UNIVERSITÀ DEGLI STUDI DI TRENTO
CIMeC - Center for Mind/Brain Sciences

Master's Degree in Cognitive Science

Convolutional Neural Network Language Models

Marco Baroni ***Ngoc-Quan Pham***
Gemma Boleda

Academic Year 2014-2016

CONVOLUTIONAL NEURAL NETWORK LANGUAGE MODELS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF UNIVERSITY OF TRENTO
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Ngoc-Quan Pham
September 2016

Acknowledgements

My first gratitude is to Gemma Boleda, German Kruszewski and Marco Baroni, my thesis supervisors, who gave me full support for my master thesis, especially at the first stage when everything was a mess. With their constant guidance and encouragement, my personal competencies, from foreign languages to research skills have improved considerably. I would like to thank the members of the jury in University of Malta and University of Trento, for spending a lot of time and effort to review this thesis.

I also love the days working with the reading groups in CIMeC, where I had the fortune of benefiting from all of them. Marco Baroni paid my first lunch in Rovereto. Ducky (my Vietnamese fellow) and I tried to setup the computer for the whole day just to realise that the driver was impaired. Angeliki Lazaridou helped me to realise the beauty of learning models, especially how simple they are. In contrast, Aurelie Herbelot and Raffaella Bernardi showed me that the world is so naturally complex. There are also many other people in CIMeC that shared with me their stories and experience. Thanks to them, the horizon that I can see has been enlarged. The Erasmus Mundus scholarship and people who are involved to this program also helped me greatly in the master course and the thesis in particular.

I would like to thank Luong Chi Mai for introducing me to NLP and Speech Processing. Thanks to Le Hai Son for mentoring me about neural network language models and translation models. Thanks to Enrica and Martina for being my first Italian friends. Thanks to Volodimir Horowitz and Sergei Rachmaninoff, who are my endless source of inspiration with their planet-moving and heartfelt concerti.

Finally, my biggest gratitude is obviously to my parents and my brother who are my ultimate motivation to train myself harder and harder everyday in order to be a better human

being.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Exploiting the information from textual data (corpora) for artificial intelligent systems is one of the most important targets of Natural Language Processing (NLP). Applications that can benefit from this textual knowledge include Machine Translation, Information Extraction, Information Retrieval, Automatic Speech Recognition or Speech Synthesis .... The main applications of language models is to lead artificial intelligence systems to generate texts that can be comprehensible by human. For example, in speech recognition, a good language model should be distinguish two spoken utterances with similar signals: "Let music be the food of love" and "Let music be the foot of dove" because the former utterance is more intuitive and likely to occur in the current English.

The main purpose of a language model is to model the linguistic regularities of language by capturing the morphological, syntactical and semantic properties of a given language. Such a model should ideally be able to assess the likelihood of an arbitrary sentence in a particular context. In the last decades, the language modeling problem has been approached with statistical methods as a standard. Statistical language models are trained so as to predict the upcoming word given its previous context, which are the words appearing before the target word to be predicted. The word and its context is usually referred as an n-**gram**, which is a contiguous sequence of n tokens (words in textual data), consisting of one word to be predicted and $n-1$ words in the given context. For example, the word combination "The dog runs" is an n-gram with order of 3, and the language models aim at predicting the word "runs" given the context "the dog". The conventional approach

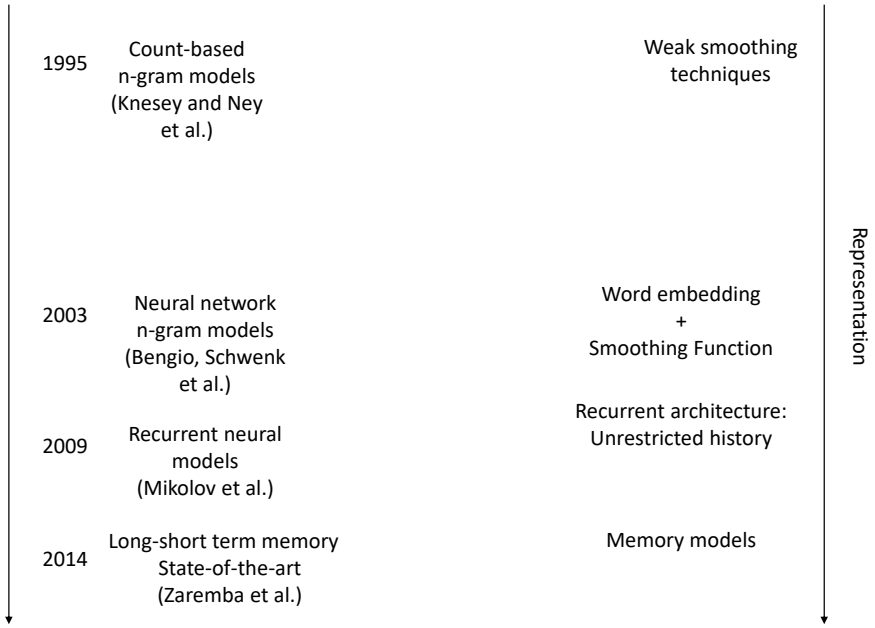

Figure 1.1: Language modeling development timeline.

to this problem relies on the statistical point of view, which is assigning probabilities for sentences by counting the number of occurrences of n-grams in the corpora to estimate the probabilities. Problems arise when the model has to estimate the probabilities of word sequences that are rare in the corpus, or do not appear at all, but still make sense in human languages. There are techniques that estimate the distribution of rare n-grams based on the seen ones (also referred to as smoothing techniques) or using the statistics of lower order n-grams (back-off techniques) [11, 18, 27, 37, 51, 72]. The main disadvantage of these models is that each individual word is treated independently of the others, that is, count-based n-gram models fail to capture semantic relations between words. As a result, those models suffer from two weaknesses. First, they are limited by a small context size, since the number of possible n-grams grows exponentially with the n-gram order. For example, if the language vocabulary contains 10000 words then there are 10^{24} possible 6-grams to be estimated, which cannot be covered by any corpora as a consequence of Zipf's law [36]. Second, it is not able to infer the estimation of an unknown word string based on similar sequences. For example, if a good language model has seen the sentence "The cat

runs", then it should be able to tell that the unseen sentence "A dog runs" is valid, given the semantic similarity between the articles "a" and "the", and the pets "cat" and "dog". Consequently neural network language models [5] became a solution for the problems of n-gram models. The success of neural network language models is based on two key ideas. First, the models learn the *distributed representation* of words, which express the word meaning with low dimensional vectors which can represent the similarity structure between words. Second, the statistical language model is viewed as a machine learning classification problem, in which the model tries to *discriminate* the predicted word from other words in the vocabulary. The models, initially proposed by Bengio et al. [5] and then enhanced by Hai Son et al. [24], Mikolov et al. [45], Schwenk [59], Vaswani et al. [70] have been shown to outperform count-based n-gram models.

Neural networks (or Artificial neural networks) are a class of machine learning models that is inspired by biological neural networks. They are well-known to be flexible and powerful in terms of learning representations in various problems, including Natural Language Processing. Among the commonly used architectures of neural networks, there are two different structures applied successfully in language modeling: feed-forward networks and recurrent networks, making different architectural decisions. Recurrent networks [45, 46] treat language modeling as a sequence prediction problem by taking one word token at a time together with a hidden "memory" mechanism to produce a predicted word and update the memory for next time step. Importantly, they are strong enough to represent any n-gram at arbitrary size. Contrarily, feed-forward models take as input the last $n-1$ words in the n-gram, where n has to be a fixed window size, and use them jointly to predict the upcoming word.

The little summary above, as can be seen in Figure 1.1 has shown that the 20-year development of language modeling has experienced an evolution in terms of representation learning. On the one hand, the feed-forward neural language models [4] improve the basic n-gram models by introducing word-based distributed representation. On the other hand, the level of representation was significantly improved with the recurrent neural network architecture, which allows to model longer sequences without limiting the context to n words. However, it is believable that there are still rooms for improvement in language modeling, and language representation in particular.

Despite the fact that the state-of-the-art language models consist of a shared vector space between words, they still treat the context as a sequence of discrete symbols. The feed-forward network simply concatenate the word vectors and encodes the context into the hidden layer, while the recurrent models take one word as a time step, whose weakness is that a particular time step does not have any information about future steps. In fact, it is useful to take into account the collocations and multi-word expressions in the context. For example, the English word "get" can acquire different meanings when paired with different prepositions, which the word embeddings hardly can represent. Such word combinations are often identified by statistical methods [58], which requires an intensive scanning through the data to find frequent combinations. In the context of the neural networks, it is intuitive to use a special neural network architecture that can search the context for patterns and combinations.

In the literature of machine learning and computer vision, convolutional neural networks (CNNs) are the family of neural network models that feature a type of layer known as the convolutional layer. The main idea of this layer is to look at each position of the input (at any dimension) with a fixed size window and find local features - distinctive attribute that are useful for the learning task. The network has been applied successfully in image processing and speech processing [22] but they have received less attention in Natural Language Processing. Mainly, they have been somewhat explored in *static classification tasks* where the model is provided with a full linguistic unit as input (e.g. a sentence) and classes are treated as independent of each other Examples of this are sentence or document classification for tasks such as Sentiment Analysis or Topic Categorization [32, 34], sentence matching [30], and relation extraction [52]. However, their application to *sequential prediction tasks*, where the input is construed to be part of a sequence (for example, language modeling or POS tagging), has been rather limited (with exceptions, such as Collobert et al. [14]).

The language modeling in the work of Collobert et al. [14] utilised the convolutional layers[1] to extract features for multiple Natural Language Processing tasks including language modeling. However, due to the computational limit, the model was altered to focus

[1]In the original work, the author uses the term Time-delayed networks (TDNN) to describe his model. In fact, TDNN and CNN share the same properties and were designed roughly at the same time. These two terms can be used interchangeably in this manuscript.

on learning vectorised word representation, leaving the language modeling quality a question mark. In this thesis, we investigate the **feed-forward** neural language models with the temporal convolutional layer inspired by the work of [14]. Starting by analysing the existing neural language models including feed-forward and recurrent ones, we investigate the convolutional models with some reasonable changes in terms of architecture to verify if the convolutional neural network deliver any improvement over the feed-forward counterpart. For that purpose, we initially train a carefully tuned feed-forward neural language model , which yields competitive performance in various corpora. Subsequently, we enhance the baseline with an additional convolutional layer that convolves over the word embeddings, and we can effortlessly limit the customization space for network architecture during training. The experimental results indicate that using convolutional layers can improve 11-26% perplexity compared to the solid baseline, and the CNN model can perform comparably with the similarly-sized recurrent models and has lower performance with respect to larger state-of-the-art models. Our analysis also shows that the convolutional layer independently learned to focus on different linguistic patterns, and the model can take into account context words at very far from the target.

Chapter 2

Background

2.1 Statistical Language Modeling

2.1.1 Problem overview

As we introduced before, the statistical approach in Language Modeling aims at measuring the fluency of all possible word strings, which are considered as a stochastic process, with probabilities. The sequence length can be arbitrary, while the words are taken from a limited vocabulary. A trained language model should be able to show that the likelihood of "the end of our world" is much higher than "tea end of our word", because the latter string is much less likely to be found in available English text. Concretely, let us denote W_1^L to be a word sequence with length L which is composed by the constituent words $w_1, w_2, \ldots, w_L$, then a statistical language model aims at predicting the probability distribution of all possible sequences W_1^L in a particular language:

$$P(W_1^L) = P(w_1 w_2 \ldots w_L) \tag{2.1}$$

Since the direct estimation for that probability distribution is intractable, the probability of a sentence $P(w_1^L)$ is factorised using the chain rule:

$$P(W_1^L) = P(w_1|\text{<s>}) \prod_{i=2}^{L} P(w_i|\text{<s>}W_2^{L-1}) = \prod_{i=l}^{L} P(w_l|H_l) \tag{2.2}$$

Word to be predict	Context	Probability (example)
The	<s>	0.1
relationship	<s>The	0.002
between	<s>The relationship	0.05
Obama	<s>... relationship between	0.00001
and	<s>... between Obama	0.2
Netanyahu	<s>... Obama and	0.00001
is	<s>... and Netanyahu	0.05
not	<s>... Netanyahu is	0.02
exactly	<s>... is not	0.001
friendly	<s>... not exactly	0.0007
.	<s>... exactly friendly	0.2
Perplexity		210

Table 2.1: An example of perplexity computation for the sentence "The relationship between Obama and Netanyahu is not exactly friendly".

in which, <s> is used to denote beginning token of the sentence/string. The history H_i represents the string before the current word w_i. Instead of directly modeling the original distribution, the target probability is factorised into constituent *conditional* probabilities, which are more practical to estimate.

2.1.2 Evaluation Metrics

The quality of statistical language models can be evaluated by the capability to predict a new corpus, which is defined by using *Perplexity*. Let us assume that we have a trained language model M and a corpus D containing L words, which the language model has not observed during the training process. The quality of model M is evaluated by using it to predict the distribution of the corpus D. The resulted perplexity (PPL) is then obtained by estimating the probability of all words given their context in the corpus D, we use P_M to denote the probability distribution produced by model M. An example of PPL computation is illustrated on Table 2.1.

$$PPL(D) = \exp(\frac{\sum_{l=1}^{L} - \ln P_M(w_i|H_i)}{L}) \tag{2.3}$$

The idea of perplexity is that, minimising perplexity is corresponding to maximising

the ability to predict every word in the data. A low perplexity value corresponds to the fact that the language model is able to fits better the data, since the distribution of the model is closer to the unknown distribution of the test data. One property of perplexity is that, it is correlated to the average number of guessing needed to predict all words in the data. Therefore, a random distribution or the "worst" model (usually the model with parameters initialised) often gives the perplexity very close to the vocabulary size V. This property is often used to check the implementation of the model and perplexity.

In practice, the quality of language models can also be evaluated through their impacts on other applications such as Automatic Speech Recognition (ASR) or Statistical Machine Translation (SMT), by reducing the errors in the output of such systems. For example, two strings "Tea end of our word" and "The end of our world" may have similar speech signals and the recognizer has to rely on a good language model to distinguish them.

The main advantage of perplexity is that it is fast to perform and independent to other complex systems. Therefore, in this thesis, PPL would be the only measurement used in the experiments. However, it is important to note that an improvement in terms of perplexity does not always result in the application improvement. For example, the improvement is required to be at least 10% to be noteworthy for an ASR system [56]. Also, there are some language models that aim at ranking the words in the vocabulary by using unnormalised distributions (the sum of the probabilities does not equal to 1). In that case, PPL is also unusable.

2.1.3 N-gram models

Going back to equation 2.2, the target of statistical language models is to estimate the conditional probabilities $P(w_i|H_i)$ with H_i being the history context of w_i (i is an arbitrary index in the corpus). A common approach is to limit the length of the history H_i by the Markov assumption: the meaning of the word w_i only depends on $n-1$ words before, thus limiting the history for all words in the data into n-grams:

$$P(w_i|H_i) \approx P(w_i|W_{i-n+1}^{i-1}) \tag{2.4}$$

With W_j^k denotes the word sequence from index j to index k. It is natural to use $n-1$ to

denote the order of language models because it is the order of the Markov model reflected in Equation 2.4, however in the literature, authors always use n to denote the order of n-grams. Therefore, in order to be clear, an n-gram contains one word to be predicted, and $n-1$ words in the context.

Count-based models estimate the probability of each n-gram based on simple counting with maximum likelihood estimation. Let $c(W_j^i)$ denote the number of times that this word sequence occurs, the estimated distribution can be derived as follows:

$$P(w_i|W_{i-n+1}^{i-1}) = \frac{c(W_{i-n+1}^{i})}{c(W_{i-n+1}^{i-1})} \tag{2.5}$$

The method is unreliable since it struggles at estimating the distribution for rare and unseen n-grams (the nominator is 0), many of which actually make sense in natural language. Many techniques have been proposed to overcome this weakness, from re-distributing the probabilities of the frequent n-grams to the less frequent ones (smoothing techniques) or deriving the distributions of the rare n-grams from the lower order n-grams (interpolation and back-off techniques) [11, 18, 27, 37, 51, 72]. However, even with complicated smoothing techniques, the main weaknesses of n-gram language models are still exposed as follows:

First, each word in the vocabulary is treated as a totally discrete random variable without any linguistic feature associated. The model relies on statistical occurrences and ignores morphological, syntactic and semantic relationship, by which the lexicon is formulated. There are several attempts to incorporate the word similarity in n-gram based language models. Notably, the class-based language models [9, 53] introduced word clustering and assumed that the distribution of unseen or rate words can be achieved by using the richer statistics from the corresponding class, which is less sparse than the word itself. Also, structured language models [10, 19] try to filter out irrelevant context words and focus on important counterparts by using parse trees, which compensates for the lack of syntactic information in n-gram models. Despite such efforts, the language modeling results were still unreliable compared to the Knesey-Ney smoothing technique. However, those works in the literature also suggests that the syntactic and semantic properties of words need to be automatically learned from the data. Second, n-gram language models struggle to model

a long range dependency between the predicted word and the context. Due to the fact that each word in the vocabulary is a separated random variable, the number of parameters to be estimated (statistics of n-grams) grow exponentially with the size of context. The *the curse of dimensionality* refers to the fact that one needs more amount of training data in order to reliably estimate the model when the number of learned parameters increases. For example, if the vocabulary size is 10000, the total number of n-grams for $n = 6$ is 10^16 theoretically, which is also the number of parameters to be estimated accordingly. Also, Zipf's law [36] indicates that only a small subset of the vocabulary accounts for most occurrences in the training data, thus it is almost impossible to have a training data that covers all possible n-grams.

The neural network language model, as a consequence, is investigated in order to tackle both problems that traditional count-based models cannot solve.

2.2 Neural network language models

In this section, we describe the neural language models (also known as continuous space language models), which are designed to fight the curse of dimensionality in the conventional approaches, by utilising two properties as follows.

- Words are no longer discrete variables without any relationship to each other. They are represented as real-valued vectors in a continuous space, where similar words are neighbors in the space, such approach is referred as *Distributed word representation* or *Word embedding* [4, 45]. With such representation, each n-gram is a combination of the word vectors and n-grams with similar words share the same word vectors. In contrary to the count-based approach, if we increase the n-gram order then thanks to the sharing property, we just need to adjust the size of the context vector, which can grow linearly with the n-gram order (in the case of feed-forward neural networks) or remains the same (in the case of recurrent neural networks).

- When the n-gram context H_i is the combination of the word vectors, the conditional probability distribution $P(w_i|H_i)$ of each n-gram is then expressed as a parameterised smooth function of the context vector. In machine learning, neural networks

are a class of models that can be used to approximate functions. As a result, Bengio et al. [4] proposed to incorporate the continuous word space with a neural network architecture in order to learn the smooth function. The training methods of neural network allow us to **jointly learn** the word space and the parameters of that probability function.

In the following, we provide a detailed description of the two most successful architectures used for neural language models: the feed-forward models and the recurrent models. Note that, in the following sections, the term "Neural language models" or "Neural network language models" are used to refer to both architectures. They only differ in the architectural choice of design, but still possess the two described properties.

2.2.1 Feed-forward neural language models

The standard architecture of a neural language model [4] is illustrated in Figure 2.1. The model takes the context words H_i as inputs and outputs the conditional probability distribution over *all* words in the vocabulary which expresses $P(w_i|H_i)$. It consists of three basic components: the input layer, the hidden layer and the output layer. The notations are covered in Table 2.2. In a standard feed-forward neural network, each layer is a real-valued **vector**, while the (learnable) weights (or parameters) are real-valued **matrices** connecting the layers together. Also, in the standard architecture we also consider networks with only one hidden layer for ease of understanding. In practice, it is possible (and beneficial) to extend the network with multiple hidden layers.

Input layer The input layer constructs the embeddings of the words in the context, by using a shared word space and mapping each word in the context to a real-valued vector. Concretely, each word w_i is represented as an 1-of-V coding, which is a long vector with the size of vocabulary V with all zeros except for the element corresponding to the word's index in the dictionary. Using this form of sparse coding, the word space (also called projection matrix R) is a matrix that contains V rows and each word embedding v_i corresponds to one row of the matrix. The number of columns of the matrix is the size of the embedding, which is a tunable hyper-parameter. Notably, the values of the word vectors do not depend

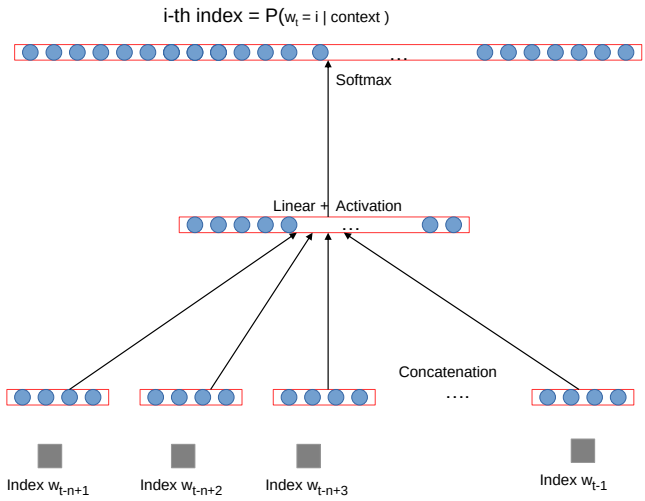

Figure 2.1: Neural language model architecture by Bengio et al [4].

on the position of the word in the context. The context vector i is the concatenation of the word vectors.

$$i = \{R^T v_1; R^T v_2; ....; R^T v_{n-1}\} \tag{2.6}$$

Hidden layer In the hidden layer, the input (context vector) is transformed nonlinearly, where each layer activation values are defined by

$$h = f(W^h i + b^h) \tag{2.7}$$

In equation 2.7, the hidden layer h has the corresponding weights W^h and b^h. The input of the hidden layer is the context vector produced from the input (projection) layer. The size of the hidden layers are tunable hyper parameters. f denotes a nonlinear activation function. Popular choices for the activation function are Tangent Hyperbolic, Sigmoid or

notation	meaning
V	Vocabulary size
m	word vector dimension
n	order of language model
w	a word; its index in the vocabulary
v_w	one-hot vector for word w. A vector with size V with all null except the w^{th} index (=1)
$i \in \mathbb{R}^{(n-1)M}$	input vector of the network.
o	output layer of the network which expresses the unnormalised probability distribution
h	hidden layer of the network
p	the output vector of the network which is normalised for the conditional probability distribution
i_j	the j^{th}-index of vector i, which is also used for o, h or p
f	nonlinear function
H	hidden layer size
$W^h \in \mathbb{R}^{H \times (n-1)M}$	the weight matrix connecting the input layer with the hidden layer
$b^h \in \mathbb{R}^H$	the bias vector for the hidden layer
$W^o \in \mathbb{R}^{H \times V}$	the weight matrix connecting the hidden layer with the output layer
$b^o \in \mathbb{R}^V$	the bias vector for the output layer
$\mathcal{L}$	The objective (loss) function for learning the network
dX	The Jacobian matrix - the matrix of all of the first order partial derivatives of the loss function with respect to the vector/matrix X.

Table 2.2: Notations for neural network layers.

ReLU, expressed in equation 2.8.

$$f(x) = \begin{cases} \frac{\exp(x)-\exp(-x)}{\exp(x)+\exp(-x)} \text{ if } f = \text{Tanh} \\ \frac{1}{1+\exp(-x)} \text{ if } f = \text{Sigmoid} \\ \max(0, x) \text{ if } f = \text{ReLU} \end{cases} \tag{2.8}$$

Output layer The final layer of the network produces the probability distribution for all words in the vocabulary, thus having totally V nodes. Each neuron in the layer is associated to the probability of one word, as shown in Figure 2.1. First, a linear transformation is used to obtain the unnormalised distribution:

$$o = W^o h + b^o \tag{2.9}$$

Notably, the values of o are unnormalised because each element in o is associated to the score of each output word given the context vector. W^o and b^o are the corresponding weights and biases of the layer. Importantly, W^o has the same form as the projection matrix at the input layer, since it also learns embedding for each word in the output vocabulary. Subsequently, the true probability distribution is estimated thanks to the *softmax* function:

$$p(w_i|h) = \frac{\exp(o_i)}{\sum_j \exp(o_j)} \tag{2.10}$$

In equation 2.10, the probability of each word w_i given the encoded context h is estimated by normalizing all values in o. Overall, the main tunable hyper-parameters of the network are the order of n-grams (the number of words in the context, which can range anywhere beyond 4, which is the typical range of a count-based model), the sizes of hidden layers, and the word embedding size. The set of free parameters Θ that are iteratively updated by learning from the data includes the projection matrix R, the weight matrices W^h at the hidden layers, W^o at the output layer and the biases b^h, b^o.

2.2.2 Training method

From the machine learning perspective, the neural network language model has transformed the statistical language modeling problem from a generative learning process to a discriminative classification problem. The free parameters of the network are trained by minimising the objective function, which is the log-likelihood $\mathcal{L}$ of the parameters Θ given the training samples. The parameters are updated after each iteration based on some optimization techniques, among which stochastic gradient descent (SGD) is most commonly used in neural language models [24, 45, 74]. SGD and other variants such as Adadelta [75] or RMSProp [69] require the computation of the first order derivatives of the loss function with respected to the parameters, which can be performed efficiently with the back-propagation algorithm [41].

2.2.3 Training/Optimisation Process

In this section, we describe the back-propagation flow in the standard feed forward neural language model - the core of the optimisation process. Back-propagation [57] involves using a dynamic programming strategy to compute the derivatives of the loss function with respect to the parameters layer by layer, based on the chain rules. In the standard network, the error derivatives are back-propagated from the output layer to the input (projection) layer.

Objective Function The smoothing function that we approximate with the neural network has parameters that can be iteratively tuned in order to **maximise the log-likelihood of the training data** [4]. The objective function is therefore chosen as the Negative Log-Likelihood function, since SGD requires the training objective to be minimised. Assuming we have N samples in the training data, each of which is an n-gram, we can compute the loss function over the training data as follows:

$$\mathcal{L} = -\sum_{i}^{N} \log P(w_i | w_{i-n+1}^{i-1}) \tag{2.11}$$

The loss function is also in line with the Perplexity in Equation 2.3. For ease of understanding, we denote the derivative of the loss function $\mathcal{L}$ at *each* sample or mini-batch of samples with respect to a variable $x \in \Theta$ by dx.

For each sample w_i and its context H_i, we have:

$$\begin{aligned} -\log P(w_i|w_{i-n+1}^{i-1}) &= -\log P(w_i|H_i) \\ &= -log(\frac{\exp(o_w)}{\sum_i exp(o_i)}) \\ &= log(\sum_i \exp(o_i)) - o_w \end{aligned} \tag{2.12}$$

Subsequently, we compute the error derivatives dx given the parameters in each layer using back-propagation:

Output layer The derivatives at the output layer:

$$do_i = \begin{cases} 1 - p_i \text{ if } i == w \\ -p_i \text{ otherwise} \end{cases} \tag{2.13}$$

Notably, o_i denotes the i^{th} element of the vector o, which is the unnormalised conditional distribution of the vocabulary given the encoded context h.

Hidden layers As a result, we can compute the derivatives with respect to the parameters and the previous hidden layer h, based on the original inference from Equation 2.9.

$$\begin{aligned} dW^o &= doh^T \\ db^o &= do \\ dh &= W^{oT} do \end{aligned} \tag{2.14}$$

Input Layer The inference equation for the hidden layer from the input layer:

$$h = f(W^h i + b^h) \tag{2.15}$$

which implies that:

$$\begin{aligned} d[W^h i + b^h] &= f'(h) * dh \\ db^h &= d[W^h i + b^h] \\ dW^h &= d[W^h i + b^h] i^T \\ di &= {W^h}^T d[W^h i + b^h] \end{aligned} \tag{2.16}$$

In order to have the derivatives for the activation function f, we have:

$$f'(x) = \begin{cases} 1 - \text{Tanh}(x)^2 \text{ if } f = \text{Tanh} \\ \text{Sigmoid}(x) - \text{Sigmoid}(x)^2 \text{ if } f = \text{Sigmoid} \\ 1 \text{ when } x > 0 \text{ and 0 otherwise if } f = \text{ReLU} \end{cases} \tag{2.17}$$

Parameter Update After obtaining the derivatives of the loss function with respect to all parameters in the network, we can update the parameters following stochastic gradient descent. The method is based on the phenomenon that the gradient of a function always points towards the direction of maximal increase at any point. The update rule is as follows with the learning rate parameter $\alpha > 0$ and an arbitrary parameter x:

$$x = x - \alpha dx \tag{2.18}$$

The learning rate is also considered as a function of the number of samples trained in the data. From experiments, the learning rate is updated after the model observes a number of training examples with two typical ways. The first way is to exponentially decrease the learning rate after some training samples with a *learning rate decay*, normally an epoch (training all samples in the training data). The second way is to reduce the learning rate based on a validation data. After each epoch, if the perplexity on the validation data is decreased, the learning rate is kept the same, otherwise it is multiplied by the learning rate decay.

Methods to prevent overfitting Overfitting is a phenomenon that the model has poor predictive performance, even if the model is well trained on the training data. The possibility of overfitting exists because the criterion used for training the model is different than the criterion used to measure the efficacy of the model. It is very likely that the training data yields a different distribution than the test data, therefore fitting the model on the training data does not guarantee a good prediction performance.

For neural network language models, the main methods used to prevent this phenomenon to happen is to apply **regularisation methods** and **early-stopping** strategy.

- Regularisation: The most efficient method is to apply **Dropout technique**, which refers to dropping out units (in most case, hidden units) in neural networks [28, 64]. Concretely, we temporarily set the unit values to 0 based on a random distribution (Bernoulli distribution, for example) during the training phase. In the testing phase, the unit is always present. Dropout is usually applied in the hidden layers of feed-forward neural networks. The choice of which unit to drop in the layer is random which is normally associated with a fixed probability p independent for each unit. Depending on the network size and the amount of training data, the value of p is chosen empirically.

- Early-stopping: In order to prevent overfitting, we use a validation set during training. The validation data is a separate set with the size similar to the test data. After each epoch (normally a whole scan over the training data), we measure the perplexity of the model on the validation data. If the perplexity does not decrease, the learning rate is reduced in the next epoch. The training process is halted when the learning rate reaches a threshold.

2.2.4 Recurrent neural language models

Compared to statistical n-gram models, feed-forward neural language models created a considerable leap in representation by combining distributed representation of words with a robust classifier to generalise from observed sequences. As can be seen from various works [24, 59], the feed-forward language model significantly outperformed the traditional

count-based models. However, feed-forward models require a fixed input size, thus still rely on the Markov Assumption which limits the context to a particular number of words, even when we manually set the context size can be large. In order to model long sentences, or even paragraphs with long-term dependencies, it is beneficial to investigate in models that can be flexible in terms of input size. For example, if the distance between the open bracket and the closed counterpart is further than the n-gram input size, the feed forward model may forget to close the brackets after seeing the initial one. Ideally, learning processes in human are associated with a memory that keeps the current information (such as topics), a similar structure should be simulated and integrated in the network.

Recurrent neural networks RNN [17] are a class of neural networks that can efficiently model sequences by using a dynamic memory structure. While the feed-forward network can only receive one input and compute the corresponding output without any relation with other inputs, the recurrent counterpart takes the input as a *series of time step* $x_1, x_2, \ldots, x_n$ and processes them one by one, taking into account the information stored in the previous steps. Concretely, for each input x_i, the network updates the hidden memory h^i based on the previous one h^{i-1}.

$$h^i = \mathcal{F}(x_i, h^{i-1}) \tag{2.19}$$

We will cover some popular RNN variations in the upcoming sections. In general, the strength of these models lies in the ability to dynamically model sequences with arbitrary length, which the feed-forward neural networks cannot achieve. The advantage comes with the cost that the recurrent models are generally hard to train, due to the properties of back-propagation. A change in an arbitrary position of the sequence can lead to a change in the objective function, therefore training methods for RNNs typically have to trace back the previous time steps. In other words, the RNNs are equivalent to feed-forward neural networks with many hidden layers that share parameters across each other. Importantly, the model capacity of RNNs do not depend on the length of the sequences, but in the recurrence mechanism - the way the hidden layers are updated.

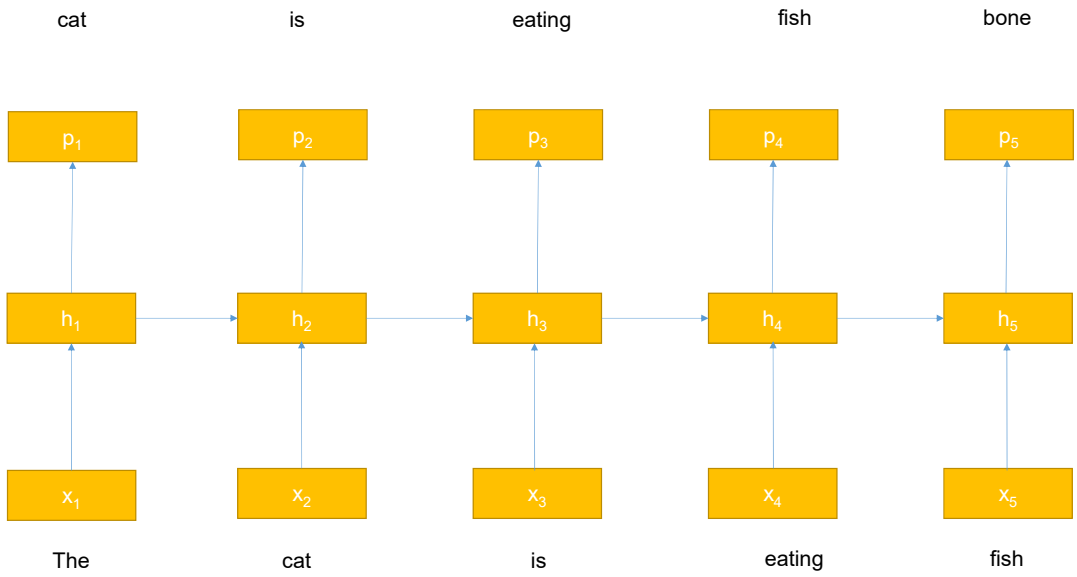

Figure 2.2: A simple RNNLM [45] predicting the sequence "The cat is eating fish bone".

Recurrent language models Language modeling can be viewed as a sequence modeling problem, in which each time step corresponds to one word. We reuse the notations from Table 2.2 with another notation for recurrent layers added by Table 2.3.

The first recurrent language model (RNNLM) [45] employed the "Vanilla" model of Elman et al [17] as can be seen in Figure 2.2, in which the hidden steps are updated as follows:

$$h^t = f(W^i i^t + W^h h^{t-1} + b^h) \tag{2.20}$$

notation	meaning
V	Vocabulary size
M	word vector dimension
n	order of language model
w	a word; its index in the vocabulary
H	hidden layer size
v_w	one-hot vector for word w. A vector with size V with all null except the w^{th} index (=1)
X^T	the matrix / vector X transposed
$X * Y$	The element-wise matrix multiplication between matrices X and Y
$i^t \in \mathbb{R}^M$	input vector of the network at time t.
$h^t \in \mathbb{R}^V$	hidden layer of the network at time t
$o^t \in \mathbb{R}^V$	output layer of the network which expresses the unnormalised probability distribution at time t
$p^t \in \mathbb{R}^V$	the output vector of the network which is normalised for the conditional probability distribution
f	nonlinear function
$W^i \in \mathbb{R}^{H \times M}$	the weight matrix connecting the input layer with the hidden layer
$W^h \in \mathbb{R}^{H \times H}$	the weight matrix connecting the hidden layer of the previous step with the current hidden layer
$b^h \in \mathbb{R}^H$	the bias vector for the hidden layer
$W^o \in \mathbb{R}^{H \times V}$	the weight matrix connecting the hidden layer with the output layer
$b^o \in \mathbb{R}^V$	the bias vector for the output layer
$\mathcal{L}$	The objective (loss) function for learning the network
dX / dx	The Jacobian matrix - the matrix/vector of all of the first order partial derivatives of the loss function $\mathcal{L}$ with respect to the matrix X or vector x.

Table 2.3: Notations for recurrent neural network layers.

The activation function f can be either Tangent Hyperbolic, Sigmoid or ReLU as mentioned before. The starting state h_0 is set to 0 to denote the initial state of the memory. In each time step, the RNNLM can optionally produce the probability distribution for a predicted word, given the sequence that the network has scanned previously. The probability distribution over the vocabulary is derived similarly to the feed-forward networks:

$$\begin{aligned} o^t &= W^o h^t + b^o \\ p^t &= \text{softmax}(o^t) \end{aligned} \tag{2.21}$$

with the Softmax function explained in Equation 2.10. To be clear, o^t and p^t denote the unnormalised and normalised distribution generated at time step t. For the language modeling scenario, the input and output samples of the network in each training iteration are two sequences i and y in which the output sequence is the shift-by-1 version of the input sequence. The parameter set of the network including W^i, W^h, b^h, W^o and b^o are shared across time steps.

2.2.4.1 Training Recurrent Networks

Similarly to the feed-forward models, recurrent models can be efficiently trained with stochastic gradient descent (SGD). However, since the networks contain shared parameters at arbitrary numbers of time steps, the gradients are computed differently using back-propagation through time (BPTT). It can be observed that, a change in the parameters in an arbitrary time step t can lead to the change of the objective function in **all** subsequent steps.

The details of this algorithm are divided into two parts: the local gradients computed at each time step, and the global gradients accumulated by un-folding the networks.

At each time step The input and output sequences are denoted as i and y. At each time step t, the network receives one input i^t and predicts one output word y^t. Similar to the back-propagation process for feed-forward neural language models, we need to derive the derivatives of the per-time-step loss function $\mathcal{L}^t$ with respect to the RNN parameters $\{W^i, W^h, b^h, W^o$ and $b^o\}$ and the inputs $\{i^t, h^{t-1}\}$. Notably, the network at each time step receives two inputs: the input word vector i^t and the previous hidden layer h^{t-1}.

The Loss (Objective) function is identical to the feed-forward network:

$$\mathcal{L}^t = -\ln p^t \tag{2.22}$$

with p^t being the conditional probability of the output word at time t given the history from time 1 to $t-1$, derived with Equation 2.21. The corresponding gradients are as follows:

$$do_j^t = \begin{cases} 1 - p_j^t \text{ if } j == y^t \\ -p_j^t \text{ otherwise} \end{cases} \tag{2.23}$$

The derivatives at the hidden layer can be derived:

$$\begin{aligned} dh^t &= W^{oT} do^t \\ dW^o &= do^t h^{tT} \\ db^o &= do^t \end{aligned} \tag{2.24}$$

After that, the errors are propagated to the input and the previous hidden layer. First of all, we use the simplified version of the RNN formulation in Equation 2.20, by denoting $S = [W^i W^h]$ and $z^t = [i^t; h^{t-1}]$. The RNN formulation is neatly simplified as:

$$h^t = f(Sz^t + b^h) \tag{2.25}$$

We can derive the derivatives for the necessary weights (S) and layers (z^t) as follows:

$$\begin{aligned} dz^t &= S^T(f'(Sz^t + b^h) * dh^t) \\ dS &= (f'(Sz^t + b^h) * dh^t) z^{tT} \\ db^h &= d(Sz^t + b^h) \end{aligned} \tag{2.26}$$

From there, we can compute the gradients of the original variables (W^i, W^h) and layers (i^t, h^{t-1}).

$$
\begin{aligned}
di^t &= W^{iT}(f'(Sz^t + b^h) * dh^t) \\
dh^{t-1} &= W^{hT}(f'(Sz^t + b^h) * dh^t) \\
dW^i &= (f'(Sz^t + b^h) * dh^t)i^{tT} \\
dW^h &= (f'(Sz^t + b^h) * dh^t)h^{t-1T} \\
db^i &= (f'(Sz^t + b^h) * dh^t)
\end{aligned}
\tag{2.27}
$$

The derivatives of the activation function has already been provided for the feed-forward neural network language model, in Equation 2.17.

Back-propagation through time The idea of BPTT is that the network is assumed to have different weights at each time-step. During training, the gradients are back-propagated to the previous layers through the recurrent connections. We compute the gradients for the weights at each time step with the formulation that we have just derived for a single time-step. Eventually, we force the shared weights at each time step to have the same value by **accumulating** the gradients together.

Algorithm 1: BPTT algorithm for "vanilla" RNNs

Inputs: input sequence i and output sequence y with length T.
Initialize the gradients $do^t, dW^o, dW^i, dW^h, di^t, dh^{t-1}$ as zero vectors and matrices.
for $t = T \rightarrow 1$ **do**
    `// At the output layer`
    $do^t \leftarrow v_{y_t} - p_t$
    $dW^o \leftarrow dW^o + do^t \cdot h^{tT}$
    $dh^t \leftarrow dh^t + W^{oT} do^t$
    `// RNN Backprop`
    $dW^i \leftarrow dW^i + (f'(Sz^t + b^h) * dh^t)i^{tT}$
    $dW^h \leftarrow dW^h + (f'(Sz^t + b^h) * dh^t)h^{t-1T}$
    `// To the input layer and previous hidden layer`
    $di^t \leftarrow di^t + W^{iT}(f'(Sz^t + b^h) * dh^t)$
    $dh^{t-1} \leftarrow dh^{t-1} + W^{hT}(f'(Sz^t + b^h) * dh^t)$
end

The whole BPTT process is illustrated in algorithm 1. The main difference between

BPTT and BP for feed-forward network is that the gradients at the hidden state can be propagated by two paths: a "vertical" line from the hidden layer to the input layer, and the "horizontal" line to go back to the previous step. The network becomes harder to train when the sequence length T is too large, and the Vanilla RNN is not efficient to learn to remember long sequences.

Problems with BPTT Although truncated back-propagation through time provides a practical training method for RNNs, the nonlinear iterative nature of the simple RNN architecture still makes capturing long-term dependencies difficult. The two common problems encountered while training RNNs are the *exploding* and the *vanishing* gradients [3, 54]. On the one hand, the gradients can be exponentially large as in the back-propagation through time process which is detrimental for learning. One the other hand, we can also experience the phenomenon that gradients go quickly towards zero after being propagated through time steps. Consequently, the model is not able to track the signal and complete loses the memory trace in the past. For example, the original RNNLM normally has to truncate the BPTT at about $5 - 10$ steps, which has the same modeling capacity and performance with 10-gram feed-forward models [25, 46]. The gradient exploding problem can be tackled adequately using gradient clipping. Pascanu et al [54] suggested to clip the norm of the gradients (for all parameters): given a gradient vector dx that is computed with BPTT, if the norm $||dx||$ is greater than a threshold value δ, then dx would be softly scaled:

$$dx \leftarrow dx \frac{||dx||}{\delta} \tag{2.28}$$

Dealing with gradient vanishing A number of solutions were proposed to solve the gradient vanishing problem. We will make a brief review of the most prominent approaches. First, Mikolov et al [48] proposed to integrate another memory layer which is formed by the bag-of-word addition of the input words over time, decays slower than the main hidden memory and is initialised as an identity matrix. The same initialisation trick is applied together with using Rectified Linear Units (ReLU) as the activation function in [40]. While both works mentioned are fairly simple, the RNNs can also be trained efficiently with second order derivatives using Hessian-Free optimization [44]. Even though the method was

proved to allow the network to acquire a reliable memory which can remain stable after hundreds of time steps, it is not easy to implement efficiently compared to traditional back-propagation. The most successful method that is applied to sequential modeling in general and language modeling in particular is the Long-Short Term Memory LSTM networks [29] that use an explicit memory cell combined with a gating mechanism to intensively deal with the gradient vanishing problem.

LSTM Structure The intuition of an LSTM starts from the integration of a linear memory unit, so that the gradient can flow smoothly during the back-propagation through time steps using a memory cell c^t.

$$\begin{aligned} c^t &= c^{t-1} + f(Wx^t + Uh^{t-1} + b) \\ h^t &= c^t \end{aligned} \tag{2.29}$$

This approach is referred as "Leaky integration units" [6]. In the BPTT process, the gradient can flow over exactly one path through the memory units c^t, and since $dc^t = dc^{t-1}$, the gradients are guaranteed to not vanish. The recurrent architecture should also be able to be adequately robust to train long sequences, where there are certain inputs which are irrelevant to the modeling task. Sometimes, the memory of the network should be refreshed, for example at the beginning of a new utterance in Speech Recognition [21] or a new sentence in Machine Translation [67]. Hochreiter and Schmidhuber [29] enhanced the architecture by adding flexible and trainable gates that allows the RNN to reset the memory, control the amount of input and output respectively. The adaptive gates are built from the current input x^t and the previous hidden memory h^t.

The gates of the network include: the forget gate f^t is used to directly control the memory flow c^t to cut the connection with the previous steps, the input gate i^t decides the amount of input to be incorporated, the output gate o^t controls the amount of memory flow to be produced for the task and finally the candidate memory unit $\tilde{C}$ that contributes to the current memory flow. All gates are defined similarly, with the first three gates use the Sigmoid activation to force the values to be in $\{0, 1\}$, while the candidate memory uses the Tanh activation function. The overall interactions of the LSTM cell is illustrated in Figure 2.3.

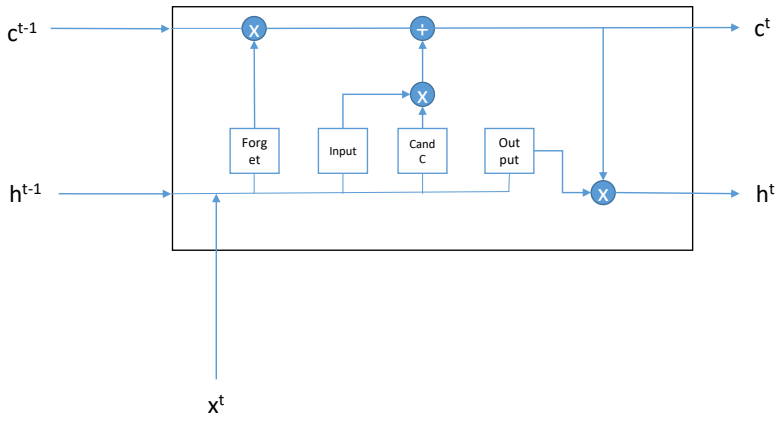

Figure 2.3: LSTM cell architecture with 4 gates [29]. The operations used in the figures are matrix element-wise multiplication (x) and matrix addition (+).

$$
\begin{aligned}
f^t &= \text{Sigmoid}(W^f x^t + U^f h^{t-1} + b^f) \\
i^t &= \text{Sigmoid}(W^i x^t + U^i h^{t-1} + b^i) \\
o^t &= \text{Sigmoid}(W^o x^t + U^o h^{t-1} + b^o) \\
\tilde{C}^t &= \text{Sigmoid}(W^c x^t + U^c h^{t-1} + b^c)
\end{aligned}
\tag{2.30}
$$

In the next step, we decide the new information to be stored in the new memory cell. The cell is updated by combine the input gate and the candidate memory unit. Also, the forget gate is employed to drop certain information from the previous memory cell. Consequently, we come up with a new memory cell as follows:

$$C^t = f^t * C^{t-1} + i^t * \tilde{C}^t \tag{2.31}$$

Finally, we update the hidden state with the new cell state and the output gate:

$$h^t = o^t * \text{Tanh}(C^t) \tag{2.32}$$

The implementation of LSTM can be efficient by computing all gates in one single matrix multiplication, then applying the activation functions on different parts of the output. In practice, one can experience different implementation variations of LSTMs and RNNs

in terms of initialisation, bias usage or different gate implementations such as the Gated Recurrent Unit [13]. The empirical research of [73] shows that there is not any substantial difference in terms of performance between different LSTM variations.

Multi-layer recurrent neural network We can extend a recurrent neural network by stacking the recurrent layers on top of each other in one single time-step. Concretely, the hidden layer at level i is the input of the hidden layer at level $i + 1$. For the recurrent connection of the network, the hidden layer at level i only depends on the previous hidden layer of the same level i.

Regularisation in RNN Similar to feed-forward networks, the recurrent networks are prone to overfitting. In order to tackle the problem, it is efficient to apply dropout before the input of each recurrent unit [55, 74].

2.2.5 Softmax approximation in neural language models

The neural language models are computationally expensive, which makes training them on large corpora impractical. The bottleneck lies in the output layer where the term $H \times V$ accounts for most of the computational cost, especially when V is large (from 60000). Looking back at the output layer, let us assume h being the last hidden layer in the feed-forward network, or the last hidden layer at time step i in the recurrent network, then we can compute the probability of the word w_i given the context:

$$\begin{aligned} o &= W^o h + b^o \\ p(w_i|H_i) &= \frac{\exp(o_i)}{\sum_j \exp(o_j)} \end{aligned} \tag{2.33}$$

In Equation 2.33, it is important to note that W^o is another word embedding space, in which each row W_i^o is the embedding corresponding to the word w_i. It is expensive to compute the denomination term of softmax, which requires computing the term $W_j^o h + b_j$ repeatedly for each word w_j. It was common to train neural network language models in the order of months on large corpora with a large vocabulary [4, 14, 45].

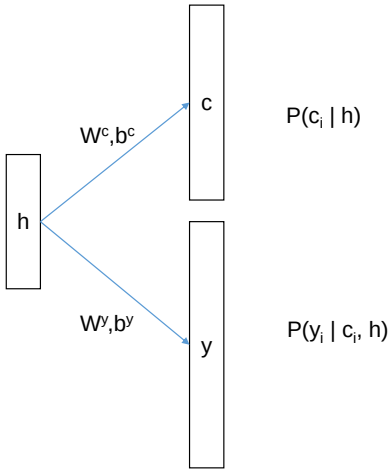

Figure 2.4: Hierarchical Softmax: Factorisation of the output layer.

In order to make training neural network language models practical, there are a lot of techniques proposed in order to avoid the computational bottleneck above. **Approximating the softmax computation** is the aim of those techniques, two most successful methods are provided in this thesis: Hierarchical Softmax (HSM) and Noise Constrastive Estimation (NCE).

Hierarchical Softmax The key idea of hierarchical softmax is to factorise the output layer based on word clustering [50]. Fundamentally, the vocabulary V is factorised into **classes** containing C classes. Every word w_i is assigned to a class c_i before training and the assignment does not change during the process. The clustering assumption can be done based on WordNet [50], using unsupervised methods on top of pretrained word embeddings [24] or simple frequency binning [46].

The softmax approximation is done as follows. First, we compute the probability of the output class c_i given the context H_i:

$$p(c_i|H_i) = \text{Softmax}(W^c h + b^c) \tag{2.34}$$

After that, we compute the probability of the word w_i belonging to class c_i given the

context H_i:

$$p(w_i|c_i, H_i) = \text{Softmax}(W^y h + b^y) \tag{2.35}$$

Note that, W^y and h^y are subset of the weights W^o and h^o in the original softmax layer. Finally, we have the probability of the word w_i given the context H_i:

$$p(w_i|H_i) = p(c_i|H_i)p(w_i|c_i, H_i) \tag{2.36}$$

Instead of computing the large softmax layer, the conditional probability of each word is factorised into two smaller softmax layers. On the CPU-based implementations, the speed-up can be easily achieved at 15 - 30 times compared to the original softmax [24, 45]. However, the method is not very friendly for GPU-based implementations.

Noise Constrastive Estimation NCE [23] was proposed for neural language models by Mnih and Teh [49]. The key idea of NCE is that: we avoid the expensive softmax function which requires the involvement of all words in the vocabulary in one calculation by training a model which discriminates the target word with a noise distribution Q.

For every word w_i given the context H_i, we generate k noise samples words n_{ik} from a noise set whose distribution has already been known. Instead of minimising the negative-loglikelihood (or perplexity) in the normal loss function $\mathcal{L}$, the NCE method involves using a logistic regression classifier in order to separate the target words w_i from the noise samples n_{ik}. As labels are necessary to perform the classification task, we designate the target words w_i as true labels ($y = 1$) and noise samples n_{ik} as false labels ($y = 0$).

Given N samples in the training data (in most case, N is the total number of words in the training data), the logistic regression loss function is as follows:

$$\mathcal{LR} = -\sum_{i}^{N}[\log P(y = 1|w_i, H_i) + k\sum_{j=1}^{k}\log P(y = 0|n_{ij}, H_i)] \tag{2.37}$$

The intuition is to maximise the probability of the target word given the context, and minimise the probability of the noises. We can represent both distributions based on the mixtures: the noise distribution Q which is known, and the distribution that we obtain

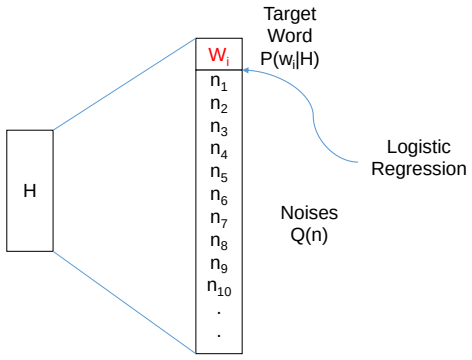

Figure 2.5: Noise Contrastive Estimation illustration.

during training which is the unnormalised distribution (o in Equation 2.33 that we call P. We can compute P individually for each word (note that we do not use softmax, so this is an unnormalised distribution).

$$P(w_i|H_i) = \frac{W_i^h h}{Z} \tag{2.38}$$

Z is a normalisation term which is chosen experimentally and totally independent to the history context. Given P and Q, we can calculate the left and right term of Equation 2.37 as follows:

$$P(y = 1|w_i, H_i) = \frac{P(w_i|H_i)}{P(w_i|H_i) + kQ(w)} \tag{2.39}$$

$$P(y = 0|w_i, H_i) = \frac{kQ(w_i)}{P(w_i|H_i) + kQ(w_i)} \tag{2.40}$$

During training, the network tries to minimise the loss function $\mathcal{LR}$. The gradients are computed in a similar manner to the negative log-likelihood function. In the testing phase, it is required to compute the normalised probability distribution, therefore normal softmax function is used again (we do not sample noise during the testing phase). NCE can theoretically guarantee that, if the number of noise samples k is increased, the NCE

derivative tends toward the gradient of the softmax function.

For using the NCE training method, we have to choose two hyper parameters: the number of noise samples k and the normalisation term Z. NCE can effectively speed-up training neural language models because the computation at the output layer is reduced dramatically from the whole vocabulary to only the target word and k noise samples. More importantly, the NCE technique is insensitive to the output layer size (vocabulary size) unlike the HSM technique which allows NCE to be effective at any large vocabulary size.

In the literature, there are several techniques used to train language models that resemble NCE, including Importance Sampling [2], Sampled Softmax [60] or Negative Sampling [47].

2.3 Convolutional Neural Networks

In this section, we provide the background information about the general Convolutional Neural Networks (CNN) which are mostly applied in the field of Computer Vision. We briefly cover the basic computation flow for the Convolutional layer - the fundamental component of the networks, whose intuitions are somewhat easier to understand for Computer Vision use case. The application of CNN for Natural Language Processing and Language Modeling in particular will be described in the next chapter.

2.3.1 The convolutional layer

Fundamentally, the convolutional layer is a biologically inspired variant of the fully connected layer (typical layers used in the feed-forward neural network language models). While the fully-connected layer expects a vector (1-dimension) as input and outputs a new vector using matrix multiplication, the convolutional layer receives inputs up to 3-dimension (such as images with channels, width and height) and, in most cases, produces an output that keeps the same dimensionality. Originally, the network was designed under the form of **Time-delayed neural networks** (TDNN) [71] in order to deal with $2D$ inputs such as speech signals. To clarify, the term TDNN and CNN are different names of a single concept that appeared in the literature at the same time, which refers to the use of *Local*

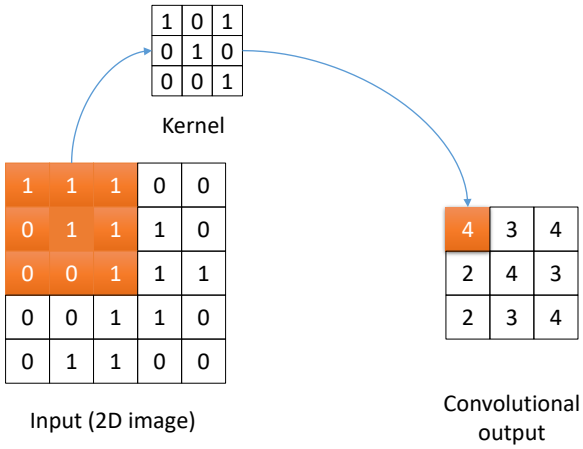

Figure 2.6: Simple illustration for convolution. The input is a 2D image, the output is obtained by sliding the kernel through the image.

receptive field in the input [20, 61].

The core idea of the layer is that the neurons in the CNN output are only connected to a small region (local receptive field) of the layer before it, instead of all neurons in a fully-connected manner. The purpose of local receptive fields is to find features within the input that are invariant to position. For example, if we rotate an image then the textures are not changed, or if we look for the adjectives in the a sentence then the position of them is not important. The basic computation is to convolve (sliding) a window function applied to the input. The operation is illustrated on Figure 2.6. Each index in the output is obtained by performing the values of the kernel element-wise with the corresponding values in the input where the kernel is sliding on, then summing them up. The full convolution is done by repeating the operation by sliding the kernel over the input.

Forward propagation For the purpose of simplicity, in the following operations, we derive the forward and backward formulas for 2D inputs and the delay between convolutional kernels is just one unit.

In order to mathematically formulate the convolutional layer, let us assume that the input is a $N \times N$ square neuron layer X which is followed by the convolutional layer. Let

the filter (K) size be $m \times m$, the convolutional output Y will be of size $(N-m+1) \times (N-m+1)$. Each unit Y_{ij} in the output layer is computed by summing up the contributions from the input layer which are weighted by the kernel:

$$Y_{ij} = \sum_{a=1}^{m} \sum_{b=1}^{m} K_{ab} \cdot X_{(i+a)(j+b)} \tag{2.41}$$

The convolutional outputs are usually fed into non-linear functions in a similar manner to the fully-connected layers. In the convolutional layer, we specify the kernels as **learnable parameters** which are automatically adjusted during training based on the task of the network. For example, in the image classification tasks, CNNs have been found to be able to extract from low level features such as edges, colors, shapes to higher-level features such as facial shapes [39].

Backward propagation Let us assume that we have the loss function $\mathcal{L}$ and we know the error values at the convolutional output, dY_{ij} (we keep the same notation as previous network types). Based on the output gradients, we want to compute the derivatives of the loss function with respect to the weights dK_{ij} (i and j are arbitrary indices).

First, using the chain rule, we sum all of the contributions of all expressions in which the variable occurs:

$$dK_{ab} = \sum_{i=1}^{m} \sum_{j=1}^{m} dY_{ij} \cdot X_{(i+a)(j+b)} \tag{2.42}$$

So that we can update the kernel parameters when we receive the back-propagation signal from the convolutional output. Furthermore, we also need to back-propagate the errors back to the convolutional input, in other words computing dX_{ij}

$$dX_{ij} = \sum_{a=1}^{m} \sum_{b=1}^{m} K_{ab} \cdot dY_{(i-a+1)(i-b+1)} \tag{2.43}$$

From Equations 2.42, 2.43, the backward operations can be efficiently implemented as convolutions similar to the forward operations. Also, the formulas also suggest that, we can expand the convolution with **padding** the inputs (so that the backward operations

make sense for input units which are close to the border) and increasing the steps between convolutional steps (**stride**).

2.3.2 Hyper parameters for convolutional neural networks

Given the explanation of convolution - which is the connectivity between the neurons in the input layer and output layer, we can decide the number of neurons in the output layer (which has not been explicitly mentioned in the previous sections). The output layer size depends on the following hyper parameters (which are typically tried experimentally, or with a grid-search strategy).

- **Kernel size** The size of the window that we use to convolve the input.
- **Convolution depth** it corresponds to the number of kernels (sliding windows that we use to scan the input each of which learns a different feature of the input. For example, if we use convolutional neural networks to extract features for sentence classification, then multiple kernels can be distributed to learn features related to nouns, verbs or adjectives.
- As mentioned above, we can alter the step which we slide the filter over the input with the parameter **stride**. The larger the stride is, the smaller output is produced by convolution.
- We can also control the output size by adding zero values to the the borders of the input. For example in Figure 2.6, the convolution operator was not able to be performed at the index $(1, 1)$ unless we pad 2 zero neurons to each dimension (and each side) of the input. In that case, we will obtain an output with the same size as the input.

Two hyper-parameters that affect the number of free parameters are kernel size and depth, while the other two control the output size of the layer.

Chapter 3

Convolutional Neural Language Models

The convolutional neural networks have been successfully applied in Computer Vision [26, 39, 68] and Speech Recognition [1, 21]. The main idea of applying CNNs into those problems is to extract features related to short-time lags [29] - the arrangement of input units in a local area. For example, the CNNs are effective at extracting edges and tiny patterns in vision, or sequences of phonemes in speech signals.

A prominent work of applying CNNs in NLP is from Collobert et al. [14] who propose a neural network model with a convolutional layer in order to extract n-gram features for multiple NLP tasks including language modeling and part-of-speech tagging, chunking, named entity recognition. The network resembles the feed-forward neural language model [4], which includes the first layer that learns the continuous word representation and subsequent hidden layers. The main difference is that Collobert et al. [14] employ a convolutional layer (referred in the original work as Time-delayed neural network). Unfortunately, due to the computational limitations, the authors could not train the network by minimising the perplexity of the training data, but instead using a ranking-based loss function and focused on training the word embeddings. As mentioned before, the main purpose of the thesis is to investigate in the comparison between the standard feed-forward neural network and the convolutional neural network language model. Our CNN is constructed by extending a feed-forward language model (FFLM) with convolutional layers. In what follows, we first explain the implementation of the base FFLM and next, we describe the CNN that we study.

3.1 Baseline FFLM

Our baseline feed-forward language model (FFLM) is almost identical to the original model proposed by Bengio et al. [5], with only slight changes to push its performance as high as we can, producing a very strong baseline. In particular, we extend it with highway layers and use Dropout as regularization. The model is illustrated in Figure 3.1 and works as follows. First, each word in the input n-gram is mapped to a low-dimensional vector (viz. embedding) though a shared lookup table. Next, these word vectors are concatenated and fed to a highway layer [65].

Highway layer Highway layer is a variation of the conventional fully-connected layer. The mechanism of highway layers can combine the non-linear affine transformation of an input layer with itself to create the output hidden layer. Assuming the input and output of the highway layer is H_I and H_O. The materials for a highway layer is a typical non-linear transformation H and a transform gate T

$$\begin{aligned} H &= f(H_I.W_h + b_h) \\ T &= (H_I.W_t + b_t) \end{aligned} \tag{3.1}$$

W_h, b_h and W_t, b_t are linear connected weights for the model, f is a nonlinear function (ReLU in our work). Notably, the highway layer requires double the weights than a typical linear connected layer. The output of the highway layer is the combination of the transformed input and a part of the input carried away to the output.

$$H_O = H * T + H_I * (1 - T) \tag{3.2}$$

Succinctly, highway layers improve the gradient flow of the network by computing as output a convex combination between its input (called the *carry*) and a traditional non-linear transformation of it (called the *transform*). As a result, if there is a neuron whose gradient cannot flow through the transform component (e.g., because the activation is zero), it can still receive the back-propagation update signal through the carry gate. We empirically observed the usage of a single highway layer to improve importantly the performance of

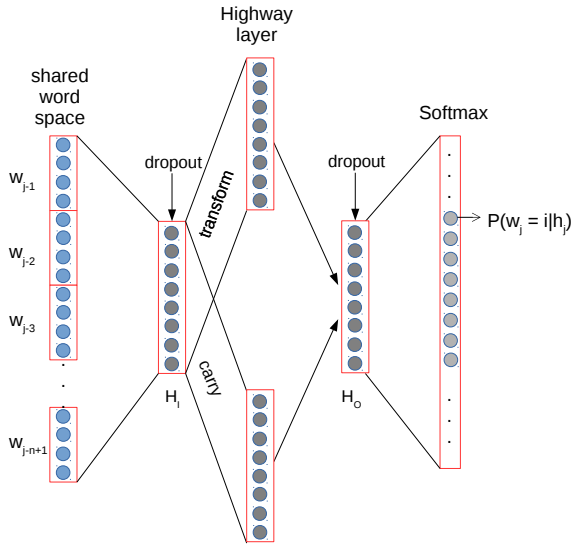

Figure 3.1: Overview of baseline FFLM.

the model. Even though a systematic evaluation for this model is beyond the scope of the current paper, our empirical results demonstrate that it is a very competitive one (see Chapter 4).

Dropout Another important technique that we applied in our model is **Dropout** [28] which is a technique used to prevent overfitting in training neural networks. The technique is to drop out neurons at hidden layers during the training process (set their values to 0) with a random distribution (Bernoulli distribution is commonly used in dropout implementation.) The main purpose of dropout is to prevent the co-adaptation of feature detectors (neurons) on the training data. When a neuron is dropped out, the neighbor neurons cannot rely on it to produce the prediction anymore, thus the network are "penalised" more severely on the training data but is more robust on unseen data.

Finally, a softmax layer (which is equivalent to the final hidden layer of the standard neural language model plus a softmax function) computes the model prediction for the upcoming word. We use ReLU for all non-linear activations and Dropout [28] is applied between each hidden layer.

3.2 CNN model and variants

3.2.1 Basic CNN network

The proposed CNN network is produced by injecting a convolutional layer right after the words in the input are projected to their embeddings (Figure 3.3). Rather than being concatenated into a long vector, the embeddings $x_i \in \mathbb{R}^k$ are concatenated transversally producing a matrix $x_{1:n} \in \mathbb{R}^{n \times k}$, where n is the size of the input and k is the embedding size. This matrix is fed to a time-delayed layer, which convolves a sliding window of w input vectors centered on each word vector using a parameter matrix $W \in \mathbb{R}^{w \times k}$. Convolution is performed by taking the dot-product between the kernel matrix W and each sub-matrix $x_{i-w/2:i+w/2}$ resulting in a scalar value for each position i in input context. This value represents how much the words encompassed by the window match the feature represented by the filter W. A ReLU activation function is applied subsequently so negative activations are discarded. This operation is repeated multiple times using various kernel matrices W, learning different features independently. Here we tie the number of learned kernels to be the same as the embedding dimensionality k and set the zero-padding so that the output of this stage will be another matrix of dimensions $n \times k$ containing the activations for each kernel at each time step. For example, for window size $w = 5$, we pad the inputs with 2 zero units at each direction. The padding value is 3 for $w = 7$ and so forth. We always move the window by 1 unit per time lag (stride $s = 1$). The main reason for such setup is to keep the network structure identical to the baseline, and also to reduce the total number of parameters to be tuned during training.

Next, we add a batch normalization stage immediately after the convolutional output, which facilitates learning by addressing the internal covariate shift problem and regularizing the learned representations [31].

Finally, this feature matrix is directly fed into a fully connected layer that can project the extracted features into a lower-dimensional representation. This is different from previous work [14, 34], where a max-over-time pooling operation was used to find the most activated feature in the time series. Our choice is motivated by the fact that the max pooling operator loses the specific position where the feature was detected, which is important for word prediction.

After this initial convolutional layer, the network proceeds identically to the FFNN by feeding the produced features into a highway layer, and then, to a softmax output.

Training the network tochastic gradient descent (SGD) can be used to train our proposed language model. The objective of the network is to jointly learn the word embeddings, the convolutional kernels and maximising the likelihood of the training data. The loss function is similar to feed-forward and recurrent neural networks, which is the negative-loglikelihood function. The derivatives of the loss function with respect to the weights at each layer are computed with the back-propagation algorithm, which is described for feed-forward and convolutional neural networks in Chapter 2.

3.2.2 Extensions

This is the basic CNN architecture. We also conducted experiments with possible expansions to the basic model as follows.

First, motivated by the successes of image recognition networks that gain performance by stacking convolutional layers [26, 63], we aim at connecting learning features at local level by using stacked convolutional layers on top of each other (Multi-layer CNN or **ML-CNN**). It is possible since after the first convolutional layer, the output has the same size and the input.

Second, we generalize the CNN by extending the shallow linear kernels with deeper multi-layer perceptrons, in what is called a MLP Convolution (**MLPConv**) structure [43]. Fundamentally, the convolutional kernels are universal function approximators that maps the embedding vectors into a value (feature). The shallow linear kernels can be upgraded into non-linear function approximators by using a fully connected neural network scanning through the input, which is illustrated in Figure 3.2. For comparison, the features extracted by a shallow linear convolution layer (the simple convolution described above) are exactly the hidden layer of the MLPConv layer. By adding another non-linear layer with the ReLU function, we can expect to have a better feature extractor (since the network is deeper).

Concretely, we implement MLPConv networks by using another convolutional layer with a 1×1 kernel on top of the convolutional layer output. This results in an architecture

that is exactly equivalent to sliding a one-hidden-layer MLP over the input. Notably, we do not include the global pooling layer in the original network structure [43] which is specifically designed for computer vision applications.

Finally, we consider combining features learned through different kernel sizes (**COM**), as depicted in Figure 3.4. For example, we can have a combination of kernels that learn filters over 3-grams with others that learn over 5-grams. This is achieved simply by applying in parallel two or more sets of kernels to the input and concatenating their respective outputs [34]. It is notable that combining different kernels increase the size of hidden layers while keeping the same word vector size.

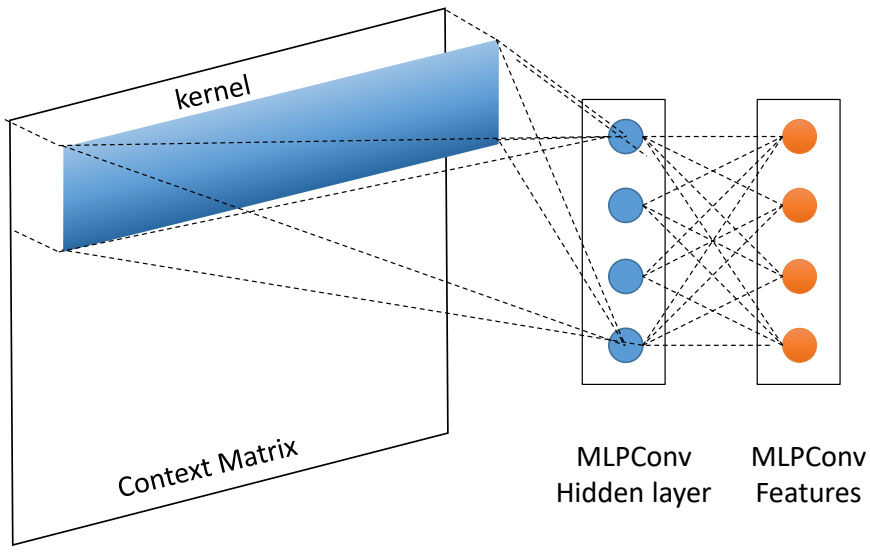

Figure 3.2: MLPConv architecture. The features are learned by a multi-layer perceptron after scanning through the network with convolution. The MLP weights are shared between kernels.

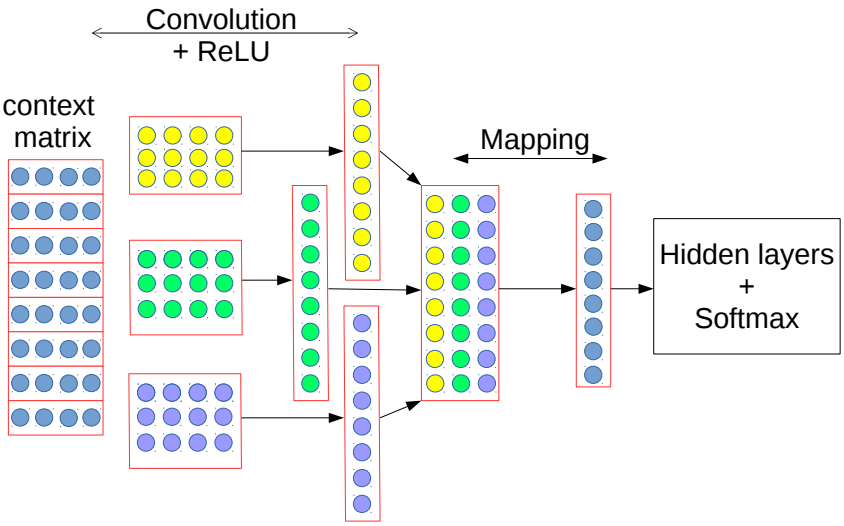

Figure 3.3: Convolutional layer on top of the context matrix.

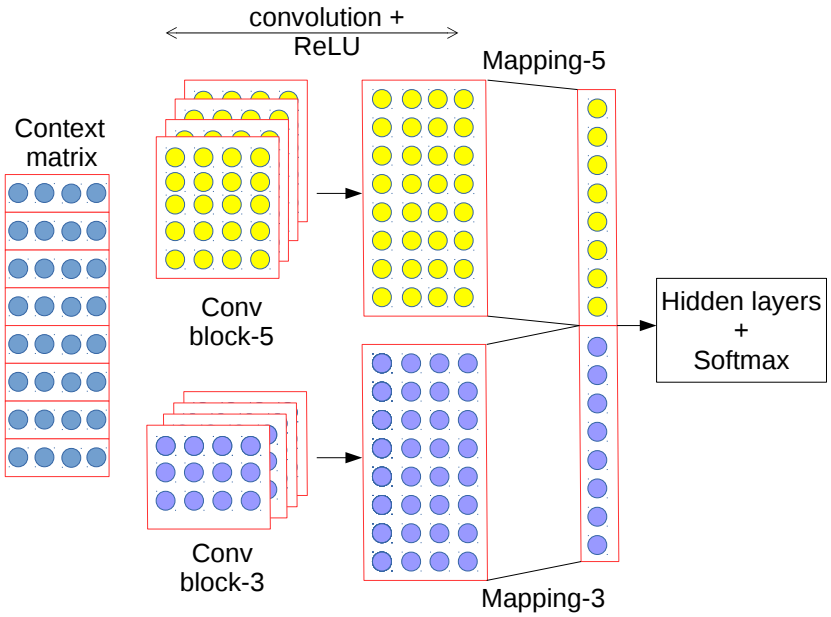

Figure 3.4: Combining kernels with different sizes. We concatenate the outputs of 2 convolutional blocks with kernel size of 5 and 3 respectively.

Chapter 4

Experiments

4.1 Experiment details

4.1.1 Datasets for evaluation

We evaluate our model on three English corpora of different sizes and genres that have been used for language modeling evaluation before. The **Penn Treebank** contains one million words of newspaper text with 10K words in the vocabulary. We reuse the preprocessing and train/test/valid division from [48]. **Europarl-NC** is a a 64-million word corpus that was developed for a Machine Translation shared task [8], combining Europarl data (from parliamentary debates in the European Union) and News Commentary data. We preprocessed the corpus with tokenization and true-casing tools from the Moses toolkit [38]. The vocabulary is composed of words that occur at least 3 times in the training set and contains approximately 60K words.

Finally, we took a fragment of the **ukWaC** corpus which is constructed by crawling UK websites. The fragment contains 200 million words and we extracted a 200k-words vocabulary with the words that appear more than 5 times in the subset. The preprocessing step is similar to Europarl-NC. The validation and test set are different subsets of the ukWaC corpus, both containing 120k words.

4.1.2 Implementation Details

In order to observe the benefit of the convolutional layers, we implemented feed-forward neural language models as baselines, recurrent and long-short term memory models and finally, our convolutional networks. The implementation steps are as follows.

Vocabulary indexing First, the words are converted from string to integers with a mapping table. A sentence $\{W_1, W_2 \dots W_n\}$ is then converted into an array of integers: $\{w_1, w_2 \dots w_n\}$, with w_i being the index of word W_i in the vocabulary. Basically, the corpus (a very long sequence of words including new line tokens separated by space) is transformed into a very long tensor of integer, which is then used for extracting the samples to train our networks. At the end of the each sentence (each line), we put a token "eos" to inform the networks about the sentence boundary, which is a common practice in the literature. Also, it is important that all words that do not exist in the vocabulary are mapped into one token "unk".

Sample extraction In this step, we need to decide the input and output of the networks. For the feed-forward architectures (baseline FFLM and CNN), each input example is a fixed length context and each output is one word only. The extraction process is performed by looping over the corpus, taking each word and its context. In each training iteration (feeding one example or one mini-batch described below to the network), we take one random sample from the training data. During testing, it is not necessary to randomise the order of the samples.

For recurrent architecture, we follow the method of Karpathy et al. [33] to extract the samples for the network. Concretely, we iteratively take a subset of the corpus (the long tensor) and ensure that the next subset is the successor of the current subset. For example, if the sequence length is 5 then the first sample is $\{w_1, w_2, w_3, w_4, w_5\}$ and the next one is $\{w_6, w_7, w_8, w_9, w_{10}\}$. Note that, the RNNs receive one word at one step (w_3 at time 3 for example) and produce the next word (w_4).

Hyper-parameter	Description
k	Embedding size (FFNN, CNN) and kernel size (CNN) or hidden layer size for RNN and LSTM models
dropout	the probability to drop neurons at hidden layers
w	The number of words scanned in each kernel (CNN)
lr	learning rate for SGD
L	number of hidden (recurrent) layers in RNN and LSTM models

Table 4.1: Hyper-parameters to be chosen when training neural network language models.

Batching One important feature in neural network implementation is mini-batching. In the training algorithms described in Chapter 2, we showed the basic computation for propagating one sample forward and backward through the network. Additional performance can be achieved by propagating several examples (a mini-batch) at one through the network, because of two reasons [7]: First, the basic vector-matrix multiplication is transformed into matrix-matrix multiplication, which can be accelerated with Basic Linear Algebra Subprograms (BLAS) libraries or with GPU computing. Second, the distribution of a randomised mini-batch is closer to the distribution of the whole data, which benefits SGD optimisation.

For the feed-forward architecture, a mini-batch is formed by randomly picking several examples together. In that case, the input of the network is a matrix of size $B \times m$. B is the mini-batch size and m is the order of n-grams. The network output is a vector of size B. For recurrent networks, the input and output at each time step is a matrix of size B.

Training details We train our models using Stochastic Gradient Descent (SGD) and we reduce the learning rate by a fixed proportion every time the validation perplexity increases after one epoch. The values for learning rate, learning rate shrinking and mini-batch sizes as well as context size are fixed once and for all based on insights drawn from previous work [16, 24, 66] and through experimentation with the Penn Treebank validation set. Experimentally we found SGD to be efficient and adequately fast, while other learning algorithms involve additional hyper parameters (such as alpha in RMSprop [69]). The set of hyper parameters to be trained for models are showed in Table

Specifically, the learning rate is set to 0.05, with mini-batch size of 128 (we do not take the average of loss over the batch, and the training set is shuffled). We multiply the learning

rate by 0.5 every time we shrink it and clip the gradients if their norm is larger than 12. The network parameters are initialized randomly on a range from -0.01 to 0.01 and the context size is set to 16. In Section 4.3 we show that this large context window is fully exploited.

For the base FFNN and CNN we study embedding sizes (and thus, number of kernels) $k = 128, 256$. For $k = 128$ we explore the simple CNN, incrementally adding NIN and COM variations (in that order) and, alternatively, using a ML-CNN. For $k = 256$, we only explore the former three alternatives. Because of the computational restriction, we chose the kernel size w, stride s and zero-padding z based on experiments in Penntreebank. For the kernel size, we set it to $w = 3$ words for the simple CNN (out of options $3, 5, 7, 9$), whereas for the COM variant we use $w = 3$ and 5. However, we observed the models to be generally robust to this parameter. Dropout rates are tuned specifically for each combination of model and dataset based on the validation perplexity. We also add small dropout $(0.05 - 0.15)$ when we train the networks on the small corpus (Penn Treebank).

The experimental results for recurrent neural network language models, such as Recurrent Neural Networks (RNN) and Long-Short Term Memory (LSTM), on the Penn Treebank are quoted from previous work; for Europarl-NC, we train our own models (we also report the performance of these in-house trained RNN and LSTM models on the Penn Treebank for reference). Specifically, we train LSTMs with embedding size $k = 256$ and number of layers $L = 2$ as well as $k = 512$ with $L = 1, 2$. We train one RNN with $k = 512$ and $L = 2$. To train these models, we use the published source code from [74]. Our own models are also implemented in Torch7[1] for easier comparison.

For all models trained on Europarl-NC and ukWaC, we speed up training by approximating the softmax with Noise Contrastive Estimation (NCE) [23], with the parameters being set following the previous work from [12]. Concretely, for each predicted word, we sample 10 words from the unigram distribution, and the normalization factor is such that $\ln Z = 9$ [2].

For comparison, we also implemented the origial version of the FFNN [4] with two hidden layers with the size of 2 times the embedding size (k). These networks do not have dropout as well as the highway layers in our baseline, yet still share the same number of

[1]The software is available at https://github.com/quanpn90/NCE_CNNLM

[2]We also experimentally tried with Hierarchical Softmax [46] and found out that the NCE gave better performance in terms of speed and perplexity.

parameters.

4.2 Results

Our experimental results are summarized in Table 4.2.

First of all, we can see that even though the FFNN gives a very competitive performance, the addition of convolutional layers is clearly effective to increase it even further[3]. Concretely, we observe a solid 13% reduction of perplexity compared to the feed-forward network after using Network-in-Network in all setups for both corpora. CNN alone yields a 6% improvement, while MLPConv, in line with our expectations, adds another 5% reduction in perplexity. A final (smaller) improvement comes from combining kernels of size 3 and 5, which can be attributed to a more expressive model that can learn patterns of n-grams of different sizes.In contrast to the successful two variants above, the multi-layer CNN did not help in better capturing the regularities of text, but rather the opposite: the more convolutional layers were stacked, the worse the performance. This also stands in contrast to the tradition of convolutional networks in Computer Vision, where using very deep convolutional neural networks is key to having better models. In text, deep convolution for text representation is rather rare, and has only been applied in sentence representation [32]. We conjecture that the reason why deep CNNs may not be so effective for text could be the non-recursive nature of the textual data after convolution. In other words, the convolution output for an image can be construed to be a new image, which yet again can be subject to new convolution operations, whereas the textual counterpart may no longer have the same property.

Regarding the comparison with a stronger LSTM, our models can perform competitively under the same embedding dimension (e.g. see $k = 256$ of $k = 512$) on the first two datasets. However, the LSTM can be easily scaled using larger models, as shown in Zaremba et al. [74], which gives the best known results to date. This is not an option for our model which heavily overfits with large hidden layers (around 1000) even with very large dropout values. Furthermore, the experiments on the larger ukWaC corpus show an

[3]In our experiments, increasing the number of fully connected layers is harmful. Two hidden layers with highway connections is the best setting we could find.

Model	k	w	Penn Treebank			Europarl-NC			ukWaC		
			val	test	#p	val	test	#p	val	test	#p
Vanilla FFNN	128	-	156	147	4.5M	-	-	-	-	-	-
Baseline FFNN	128	-	114	109	4.5M	-	-	-	-	-	-
+CNN	128	3	108	102	4.5M	-	-	-	-	-	-
+MLPConv	128	3	102	97	4.5M	-	-	-	-	-	-
+MLPConv+COM	128	3+5	96	**92**	8M	-	-	-	-	-	-
+ML-CNN (2 layers)	128	3	113	108	8M	-	-	-	-	-	-
+ML-CNN (4 layers)	128	3	130	124	8M	-	-	-	-	-	-
Vanilla FFNN	256	-	161	152	8.2M	-	-	-	-	-	-
Baseline FFNN	256	-	110	105	8.2M	133	174	48M	136	147	156M
+CNN	256	3	104	98	8.3M	112	133	48M	-	-	-
+MLPConv	256	3	97	93	8.3M	107	128	48M	108	**116**	156M
+MLPConv+COM	256	3+5	95	**91**	18M	108	**128**	83M	-	-	-
+MLPConv+COM	512	3+5	96	92	52M	-	-	-	-	-	-
Model	k	L	Penn Treebank			Europarl-NC			ukWaC		
			val	test	#p	val	test	#p	val	test	#p
RNN [48]	300	1	133	129	6M	-	-	-	-	-	-
LSTM [48]	300	1	120	115	6.3M	-	-	-	-	-	-
LSTM [74]	1500	2	82	**78**	48M	-	-	-	-	-	-
LSTM (trained in-house)	256	2	108	103	5.1M	137	155	31M	-	-	-
LSTM (trained in-house)	512	1	123	118	12M	133	149	62M	-	-	-
LSTM (trained in-house)	512	2	94	90	11M	114	**124**	63M	79	**83**	205M
RNN (trained in-house)	512	2	129	121	10M	152	173	61M	-	-	-

Table 4.2: Results on Penn Treebank and Europarl-NC. Figure of merit is perplexity (lower is better). Legend: k: embedding size (also number of kernels for the convolutional models and hidden layer size for the recurrent models); w: kernel size; *val*: results on validation data; *test*: results on test data; *#p*: number of parameters; L: number of layers.

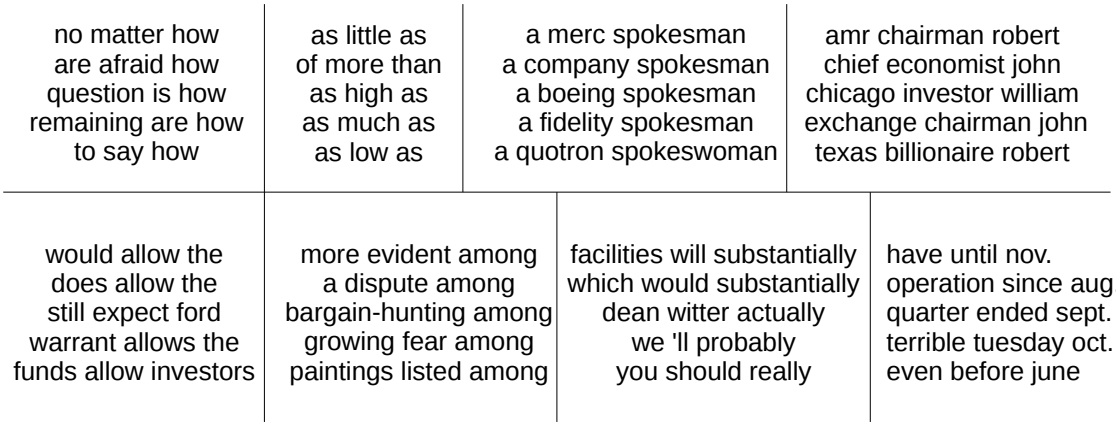

Figure 4.1: Some example phrases that have highest activations for 8 example kernels (each box), extracted from the validation set of the Penn Treebank. Model trained with 256 kernels for 256-dimension word vectors.

advantage for the LSTM, which seems to be more efficient at harnessing this volume of data. On the other hand, the improvement of the convolutional model with respect to the FFNN becomes even more dramatic (more than 20%) using similarly sized models.

To sum up, we have established that the results of our CNN model are well above those of simple feed forward networks and recurrent neural networks. While they are below state of the art LSTMs, they are able to perform competitively with them for small and moderate-size models. Scaling to larger sizes may be today the main roadblock for CNNs to reach the same performances as large LSTMs in language modeling.

4.3 Model Analysis

In what follows, we obtain insights into the inner workings of the CNN by looking into the linguistic patterns that the kernels learn to extract and also studying the temporal information extracted by the network in relation to its prediction capacity.

Learned patterns To get some insight into the kind of patterns that each kernel is learning to detect, we fed trigrams from the validation set of the Penn Treebank to each of the kernels, and extracted the ones that most highly activated the kernel. Some examples are

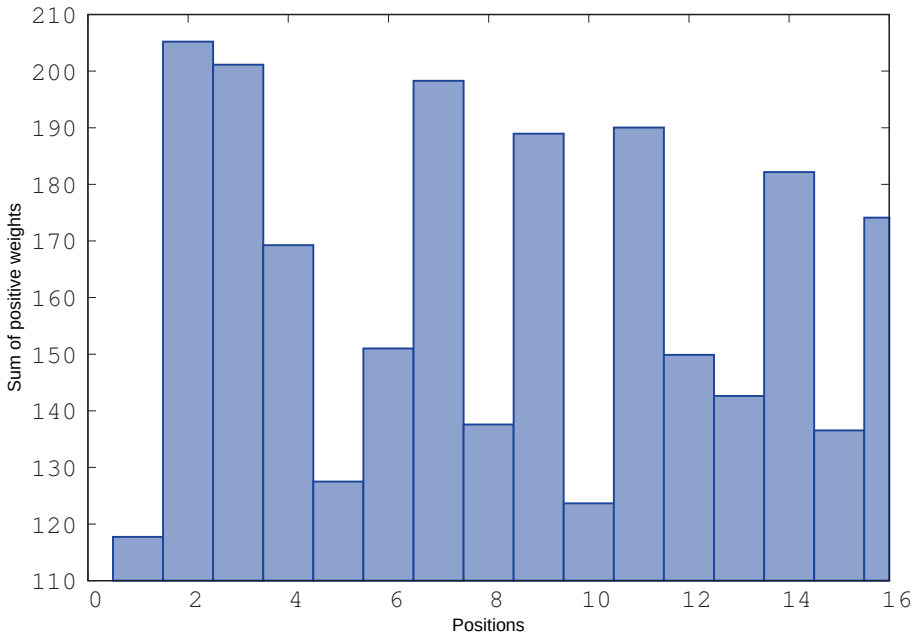

Figure 4.2: The distribution of positive weights over context positions, where 1 is the position closest to the predicted word.

shown in Figure 4.1. Since the word windows are made of embeddings, we can expect patterns with similar embeddings to have close activation outputs. This is borne out in the analysis: The kernels specialize in distinct features of the data, including more syntactic-semantic constructions (cf. the "comparative kernel" including *as* ... *as* patterns, but also *of more than*) and more lexical or topical features (cf. the "ending-in-month-name" kernel). Even in the more lexicalized features, however, we see linguistic regularities at different levels being condensed in a single kernel: For instance, the "spokesman" kernel detects phrases consisting of an indefinite determiner, a company name (or the word *company* itself) and the word "spokesman". We hypothesize that the convolutional layer adds an "I identify one specific feature, but at a high level of abstraction" dimension to a feed-forward neural network, similarly to what has been observed in image classification [39].

Temporal information To the best of our knowledge, the longest context used in feed-forward language models is 10 [25], where no significant change in terms of perplexity was observed for bigger context sizes, even though in that work only same-sentence contexts

were considered. In our experiments, we use a larger context size of 16 while removing the sentence boundary limit (as commonly done in n-gram language models) such that the network can take into account the words in the previous sentences.

To analyze whether all this information was effectively used, we took our best model, the CNN-NIN-COM model with embedding size of 256 (fourth line, second block in Table 4.2), and we identified the weights in the model that map the convolutional output (of size $n \times k$) to a lower dimensional vector (the "mapping" layer in Figure 3.3). Recall that the output of the convolutional layer is a matrix indexed by time step and kernel index representing the activation of the kernel when convolved with a window of text centered around the given time step. Thus, output units of the above mentioned mapping predicate over an ensemble of kernel activations for each time-step . We can identify the patterns that they learn to detect by extracting the time-kernel combinations for which they have positive weights (since we have ReLU activations, negative weights are equivalent to ignoring a feature). First, we asked ourselves whether these units tend to be more focused on the time-steps closer to the target or not. To test this, we calculated the sum of the positive weights for each position in time using an average of the mappings that correspond to each output unit. The results are shown in Figure 4.2. As could be expected, positions that are close to the token to be predicted have many active units (local context is very informative; see positions 2-4). However, surprisingly, positions that are actually far from the target are also quite active (see positions 10-14, with a spike at 11). It seems like the CNN is putting quite a lot of effort on characterizing long-range dependencies.

Next, we checked that the information extracted from the positions that are far in the past are actually used for prediction. To measure this, we artificially lesioned the network so it would only read the features from a given range of timesteps (words in the context). To lesion the network we manually masked the weights of the mapping that focus on times outside of the target range by setting them to zero. We started using only the word closest to the final position and sequentially unmasked earlier positions until the full context was used again. The result of this experiment is presented in Figure 4.3, and it confirms our previous observation that positions that are the farthest away contribute to the predictions of the model. The perplexity drops dramatically as the first positions are unmasked, and then decreases more slowly, approximately in the form of a power law ($f(x) \propto x^{-0.9}$). Even

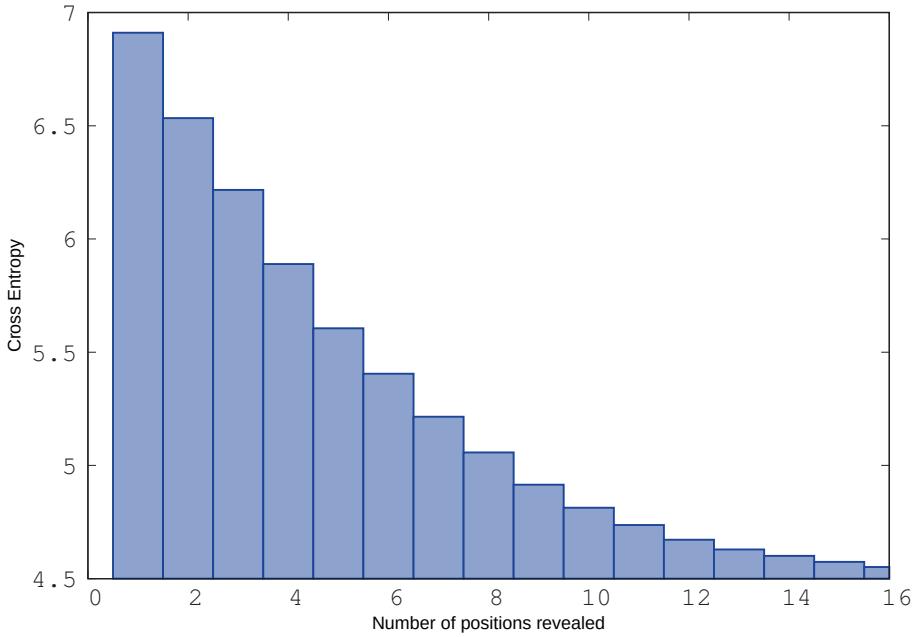

Figure 4.3: Perplexity change over position, by incrementally revealing the Mapping's weights corresponding to each position.

though the effect is smaller, the last few positions still contribute to the final perplexity.

Chapter 5

Related Work

Provided our proposed models for language modeling using convolutional neural networks, we cover the related works that apply convolutional neural networks in NLP problems.

Time-delay neural networks or convolutional neural networks (CNNs) were originally designed to deal with hierarchical representation in signal processing [71] and computer vision [42]. Deep networks with many stacked convolutional layers have been successfully applied in image classification and understanding [26, 63]. In such systems the convolutional kernels manage to learn to detect visual features at both local and more abstract levels.

In NLP, CNNs have been mainly applied to static classification task for discovering latent structures in text. Kim [34] use a CNN to tackle sentence classification, with competitive results. This work also introduces kernels with varying window sizes to learn complementary features at different aggregation levels. Kalchbrenner et al. [32] propose a convolutional architecture for sentence representation that vertically stacks multiple convolution layers, each of which can learn independent convolution kernels. CNNs with similar structures have also been applied to other classification tasks, such as semantic matching [30], relation extraction [52] and information retrieval [62].

In contrast, Collobert et al. [14] explore a CNN architecture to solve various sequential and non-sequential NLP tasks such as part-of-speech tagging, named entity recognition and also language modeling. This is perhaps the work that is closest to ours in the existing literature. However, their model differs from ours in that it uses a **max-pooling** layer that

picks the most activated feature across time, thus ignoring temporal dependencies, whereas we explicitly avoid doing so. More importantly, the language models trained in this work are only evaluated through downstream tasks and through the quality of the learned word embeddings, but not on the sequence prediction task itself.

Besides being applied on word-based sequences, the convolutional layers have also been used to model sequences at the character level. Kim et al. [35] propose a recurrent language model that replaces the word-indexed projection matrix with a convolution layer fed with the character sequence that constitutes each word to find morphological patterns. The main difference between their and our work is that we consider words as the smallest linguistic unit, and thus apply the convolutional layer at the word level. Also, convolutional layers can be applied to a full sentence whose fundamental units are characters [76]. State-of-the-art results in text classification has been observed by using very deep convolutional neural networks [15].

In this work we focus on statistical language modeling, which differs from most of the tasks where CNNs have been applied before in multiple ways. First, the input normally consists of incomplete sequences of words rather than complete sentences. Second, as a classification problem, it features an extremely large number of classes (the words in a large vocabulary). Finally, temporal information, which can be safely discarded in many settings with little impact in performance, is critical here: An n-gram appearing close to the predicted word may be more informative, or yield different information, than the same n-gram appearing several tokens earlier.

Chapter 6

Conclusion

In this work, we have investigated the use of Convolutional Neural Networks for language modeling, a sequential prediction task. We incorporate a CNN layer on top of a strong feed-forward model enhanced with modern techniques like Highway Layers and Dropout. Our results show a solid 11-26% improvement in perplexity with respect to the feed-forward model across two corpora of different sizes and genres when the model uses Network-in-Network and combine kernels of different window sizes. However, even without these additions we show CNNs to effectively learn language patterns to significantly decrease the model perplexity.

In our view, this improvement responds to two key properties of CNNs, highlighted in the analysis. First, as we have shown, they are able to integrate information from larger context windows, using information from words that are as far as 16 positions away from the predicted word. Second, as we have qualitatively shown, the kernels learn to detect specific patterns at a high level of abstraction. This is analogous to the role of convolutions in Computer Vision. The analogy, however, has limits; for instance, a deeper model stacking convolution layers harms performance in language modeling, while it greatly helps in Computer Vision. We conjecture that this is due to the differences in the nature of visual vs. linguistic data. The convolution creates sort of abstract images that still retain significant properties of images. When applied to language, it detects important textual features but distorts the input, such that it is not text anymore.

As for recurrent models, even if our model outperforms RNNs, it is well below state-of-the-art LSTMs. Since CNNs are quite different in nature, we believe that a fruitful line of future research could focus on integrating the convolutional layer into a recurrent structure for language modeling, as well as other sequential problems, perhaps capturing the best of both worlds.

Appendix A

Publications by Author

- 2016

  Ngoc-Quan Pham, Gemma Boleda and Germán Kruszewski. "Convolutional Neural Network Language Models." In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, Texas Austin PA: ACL, 2016.

  Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, **Ngoc-Quan Pham**, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda and Raquel Fernández, "The LAMBADA dataset: Word prediction requiring a broad discourse context.", In *Proceedings of ACL 2016 (54th Annual Meeting of the Association for Computational Linguistics)*, East Stroudsburg PA: ACL, 2016.

- 2015

  Ngoc-Quan Pham and Lonneke van der Plas, "Predicting pronouns across languages with continuous word spaces.", in *Proceedings of the Second Workshop on Discourse in Machine Translation*, pages 101 - 107, Lisbon, Portugal, 2015.

Bibliography

[1] Ossama Abdel-Hamid, Abdel-rahman Mohamed, Hui Jiang, and Gerald Penn. Applying convolutional neural networks concepts to hybrid nn-hmm model for speech recognition. In *2012 IEEE international conference on Acoustics, speech and signal processing (ICASSP)*, pages 4277–4280. IEEE, 2012.

[2] 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.

[3] 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.

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

[5] Yoshua Bengio, Holger Schwenk, Jean-Sébastien Senécal, Fréderic Morin, and Jean-Luc Gauvain. Neural probabilistic language models. In *Innovations in Machine Learning*, pages 137–186. Springer, 2006.

[6] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8624–8628. IEEE, 2013.

[7] Jeff Bilmes, Krste Asanovic, Chee-Whye Chin, and Jim Demmel. Using phipac to speed error back-propagation learning. In *Acoustics, Speech, and Signal Processing,*

1997. ICASSP-97., 1997 IEEE International Conference on, volume 5, pages 4153–4156. IEEE, 1997.

[8] Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Barry Haddow, Matthias Huck, Chris Hokamp, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Carolina Scarton, Lucia Specia, and Marco Turchi. Findings of the 2015 workshop on statistical machine translation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 1–46, Lisbon, Portugal, September 2015. Association for Computational Linguistics. URL `http://aclweb.org/anthology/W15-3001`.

[9] Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

[10] Ciprian Chelba and Frederick Jelinek. Structured language modeling. *Computer Speech & Language*, 14(4):283–332, 2000.

[11] Stanley F Chen and Joshua Goodman. An empirical study of smoothing techniques for language modeling. *Computer Speech & Language*, 13(4):359–394, 1999.

[12] Xie Chen, Xunying Liu, Mark JF Gales, and Philip C Woodland. Recurrent neural network language model training with noise contrastive estimation for speech recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5411–5415. IEEE, 2015.

[13] 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.

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

[15] Alexis Conneau, Holger Schwenk, Loïc Barrault, and Yann Lecun. Very deep convolutional networks for natural language processing. *arXiv preprint arXiv:1606.01781*, 2016.

[16] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard M Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *ACL (1)*, pages 1370–1380. Citeseer, 2014.

[17] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[18] Marcello Federico, Nicola Bertoldi, and Mauro Cettolo. Irstlm: an open source toolkit for handling large scale language models. In *Interspeech*, pages 1618–1621, 2008.

[19] Denis Filimonov and Mary Harper. A joint language model with fine-grain syntactic tags. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3-Volume 3*, pages 1114–1123. Association for Computational Linguistics, 2009.

[20] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[21] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

[22] Jiuxiang Gu, Zhenhua Wang, Jason Kuen, Lianyang Ma, Amir Shahroudy, Bing Shuai, Ting Liu, Xingxing Wang, and Gang Wang. Recent advances in convolutional neural networks. *CoRR*, abs/1512.07108, 2015.

[23] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *AISTATS*, volume 1, page 6, 2010.

[24] Le Hai Son, Ilya Oparin, Alexandre Allauzen, Jean-Luc Gauvain, and François Yvon. Structured output layer neural network language model. In *Acoustics, Speech and*

Signal Processing (ICASSP), 2011 IEEE International Conference on, pages 5524–5527. IEEE, 2011.

[25] Le Hai Son, Alexandre Allauzen, and François Yvon. Measuring the influence of long range dependencies with neural network language models. In *Proceedings of the NAACL-HLT 2012 Workshop: Will We Ever Really Replace the N-gram Model? On the Future of Language Modeling for HLT*, pages 1–10. Association for Computational Linguistics, 2012.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[27] Kenneth Heafield. Kenlm: Faster and smaller language model queries. In *Proceedings of the Sixth Workshop on Statistical Machine Translation*, pages 187–197. Association for Computational Linguistics, 2011.

[28] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580, 2012. URL `http://arxiv.org/abs/1207.0580`.

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

[30] Baotian Hu, Zhengdong Lu, Hang Li, and Qingcai Chen. Convolutional neural network architectures for matching natural language sentences. In *Advances in Neural Information Processing Systems*, pages 2042–2050, 2014.

[31] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[32] Nal Kalchbrenner, Edward Grefenstette, and Phil Blunsom. A convolutional neural network for modelling sentences. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014, June 22-27, 2014, Baltimore,*

MD, USA, Volume 1: Long Papers, pages 655–665, 2014. URL http://aclweb.org/anthology/P/P14/P14-1062.pdf.

[33] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[34] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[35] Yoon Kim, Yacine Jernite, David Sontag, and Alexander M. Rush. Character-aware neural language models. *CoRR*, 2015. URL http://arxiv.org/abs/1508.06615.

[36] Zipf George Kingsley. Selective studies and the principle of relative frequency in language, 1932.

[37] Reinhard Kneser and Hermann Ney. Improved backing-off for m-gram language modeling. In *Acoustics, Speech, and Signal Processing, 1995. ICASSP-95., 1995 International Conference on*, volume 1, pages 181–184. IEEE, 1995.

[38] Philipp Koehn, Hieu Hoang, Alexandra Birch, Chris Callison-Burch, Marcello Federico, Nicola Bertoldi, Brooke Cowan, Wade Shen, Christine Moran, Richard Zens, et al. Moses: Open source toolkit for statistical machine translation. In *Proceedings of the 45th annual meeting of the ACL on interactive poster and demonstration sessions*, pages 177–180. Association for Computational Linguistics, 2007.

[39] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[40] Quoc V Le, Navdeep Jaitly, and Geoffrey E Hinton. A simple way to initialize recurrent networks of rectified linear units. *arXiv preprint arXiv:1504.00941*, 2015.

[41] B Boser Le Cun, John S Denker, D Henderson, Richard E Howard, W Hubbard, and Lawrence D Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*. Citeseer, 1990.

[42] Yann LeCun and Yoshua Bengio. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks*, 3361(10):1995, 1995.

[43] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[44] James Martens and Ilya Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1033–1040, 2011.

[45] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, volume 2, page 3, 2010.

[46] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Honza Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5528–5531. IEEE, 2011.

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

[48] Tomas Mikolov, Armand Joulin, Sumit Chopra, Michael Mathieu, and Marc'Aurelio Ranzato. Learning longer memory in recurrent neural networks. *arXiv preprint arXiv:1412.7753*, 2014.

[49] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. *arXiv preprint arXiv:1206.6426*, 2012.

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

[51] Hermann Ney, Ute Essen, and Reinhard Kneser. On structuring probabilistic dependences in stochastic language modelling. *Computer Speech & Language*, 8(1):1–38, 1994.

[52] Thien Huu Nguyen and Ralph Grishman. Relation extraction: Perspective from convolutional neural networks. In *Proceedings of NAACL-HLT*, pages 39–48, 2015.

[53] Thomas R Niesler and Philip C Woodland. A variable-length category-based n-gram language model. In *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceedings., 1996 IEEE International Conference on*, volume 1, pages 164–167. IEEE, 1996.

[54] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *ICML (3)*, 28:1310–1318, 2013.

[55] Vu Pham, Théodore Bluche, Christopher Kermorvant, and Jérôme Louradour. Dropout improves recurrent neural networks for handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 285–290. IEEE, 2014.

[56] Ronald Rosenfeld. Two decades of statistical language modeling: Where do we go from here. In *Proceedings of the IEEE*, page 2000, 2000.

[57] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[58] Ivan A Sag, Timothy Baldwin, Francis Bond, Ann Copestake, and Dan Flickinger. Multiword expressions: A pain in the neck for nlp. In *International Conference on Intelligent Text Processing and Computational Linguistics*, pages 1–15. Springer, 2002.

[59] Holger Schwenk. Continuous space language models. *Computer Speech & Language*, 21(3):492–518, 2007.

[60] Jean Sébastien, Kyunghyun Cho, Roland Memisevic, and Yoshua Bengio. On using very large target vocabulary for neural machine translation. 2015.

[61] Thomas Serre, Lior Wolf, Stanley Bileschi, Maximilian Riesenhuber, and Tomaso Poggio. Robust object recognition with cortex-like mechanisms. *IEEE transactions on pattern analysis and machine intelligence*, 29(3):411–426, 2007.

[62] Yelong Shen, Xiaodong He, Jianfeng Gao, Li Deng, and Grégoire Mesnil. A latent semantic model with convolutional-pooling structure for information retrieval. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, pages 101–110. ACM, 2014.

[63] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[64] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[65] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015. URL `http://arxiv.org/abs/1505.00387`.

[66] Sainbayar Sukhbaatar, Jason Weston, Rob Fergus, et al. End-to-end memory networks. In *Advances in Neural Information Processing Systems*, pages 2431–2439, 2015.

[67] 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.

[68] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[69] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2), 2012.

[70] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *EMNLP*, pages 1387–1392. Citeseer, 2013.

[71] Alexander Waibel, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. Phoneme recognition using time-delay neural networks. *Acoustics, Speech and Signal Processing, IEEE Transactions on*, 37(3):328–339, 1989.

[72] Ian H Witten and Timothy C Bell. The zero-frequency problem: Estimating the probabilities of novel events in adaptive text compression. *Ieee transactions on information theory*, 37(4):1085–1094, 1991.

[73] Wojciech Zaremba. An empirical exploration of recurrent network architectures. 2015.

[74] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[75] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[76] Xiang Zhang and Yann LeCun. Text understanding from scratch. *CoRR*, abs/1502.01710, 2015. URL `http://arxiv.org/abs/1502.01710`.