentiating computer programs: an introduction to algorithmic differentiation*, volume 24. Siam, 2012.

[25] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.

[26] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[27] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[28] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[29] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks.

[30] Hervé Bourlard and Yves Kamp. Auto-association by multilayer perceptrons and singular value decomposition. *Biological cybernetics*, 59(4-5):291–294, 1988.

[31] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[32] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *journal of machine learning research*, 3(Feb):1137–1155, 2003.

[33] Ronan Collobert and Jason Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.

[34] Frederic Morin and Yoshua Bengio. Hierarchical probabilistic neural network language model. In *Aistats*, volume 5, pages 246–252. Citeseer, 2005.

[35] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. 2013.

[36] Richard Socher, Eric H Huang, Jeffrey Pennin, Christopher D Manning, and Andrew Y Ng. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *Advances in Neural Information Processing Systems*, pages 801–809, 2011.

[37] Ryan Kiros, Yukun Zhu, Ruslan R Salakhutdinov, Richard Zemel, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. Skip-thought vectors. In *Advances in neural information processing systems*, pages 3294–3302, 2015.

[38] Sébastien Jean Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation.

[39] Yoshua Bengio et al. Quick training of probabilistic neural nets by importance sampling.

[40] Kevin P Murphy. *Machine learning: a probabilistic perspective*. 2012.

[41] Yoshua Bengio and Jean-Sébastien Senécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

[42] Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE transactions on Information Theory*, 13(2):260–269, 1967.

[43] Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009.

[44] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach*, volume 2.

[45] Jiajun Zhang, Shujie Liu, Mu Li, Ming Zhou, Chengqing Zong, et al. Bilingually-constrained phrase embeddings for machine translation.

[46] Daxiang Dong, Hua Wu, Wei He, Dianhai Yu, and Haifeng Wang. Multi-task learning for multiple language translation.

[47] Minh-Thang Luong, Quoc V Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. *arXiv preprint arXiv:1511.06114*, 2015.

[48] Peter F Brown, Vincent J Della Pietra, Robert L Mercer, Stephen A Della Pietra, and Jennifer C Lai. An estimate of an upper bound for the entropy of english. *Computational Linguistics*, 18(1):31–40, 1992.

[49] Samuel R. Bowman, Gabor Angeli, Christopher Potts, and Christopher D. Manning. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2015.

[50] Lili Mou, Men Rui, Ge Li, Yan Xu, Lu Zhang, Rui Yan, and Zhi Jin. Recognizing entailment and contradiction by tree-based convolution. *arXiv preprint arXiv:1512.08422*, 2015.

[51] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[52] Bo Pang and Lillian Lee. Seeing stars: Exploiting class relationships for sentiment categorization with respect to rating scales. In *Proceedings of the 43rd annual meeting on association for computational linguistics*, pages 115–124. Association for Computational Linguistics, 2005.

[53] Philipp Koehn. Europarl: A parallel corpus for statistical machine translation. In *MT summit*, volume 5, pages 79–86, 2005.

[54] Yang Liu, Chengjie Sun, Lei Lin, and Xiaolong Wang. Learning natural language inference using bidirectional lstm model and inner-attention. *arXiv preprint arXiv:1605.09090*, 2016.

[55] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[56] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

[57] Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, and Yoshua Bengio. How to construct deep recurrent neural networks. *arXiv preprint arXiv:1312.6026*, 2013.