A SURVEY ON LANGUAGE MODELING USING NEURAL NETWORKS

Nikolaos Pappas Thomas Meyer[a]

Idiap-RR-32-2012

NOVEMBER 2012

[a]Idiap Research Institute, P.O. Box 592, CH-1920 Martigny

A Survey on Language Modeling using Neural Networks

Nikolaos Pappas and Thomas Meyer
Idiap Research Institute and EPFL/EDEE doctoral school
Martigny and Lausanne, Switzerland
`(nikolaos.pappas|thomas.meyer)@idiap.ch`

May 25, 2012

Abstract

A Language Model (LM) is a helpful component of a variety of Natural Language Processing (NLP) systems today. For speech recognition, machine translation, information retrieval, word sense disambiguation etc., the contribution of an LM is to provide features and indications on the probability of word sequences, their grammaticality and semantical meaningfulness. What makes language modeling a challenge for Machine Learning algorithms is the sheer amount of possible word sequences: the curse of dimensionality is especially encountered when modeling natural language. The survey will summarize and group literature that has addressed this problem and we will examine promising recent research on Neural Network techniques applied to language modeling in order to overcome the mentioned curse and to achieve better generalizations over word sequences.

1 Introduction

A language model (LM) is a statistical model that assigns a probability to a sequence of words by generating a probability distribution. This is useful in a variety of Natural Language Processing (NLP) tasks. For example, speech recognizers profit from a probability assigned to the next word in a speech sequence to be predicted. Similarly, Part-of-Speech (POS) taggers, syntactical parsers or word sense disambiguation systems all include features which an LM can provide and would inform the systems of tags (generalizations over words) and/or the word embeddings in the context of a word sequence. In Machine Translation systems, an LM is used to tune the probability scores of outputs by the system in order to improve for the actual grammaticality and fluency of a translation in the target language.

However, as there is a potentially infinite possible combination of words in natural language, it is difficult to model generalizations in training or testing time that is still practical for real-time processing systems. In most applications, language modeling has therefore been solved by using a statistical model of relative frequency counts. The so-called n-gram models count uni-, bi- or tri-grams (sometimes also 4- and 5-grams), i.e. a word in a sequence is predicted based on its preceding context (history) of 1 to 4 words.

This approach is accompanied by several problems: it explicitly matches words or strings and it is very likely that quite a big amount of the n-grams are not seen during training even with very large corpora (i.e. millions of sentences). This leads to data sparsity and possible over-fitting of the training data and to the related problem that unseen word sequences cannot be assigned a probability during testing time. A variety of smoothing techniques has been developed that currently are the state-of-the-art for n-gram language modeling and that allow for correcting/generalizing the relative frequency counts of the seen n-grams.

However, what would actually be the goal of language modeling is to recognize that word sequences such as *he has seen the man walking in the park* to be grammatically and semantically similar to *she has seen a woman running in the park*. POS tagging or syntactical parsing would provide generalizations over such sequences (you model sequences of tags instead of explicit words), but they themselves need costly manual annotation for supervised learning and engineered linguistic (expert) features.

In the 1980s and early 90s Artificial Neural Networks (NN) were a popular learning technique that was capable of learning such features automatically during training (not using any feature engineering or manually annotated class labels). The idea was to mimic the learning procedure of the human brain with connected neurons that are activated or not during a learning phase. NNs however lost their popularity due to immense amounts of training time one has needed back then. With the advent of faster and parallel computer architectures in recent years, they however regained success and have been further developed to multilayer, recurrent and/or deep neural networks that perform at the state-of-the-art of other machine learning tasks including language modeling, where the better generalizations over word sequences provided by an NN model outperform the n-gram models.

This survey provides a close look at training and testing methods of NNs applied to language modeling. We introduce into both fields, Section 2.1 for NNs and Section 2.2 for LMs. The main section of the survey, Section 3, then subsumes several NN-LM techniques, an overview of the basic formalisms and architectures of the first Neural Network Language Models (NN-LM) is given in Section 3.2. The following subsections describe advanced methods to be used for large training data (Section 3.3.1), combinations with n-gram modeling (Section 3.3.3) or related methods that have been investigated (Section 3.3.4). Section 3.4 looks at state-of-the-art NN-LM methods before the survey is concluded in Section 4 by outlining remaining challenges.

2 Background

2.1 Neural Networks

The concept of Neural Network (NN) appears to have first been proposed by Alan Turing in his 1948 paper 'Intelligent Machinery' and traditionally refers to a network or circuit of biological neurons [25]. In the field of Artificial Intelligence the neurons of an NN are replaced by artificial neurons, and the network in this case is called Artificial Neural Network (ANN) or Simulated Neural Network (SNN). ANN is an interconnected group of artificial neurons that uses mathematical or computational models to process the information exploiting the interconnections between neurons.

The Artificial Neural Networks can be divided in two major categories:

- **Feedforward:** This was the first type of artificial neural networks seen in the literature. The connections between the units or neurons do not form a directed cycle. This means that the information in the network flows only in one direction, forward from the input nodes to the hidden nodes and finally to the output nodes. There are no cycles or loops of information happening in this type of ANNs.
- **Recurrent:** Recurrent Neural Networks (RNN) include the state-of-the-art methods in the field of neural networks nowadays. Here, the connections between the units or neurons form a directed cycle (in contrast to the feedforward ANNs). This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. This type of ANNs can use their internal memory to process arbitrary sequences of inputs (in contrast to the feedforward ANNs).

In the following sections we give some background on these major categories, emphasizing more the category of the feedforward neural networks which is the most prominent in the literature of neural networks applied to language modeling.

2.1.1 Feedforward Neural Networks

2.1.2 Perceptron

The perceptron is an online supervised learning algorithm for binary classification introduced in 1957 by Rosenblatt [42] and is able to find a separating hyperplane for the input instances if they are linearly separable. For the representation of the hyperplane, it uses a linear predictor function combining a set of weights with the feature vector of the given input. Formally, a perceptron maps an input real-valued vector of attributes x to an output value $f(x)$ which is a single binary value:

$$f(X) = \begin{cases} 1, & \text{if } w \cdot x + b > 0 \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

where w is a vector of real-valued weights, $w \cdot x$ is the dot product (or simply the linear combination of the attributes $x_0, x_1, ..., x_n$ and the weights $w_0, w_1, ..., w_n$) and b is a constant parameter called 'bias'. Geometrically, the bias alters the position of the decision boundary but cannot alter its orientation. Formula 1 is used in order to classify unseen instances, if the output is 0 then the instance belongs to the negative class and when the output is 1 it belongs to the positive one.

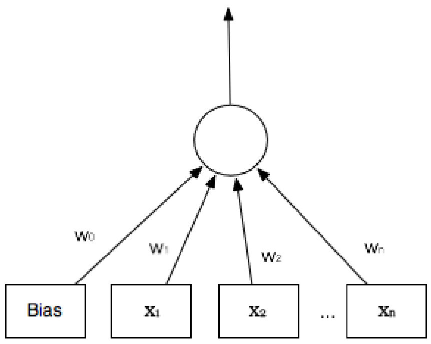

Figure 1: Representation of a perceptron as a neural network. We can observe two layers: the input layer where there is one node for every attribute, plus the bias node (set to 1) and one output node. Every node is connected to the output node and the connections between the nodes are weighted.

Algorithm 1 gives the perceptron a learning rule for finding a separating hyperplane. The algorithm iterates until a perfect solution has been found but as was mentioned before, it works properly only when the data is linearly separable. At each iteration it goes through all the training instances and if a misclassified instance is encountered, the parameters (i.e. weights) of the hyperplane are changed so that the instance moves closer to the hyperplane or even to the correct side of it. If the encountered instance belongs to the positive class, this is done by adding the attribute values to the weight vector, otherwise if it belongs to the negative class, the attribute values are subtracted from it.

At this point the attribute values are multiplied by factor a such that $0 < a \leq 1$ which is called 'learning rate'. Note that the decision boundary of a perceptron is invariant with respect to scaling of the weight vector, i.e. a perceptron trained with initial weight vector W and learning rate a is an identical estimator to a perceptron trained with initial weight vector w/a and learning rate 1. Thus, since the initial weights become irrelevant with increasing number of iterations, the learning rate does not matter in the case of the perceptron and is usually just set to one.

Algorithm 1 Learning algorithm

```
Initialize weights in $w$ to zero, set threshold $b$ and learning rate $a$.
while not all instances are classified correctly do
    for each sample $x$, class $y$ in the training data do
        if $f(x)! = y$ ($x$ is classified incorrectly by the perceptron) then
            if $y == 1$ ($x$ belongs to positive class) then
                $w = w + x * a$ (add it to the weight vector)
            else
                $w = w - x * a$ (substract it from the weight vector)
```

2.1.3 Kernel Perceptron

The algorithm of the perceptron, as mentioned before, can only be used for learning a linear classifier. What about the non-linear case? It turns out that the kernel trick which was introduced in 1964 by Aizerman [1], can also be used to upgrade the perceptron algorithm to learn non-linear decision boundaries. We can consider a similar decision function to Formula 1 as follows:

$$f(X) = \begin{cases} 1, & \text{if } w \cdot \phi(x) > 0 \\ -1, & \text{otherwise} \end{cases} \tag{2}$$

The non-linearity can be achieved by hand-crafting a non-linear $\phi(\cdot)$. The problem here is that the calculation of the dot product in high dimensions is computationally intensive. Lets consider an update during the execution of the learning rule:

$$w^{t+1} = w^t + \begin{cases} y^t \phi(x^t), & \text{if } y^t w \cdot \phi(x^t) \leq 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

where y^t is the class value (+1 or -1) of the t^{th} instance. To implement this expression of dot products we no longer keep track of an explicit weight vector. Instead, we simply store the instances that have been misclassified so far. The decision function at the t^{th} example can be written as:

$$f^t(x) = \sum_{t \in \text{`}updated'} y^t \phi(x^t) \cdot \phi(x) \tag{4}$$

This dot product reminds of a similar expression for Support Vector Machines (SVM) that enabled the use of kernels. Indeed, we can apply the same trick here and use a kernel function instead of the dot product. Using the kernel the Formula 4 can be rewritten as:

$$K(x, x^t) = \phi(x^t) \cdot \phi(x) \ ; \ f^t(x) = \sum_{t \in \text{`}updated'} y^t K(x, x^t) \tag{5}$$

In this way the perceptron algorithm can learn a non-linear classifier simply by keeping track of instances that have been misclassified during the training process using Formula 5 to form each prediction.

2.1.4 Variants of Perceptron

The perceptron algorithm does not guarantee to find the largest separating margin between the classes. Therefore, a variation, the so-called 'perceptron of optimal stability' or 'margin perceptron' has been developed that aims at finding the largest separating margin by means of iterative training and optimization schemes. Examples of this type of perceptron are the Min-Over algorithm (Krauth and Mezard, 1987) [30] and the AdaTron (Anlauf and Biehl, 1989) [2]. The perceptron of optimal stability is, together with the kernel trick, one of the conceptual foundations of support vector machines.

The pocket algorithm with ratchet (Gallant, 1990) [20] solves the stability problem of perceptron learning by keeping the best solution seen so far 'in its pocket'. The pocket algorithm then returns the solution in the pocket, rather than the last solution. It can be used also for non-separable data sets, where the aim is to find a perceptron with a small number of misclassifications.

2.1.5 Multilayer Perceptron

Using a kernel is not the only way to create a non-linear classifier based on a perceptron. Before the introduction of the kernel trick, people working on neural networks used a different approach for non-linear classification: they connected many perceptron-like models in a hierarchical structure to represent non-linear boundaries. A Multilayer Perceptron (MLP) is a feedforward artificial neural network that consists of multiple layers of nodes in a directed graph, with each graph fully connected to the next one. The layers between the first and the last are usually called hidden layers. The main goal of an MLP is to map sets of input data onto appropriate output. All the internal nodes except the input nodes are artificial neurons with a non-linear activation function (see below). In order to train the network, an MLP uses a supervised learning method called backpropagation and it is considered as a modification of the standard linear perceptron which can distinguish data that is not linearly separable.

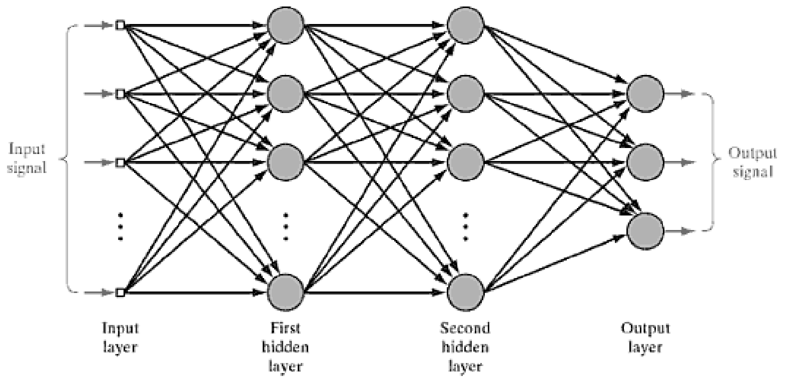

Figure 2: Architectural representation of a multilayer perceptron with two hidden layers. We can observe three layers of nodes, the first layer receives the input vector, then the feedforward mechanism activates the nodes to the other layers and finally the output vector is obtained.

The activation mechanism has the purpose to determine whether or not a neuron fires. In contrast to the simple perceptron which uses a linear activation mechanism, a multilayer perceptron's each neuron uses a non-linear activation function which was developed to model the frequency of action potentials, or firing, of biological neurons in the brain. This function is modeled in several ways, but must always be normalizable and differentiable. The most common functions used for this purpose are sigmoid functions:

$$\phi(y_i) = tanh(u_i) \ ; \ \phi(y_i) = (1 + e^{u_{i]}})^{-1} \tag{6}$$

in which the former function is a hyperbolic tangent which ranges from -1 to 1, and the latter, the logistic function, is similar in shape but ranges from 0 to 1. Here y_i is the output of the j^{th} node (neuron) and u_i is the weighted sum of the input synapses. In the case of the multilayer perceptron consisting of three or more layers, each node in one layer is connected with a certain weight w_{ij} to every node in the following layer. The weights between the internal nodes can be calculated by using backpropagation through time. Backpropagation is described in the following subsection.

2.1.6 Backpropagation

Given a fixed number of connections in the neural network, we need to determine the appropriate weights for each of them. In the case of multilayer perceptron the perceptron learning rule cannot be applied because there are hidden nodes in the network and the correct outputs of those nodes are unknown. The solution is to modify the weights of the connections leading to the hidden units based on the strength of each unit's contribution to the final prediction. Therefore, for the training of the network we use the backpropagation algorithm which is a standard mathematical optimization algorithm and a common method for minimizing an objective function. Backpropagation was firstly described as a multi-stage dynamic system optimization (Arthur E. Bryson and Yu-Chi Ho 1969) [12]. This method is mostly useful for feedforward neural networks and requires that the activation function of the nodes or neurons and the error function are differentiable. As we mentioned earlier, the activation function is a sigmoid function which is differentiable and the modeling of the error can be made using the squared-error loss function which is widely known.

Formally, we represent the error in output node j in the n^{th} data point by $e_j(n) = y_j(n) - f_j(n)$ where y is the target value and f is the value produced by the perceptron. We then make corrections to the weights of the nodes based on those which minimize the error in the entire output, given by the formula:

$$\mathcal{E}(n) = \frac{1}{2}\sum_j e_j^2(n) \tag{7}$$

Using the above error function we can apply gradient descent in order to minimize the error function and find our change in the weights to follow the formula:

$$\Delta W_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial u_j(n)} f_i(n) \tag{8}$$

where $f_i(n)$ is the output of the previous neuron and η is the learning rate, which is carefully selected to ensure that the weights converge to a response fast enough, without producing oscillations. The derivative to be calculated depends on the induced local field u_i (the weighted sum of the input synapses), which itself varies and it can be simplified as shown below:

$$-\frac{\partial \mathcal{E}(n)}{\partial u_j(n)} = e_j(n)\phi'(u_j(n)) \tag{9}$$

where ϕ' is the derivative of the activation function described above, which itself does not vary. The analysis is more difficult for the change in weights to a hidden node, but it can be shown that the relevant derivative is:

$$-\frac{\partial \mathcal{E}(n)}{\partial u_j(n)} = \phi'(u_j(n)) \sum_k -\frac{\partial \mathcal{E}(n)}{\partial u_k(n)} w_{kj}(n) \tag{10}$$

This depends on the change in weights of the k^{th} nodes, which represent the output layer. So to change the hidden layer weights, we must first change the output layer weights according to the derivative of the activation function, and so this algorithm represents a backpropagation of the activation function.

As mentioned for equation 7, an important observation is that gradient descent can be applied to figure out the slope of the error function at any particular point (based on the derivative). When the derivative is negative, the function slopes downwards and right, if positive it slopes downwards to the left and the size of the derivative determines the steepness. Gradient descent is an iterative optimization procedure that takes the value of the derivative, multiplies it with the small learning rate constant and subtracts the result from the current parameter value. This is repeated for each new parameter value and converges at a minimum. The latter however can be local and not the actual optimal, global minimum of the function. Also, depending on the amount of parameter values (can be large in Neural Networks), gradient descent might take many iterations and slows down the training process considerably. A partial solution to this is to use so-called 'Stochastic Gradient Descent', that picks a random training instance per iteration and updates the parameters based on this example only [9].

2.1.7 Recurrent Neural Networks

A Recurrent Neural Network (RNN) is a straightforward adaptation of the standard feedforward neural network to allow it to model *sequential* data. RNNs include a high dimensional hidden state and non-linear activation functions, endow great expressive power and rich dynamics, enabling the hidden state of the RNN to integrate information over many timesteps and they can be used for accurate predictions. In Figure 3 the three layers of the RNN are displayed (input, hidden, output) along with the connections between the units. At each timestep, the RNN receives an input, updates its hidden state, and makes a prediction.

Specifically, an RNN can be formalized as follows: Given a sequence of input vectors $x_1, ..., x_T$, the RNN computes a sequence of hidden states $h_1, ..., h_T$ and a sequence of outputs $y_1, ..., y_T$ by iterating the following equations from $t = 1$ to T:

$$h_t = tanh(W_{hx}x_t + W_{hh}h_t + b_h) \; ; \; y_t = W_{yh}h_t + b_y \tag{11}$$

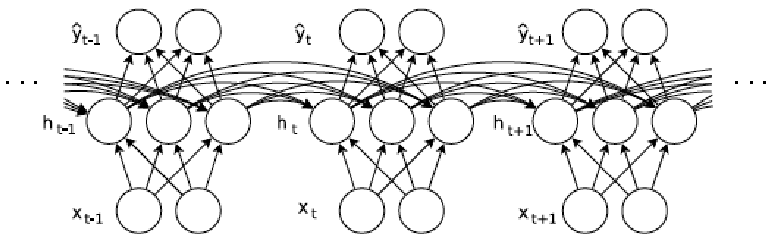

Figure 3: The architecture of a recurrent neural network. We can observe the three different layers and the connections between them: the input layer consists of the units x_{t-1}, x_t, x_{t+1}, the hidden layer with the units h_{t-1}, h_t, h_{t+1} and the output layer with the units y_{t-1}, y_t, y_{t+1}.

The equations above remind us of the feedforward NNs, with the difference of having to compute weights between hidden nodes. In these equations, W_{hx} is the input-to-hidden weight matrix, W_{hh} is the hidden-to-hidden (recurrent) weight matrix, W_{yh} is the hidden-to-output weight matrix and the vectors b_h, b_y are the biases.

Despite the similarities with feedforward NNs, the unit weights of RNNs cannot be easily trained using the backpropagation algorithm. The relationship between the parameters and the dynamics of the RNN is highly unstable which makes gradient descent ineffective. This problem was demonstrated by Hochreiter in 1991 [23] and Bengio et al. in 1994 [8] who proved that gradient decays exponentially as it is backpropagated through time, and concluded that RNNs cannot learn long-range temporal dependencies when gradient descent is used for training. The theoretical results of the above works and the popularity of gradient descent for training neural networks at the time led to the near abandonment of RNN research.

One approach that tried to deal with the inability of gradient descent to learn long-range temporal structure in a standard RNN was by Hochreiter and Schmidhuber in 1997 [24] who modified the model to include 'memory' units that are designed to store information over long time periods. Their approach is called 'Long-Short Term Memory' and has been successfully applied to complex real-world sequence modeling tasks. This approach makes it possible to handle datasets requiring long-term memorization but a more recent method by Martens and Sutskever in 2011 [33] which uses a standard RNN trained with a Hessian-Free (HF) optimizer, has even better performance.

The following subsection introduces the problem of language modeling, before the main section 3 brings the two topics together and reports on work that successfully applied Neural Networks in order to gain better generalizations in Language Models.

2.2 Language Models

As was mentioned above, in many NLP tasks, such as Machine Translation, Speech Recognition/Synthesis, Text Summarization and Natural Language Generation, a so-called Language Model (LM) is used to ensure that the output of these systems does not only convey meaningful words and the correct lexical choices but also that these words are in the right order to make the output sound fluent in the language needed. A language model captures statistical characteristics of the distribution of sequences of words in natural language and therefore allows to make predictions on the next word given its predecessors. A language model would give higher probability, for example, to the more natural word choice in context, in a word sequence like the following:

$$p_{LM}(\text{I am going home}) > p_{LM}(\text{I am going house}) \tag{12}$$

2.2.1 N-gram Language Models

The most common method to LMs is the so-called n-gram language modeling [26, 27], with an n of arbitrary order, though most often not more than tri- to five-grams are used due to computability, disk/memory space and sparsity reasons (see below). In n-gram language modeling, the process of predicting a word sequence is broken up into predicting one word at a time. The language model probability $p(w_1, w_2, ..., w_n)$ is a product of word probabilities based on a history of preceding words, whereby the history is limited to m words:

$$p(w_n|w_1, w_2, ..., w_{n-1}) \simeq p(w_n|w_{n-m}, ..., w_{n-2}, w_{n-1}) \tag{13}$$

This is also called a *Markov chain*, where the number of previous states (here: words), is the *order* of the model. The basic idea for n-gram LMs is that we can predict the probability of w_{n+1} with its preceding context by dividing the number of occurrences of w_n, w_{n+1} by the number of occurrences of w_n, which then would be called a *bigram*. If we would only base on the relative frequency of w_{n+1}, this would be a *unigram* estimator. The estimation of a trigram word prediction probability (most often used for LMs in practical NLP applications) is therefore straightforward, assuming maximum likelihood estimation:

$$p(w_3|w_1, w_2) = \frac{count(w_1, w_2, w_3)}{\sum_w count(w_1, w_2, w)} \tag{14}$$

2.2.2 Evaluation of Language Models

A language model's quality is usually measured with the so-called *perplexity* [3, 11], an information-theoretic measure and transformation of cross-entropy: $PP = 2^{H(P_{LM})}$. This means that less probable words get higher perplexity values. A good LM does not waste probability mass on 'impossible' word sequences, so when comparing LMs, the ones with lower perplexity are better modeling the given language (see [3], for a detailed discussion of the metric).

Apart from the perplexity measure, an LM's quality can also be assessed directly in the application in which it is used. For automatic speech recognition, for example, the quality measure is the so-called word error rate. Let's say one would have several LMs available, for instance of different size or a tri-gram and a five-gram LM. If these are then applied in a speech recognition system, the better LM would be the one that reduces the word error rate more. Similarly in a machine translation system, a better LM leads to more accurate translations. Translation quality is measured with the so-called BLEU score [41], and a better LM leads to an improved (here higher) BLEU score.

2.2.3 Smoothing Techniques

The above-mentioned n-gram language models have a couple of drawbacks. If we would have an unseen n-gram (not in the training data), formula 14 would assign it a probability of 0, which is too restrictive in practice. Also, when an n-gram appears in the training data, how often can one expect to see it again in a corpus of equal size? For these reasons, a variety of smoothing techniques to adjust the empirical counts to *expected* ones have been developed from very early on in the history of LMs.

The simplest smoothing technique is 'add-one smoothing', which prevents having zero-probabilities by just modifying the counting method for the maximum likelihood estimation of the probability p to: $p = \frac{c+1}{n+v}$ where c is the count, n the total number of n-grams and v the total number of possible n-grams. It is also possible to do 'add-α smoothing' (α being < 1) as using 'add-one' gives too much strength in non-observed counts. α is determined experimentally, by checking for the reduction of the perplexity measure.

Another technique is 'deleted estimation', that has proven to perform close to actual (manual) test counts. The training corpus is split (50%+50%, 90%+10% or even 99%+1%) in order to estimate the expected counts, i.e. how many times an n-gram will occur in held-out data. To not loose the held-out data for training, the method alternates between the training and held-out corpora and finally averages the count-of-counts (N) and the counts of the held-out data (T) in one direction (N_r^a and T_r^a) with the ones collected from the other direction (N_r^b and T_r^b):

$$r_{del} = \frac{T_r^a + T_r^b}{N_r^a + N_r^b} \text{ where } r = count(w_1, ..., w_n) \tag{15}$$

Other well-known smoothing techniques are the following: Good-turing frequency estimation [19]; Interpolation [11] and Back-off [27] and Witten-Bell smoothing [4].

2.2.4 Kneser-Ney Smoothing

The state of the art in language modeling and especially for non-prototypical NLP systems still are n-gram models along the lines described in the previous section. As smoothing technique, for most state-of-the-art models, the so-called 'Kneser-Ney smoothing' [28] is applied. All the above techniques still do not consider 'simple' cases like the following: a word like *york* might be fairly frequent in a corpus, though very likely, its preceding word will be *new* (for the American city *New York*) and certainly more often than occurrences of the English city *York*. The idea with Kneser-Ney smoothing therefore is to give less probability to the *unigram* 'york' than its raw count suggests, taking into account the *diversity of history* for a word. So, in the usual maximum likelihood estimation of a unigram model,

$$p_{ML}(w) = \frac{c(w)}{\sum_i c(w_i)} \tag{16}$$

with Kneser-Ney smoothing, we would replace the raw word count ($c(w)$) with the count of histories for a word:

$$p_{KN}(w) = \frac{N_1 + (\bullet w)}{\sum_{w_i} N_1 + (w_i w)} \tag{17}$$

To continue the *york* example, let's say it would occur 477 times in a corpus, which would be a quite high raw unigram count. With Kneser-Ney smoothing, however, we would only count its four different histories: *new york; in york; to york; of york*. As another example, the word *indicates* occurs with the same raw frequency, 477 counts, but has 172 different word histories, which would give this word, applying the Kneser-Ney technique, much higher probability than for *york*, which is correct and expected, see [29].

Despite all the smoothing techniques mentioned and the practical usability of n-gram LMs, the **curse of dimensionality** (as for many other learning algorithms) especially arises here as there is a huge number of different combinations of values of the input variables that must be discriminated from each other. For LMs, this is the huge number of possible sequences of words, e.g., with a sequence of 10 words taken from a vocabulary of 100,000 there are 10^{50} possible sequences.

There are other drawbacks with n-gram models: They have to rely on exact pattern, i.e. string or word sequence matching, and therefore are in no way linguistically informed. For example, one would wish from a good LM, that it recognizes a sequence like *the cat is walking in the bedroom* to be syntactically and semantically similar to *a dog was running in the room*, which cannot be provided by an n-gram model [6]. Another problem where this becomes especially obvious are so-called 'Out-of-Vocabulary' (OOV) word forms, i.e. word forms that have never been seen in training and cannot be handled by an n-gram LM, as zero probability would be assigned to such an occurrence.

These reasons lead to the idea to apply deep learning and Neural Networks to the problem of language modeling, in the hope to learn such syntactic and semantic features automatically and to overcome the curse of dimensionality by better generalizations that can be generated with NNs.

3 Neural Network Language Models

Artificial Neural Networks have been introduced to Language Modeling to overcome the three major drawbacks of current n-gram models: the curse of dimensionality (too many discrete random variables), the lack of good (more linguistic) generalizations and the problem of out-of-vocabulary (OOV) word forms. The basic idea behind NNs for the LM task is their ability to model *continuous* variables or **distributed representations**, which is needed if we would like to find better generalizations over the highly discrete word sequences possible in natural language [6]. N-gram models are a very local representation: they only incorporate the specific word forms of a sequence as features and therefore the number of features needed to capture the possible sequences grows exponentially with the sequence length as mentioned in the previous section.

However, as humans, we would probably tend to select merely morphosyntactical or semantical features to characterize a word (gender, number, animation, humanness, etc.). These can be dependent on a longer-range context of a word and are not mutually exclusive. They form a so-called distributed representation that can be learned by an NN as is described in the following subsection.

3.1 Distributed Representations

To continue the biological analogy behind NNs (see Section 2.1.5), a distributed representation resembles a cognitive representation that can efficiently represent an object by characterizing it with many features, of which each can be separately active or inactive. The brain learns objects as being similar to other ones, when many of these features are similarly active and inactive for a new object. In contrast, a local representation would only consist of one neuron (or very few) that is active at a time[1].

If one now assumes that the objects to be learned would be words, then each word is a continuous-valued vector representation that corresponds to a point in feature space. Functionally (grammatically and/or semantically) similar words are closer to each other in this space and a sequence of words can be transformed into a sequence of learned feature vectors. Also, such functionally similar words can be replaced by each other in a word sequence of the same context. This helps the NN to learn the similar feature vectors and to represent in a compact way the function that makes predictions for the word sequences used to train the model. Moreover, the distributed representation allows the NN to make predictions on test or held-out data, as it will see the same feature vectors for similar word sequences, as the NN tends to map nearby inputs to nearby outputs. This even helps to actually being able to predict OOV word forms, as these might consist of the same and/or similar compact feature vector representations.

The details of the modeling methods are given in the next subsection. Prior to NNs applied to LM, several neural network models have been proposed for learning symbolic data in general [5, 40], for modeling linguistic data (case distribution in sentences) [34] and character sequences [45].

3.2 Principle Methods

Neural Network Language Models have been introduced by Bengio et al., 2001 (revised in 2003 [6], the version we reference here) and independently by Xu and Rudnicky, 2000 [53].

Xu and Rudnicky (2000) ask the essential question right in the title of their work: *Can Artificial Neural Networks Learn Language Models?* The conclusion is 'yes' and an experiment is shown where an NN outperforms a baseline n-gram model in terms of perplexity. The method is however somewhat limited: a network without any hidden unit is used (the input units are directly and fully connected to the output units) and the input is a single word, which is why this approach is not likely to actually capture the desired bigram or n-gram statistics. We will nevertheless shortly summarize the approach as it will allow us to see how other authors addressed the remaining problems.

The NN designed in [53] consists of $|V|$ input units and $|V|$ output units with V being the vocabulary size. The i-th input unit is 1 if the current word is w_i . The i-th output unit represents the probability of w being the next word. The weights to train for are therefore $V \times (V + 1)$ (including the bias weight). To overcome the computational extensiveness of a network built this way, the authors only update the weights during backpropagation for those input values that are not zero (most of them actually are zero due to the sparsity of word sequences).

The goal finally is to reduce perplexity (and to get lower perplexity scores than a standard n-gram LM would achieve). The authors thus use the logarithm of perplexity as error function that is to be minimized during the NN training: $E = -\sum_t \log o_{t,w_t}$. Using this error function, the value of the i-th output will converge to $p(w_i|w_j)$ when the input to the network is the word wj which leads to a model being equivalent

[1]The idea of distributed representation goes back to the so-called 'connectionist' approach that was very important for the popularity of NNs. In the 1980's the goal was bringing together computer scientists, cognitive psychologists, physicists, neuroscientists etc. for joint research work ([21], [22] and [43]).

to a bigram language model without smoothing. Such a model tends to overfit the training data and performs poorly on unseen test data, which is why the authors use the 'early-stopping' technique that stops the training when the network performs best (at lowest perplexity) on a small set of held-out data. To guarantee that the output units (word probabilities) sum up to 1, a softmax activation function [10], o_i, is applied to each output unit y_i: $o_i = \frac{e^{y_i}}{\sum_j e^{y_i}}$ with NN input $y_i = w_{i,o} + \sum_j w_{ij} x_{ij}$ and x_{ij} as the j-th input to this unit.

On a very small corpus with a vocabulary size of 2500 words, the authors report a perplexity score of 11.16 outperforming a standard n-gram model (11.99) and, although by very few, a Kneser-Ney smoothed model (11.17). However, the network needs thousands of epochs of training (each epoch taking about a minute on a Pentium III, 500MHz), whereas the standard n-gram model can be obtained in less than half a minute with this vocabulary size.

Several other authors addressed this problem and have built more sophisticated NNs. The first work that was applicable to millions of words (which is what an LM normally consists of), was [6]. An overview of the network architecture is additionally given in Figure 4. There are three important components in this paper making use of the capability of an NN to learn distributed representations (see Section 3.1).

1. build a mapping C from each word i of the vocabulary V to a distributed, real-valued feature vector $C(i) \in \mathbb{R}^m$, with m being the number of features. C is a $|V| \times m$ matrix, whose row i is the feacture vector $C(i)$ for word i. Note that theoretically, the feature vectors could also be initialized using prior knowledge such as semantic features.

2. a probability function g over words, expressed with C: a function g maps the input sequence of feature vectors for words in context $(C(w_{t-n+1}), ..., C(w_{t-1}))$ to a conditional probability distribution of words in V for the next word w_t. Output of g then is a vector whose i-th element estimates the probability $\hat{P}(w_t = i|w_1^{t-1})$ or $P(w_i|\text{context})$. The geometric average $\frac{1}{\hat{P}(w_t|w_1^{t-1})}$, which is also the exponential of the average negative likelihood, is again the perplexity (see Section 2.2) and the quality measure one wants to minimize for obtaining a good LM.

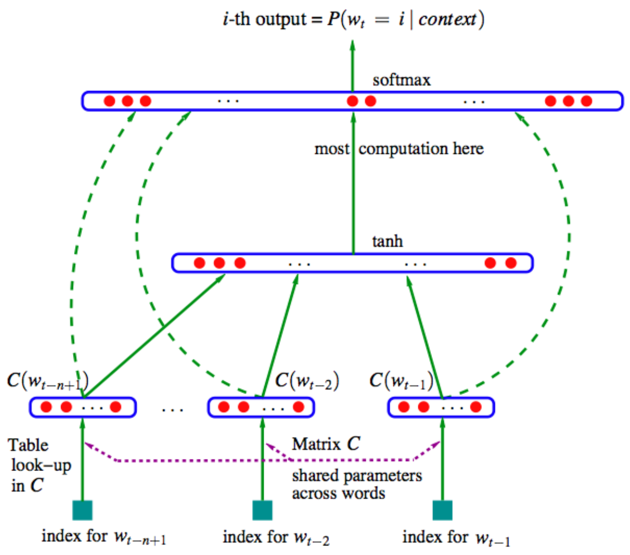

Figure 4: Architecture of a neural network applied to language modeling: $f(i, w_{t-1}, ..., w_{t-n+1}) = g(i, C(w_{t-1}), ..., C(w_{t-n+1}))$ where g is the NN and $C(i)$ is the i-th word feature vector (see equation 18).

3. finally learn simultaneously the *word feature vectors* and the parameters of that *probability function* with a composite function f, comprised of the two mappings C and g:

$$f(i, w_{t-1}, ..., w_{t-n+1}) = g(i, C(w_{t-1}), ..., C(w_{t-n+1})) \quad (18)$$

The function g can be implemented with a feed-forward or recurrent (see Section 2.1) NN with an overall parameter set of $\theta = (C, \omega)$. The training of the NN is searching for a θ (see below) that maximizes the log-likelihood that is penalized from the training corpus.

$$L = \frac{1}{T} \sum_t \log f(w_t, w_{t-1}, ..., w_{t-n+1}; \theta) + R(\theta) \tag{19}$$

where the authors applied R (a regularization term or weight decay penalty) to the weights of the neural network and to the C matrix. Such a weight decay penalty is a technique to prevent over-fitting the training data by avoiding irrelevant connections that would disturb the network's predictions. It penalizes large weights that do not contribute a correspondingly large reduction of the error.

In Figure 4 we observe that an NN with this architecture has two hidden layers (as opposed to [53] who used no hidden layer at all): the shared word features layer C and the ordinary NN hyperbolic tangent layer (see Section 2.1.5). Most computation therefore actually happens at the output layer, where, like in [53], a softmax function is applied on the output in order to guarantee positive probabilities summing up to 1:

$$\hat{P}(w_t | w_{t-1}, ..., w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}} \; ; \; y = b + Wx + U \tanh(d + Hx) \tag{20}$$

where y_i are unnormalized log-probabilities for each output word i, computed with the hyperbolic tangent $tanh$, x represents the parameters (concatenation of input word features from matrix C), W (a $|V| \times (n-1)m$ matrix for direct connections from words to the output layer, can be 0 if not desired) and b (output bias, with $|V|$ elements), d as hidden layer bias (with h elements), the hidden output weights U ($|V| \times h$ matrix) and the hidden layer weights H ($h \times (n-1)m$ matrix). Hence, the free parameters are represented by θ and the training of the NN is done with a stochastic gradient ascent method, by performing the following iterative update at the t-th word of the training corpus::

$$\theta = (b, d, W, U, H, C) \; ; \; \theta \leftarrow \theta + \epsilon \frac{\partial \log \hat{P}(w_t | w_{t-1}, ..., w_{t-n+1})}{\partial \theta} \tag{21}$$

where ϵ is the learning rate. A large part of the parameters needs not to be updated and visited after each example, as the word features $C(j)$ of all words j do actually not occur in the input window[2].

The authors then address issues of parallel implementation and parallel data and parameter processing (see Advanced Methods Section). With a system architecture of 40 CPUs, 6-12 weeks of training and 10-20 epochs, the best multi-layered, order 5 NN (with 100 hidden units) improves over a class-based tri-gram model by reducing the perplexity by 24%. Training was done on a 800k word corpus, validated on a 200k words and tested over a set of 181k words (although all the sets were normalized by removing rare words with a frequency of $\leqslant 3$, giving the vocabulary $|V|$ the actual size of about 17k words). In addition, the question of OOV words is solved by an NN architecture which is capable of guessing the initial feature vector for such a word, by taking a weighted convex combination of the feature vectors of other words that could have appeared in the same context with weights proportional to their conditional probability.

Several other open questions for the future are addressed, mostly concerning speed-up techniques, more compact probability representations (trees) and introducing a-priori knowledge (semantic information etc. to cluster the highly discrete word forms). In the following subsections we will see how other authors and more recent research continued based on these questions to solve the problem with computability and better generalization by introducing prior knowledge on word features.

3.3 Advanced Methods

Artificial Neural Networks (ANNs) have been used in several variations in the last decade by several authors instead n-gram based language models. In most of those papers the ANNs reach the performance of n-grams and even improve it further, in terms of the log-likelihood (language model's perplexity) or the classification accuracy of an application task (Speech Recognition, Statistical Machine Translation, Part-of-Speech Tagging

[2]In this model, the number of parameters only scales linearly with $|V|$, the vocabulary size and the order n. A sub-linear scaling factor could be achieved by using a time-delay or recurrent NN (see Section 2.1).

(POS), Named Entity Recognition, etc). The basic drawback of the NNs is the computational cost during training and recognition so that, in most cases, it is difficult to scale over large datasets. Inevitably, in most of the literature, the methods are effective and efficient in small datasets, where the training cost is cheaper and the improvements over traditional n-gram language models are significant. However, there exists considerable work on speeding up the training procedure and recognition (probability prediction) with several techniques. In addition, it has been noted that neural network language models and n-gram based language models make errors in different places, hence the combination of those two (averaged probabilistic predictions) often yields even better predictions than the two models separately and scales more easily over large datasets.

The neural network variants applied to language modeling that we met in the literature can be divided into several categories most of them emphasizing on the speed in training and recognition time of the neural network. In the sections below we describe the grouping of the methods we derived based on the literature as follows: Large-scale, Hierarchical, Hybrid and Other methods. We also dedicate one sub section on the methods that firstly appeared to apply neural networks to language modeling. We will provide below a complete overview of the techniques used along with the advantages and disadvantages for each of these groupings. Next, we demonstrate the most important state-of-the-art methods and finally we highlight some challenges for future research in the field of language modeling based on neural networks.

3.3.1 Large-scale Methods

As was mentioned several times in the previous sections, a considerable bottleneck using NNs for language modeling is computability and scalability over large training data sets and/or large vocabulary sizes. Several authors therefore addressed this question and developed different methods in order to overcome intolerably long training/testing times for the models.

After Bengio et al. (2001) [6] proposed the connectionist, word feature-vector based NN architecture (see Section 3.2), Schwenk and Gauvain (2002) [48] thought on how to reduce the biggest amount of computation for the probabilities with the softmax function in the ouptut layer of the NN-LM. The main idea in their work is to compute the probability in the output layer only for 'interesting words', which they call a *shortlist*, and not for words that appear only very rarely in the 64k vocabulary of their training corpus. The authors define those interesting words using two methods: dynamic shortlists or static shortlists i.e. using an NN that predicts the K most frequent words (e.g. 2000) with or without using context, respectively. To predict the probabilities for all the rest of the words (62k), a standard 3-gram back-off n-gram model is used. This can therefore be seen as a sort of model combination, but it is not a thorough hybridization as it was tried later by other authors (see Section 3.3.3). With this method the authors reduce the perplexity of the joint NN-n-gram-LM by up to 8.9% compared to the 3-gram model only. The fact that this reduction is achieved by only using the NN for calculating 47% of the probabilities speeds up the training and makes decoding (e.g. for speech recognition) feasible at testing time. The authors also found that increasing the size of the shortlist would reduce the perplexity even more, but then there is the trade-off, that the more words are on the shortlist, the more the NN is used to compute the probabilities.

One of the above authors, Schwenk (2004) [46] tried again to use an NN directly at decoding time of a speech recognizer. He developed five different methods to overcome the intractably long probability computing times:

- Lattice re-scoring: at decoding, a standard 4-gram backed-off Kneser-Ney smoothed LM is used to build a so-called 'lattice' (graph with several arcs (hypotheses) for the next word in a sequence), to which an already trained NN is applied to re-score the probabilities of the n-gram model predictions on the hypotheses.

- Shortlists: Same as mentioned above except that here, there is a combination with lattice re-scoring at testing time.

- Regrouping: the lattices are sorted and combined if there is the same word context – for such a sorted lattice, then only one forward-pass through the NN is needed.

- Block mode: gather a bunch of examples and push them through the NN at once. This makes matrix/matrix operations faster.
- CPU optimization: machine-specific libraries are used to speed-up matrix and vector operations[3].

The combination of all these points, at decoding time of a speech recognizer reduced the word error rate (a common measure in speech recognition) consistently by 0.6% with respect to a carefully tuned backoff 4-gram LM. Moreover, lattice re-scoring leads to continuous speech decoding in less than 0.05 times real time.

In 2005, it was again Schwenk, working with Gauvain [47] to propose further methods to speed up the *training* of NNs on very large corpora. In principle, the same five above-mentioned methods were used, but in combination with a new algorithm based on a training data sub-sampling: instead of running several epochs over the whole (potentially very large) training data, the algorithm only samples a small, random sub-set (of about 10% of the total amount of data) over which one epoch is run, by updating the weights for each training instance. After such an epoch the performance is tested on some small, held-out development data and the procedure is repeated until convergence. This has several advantages: there is no limit on the amount of training data; after some epochs, it is likely that all the training examples have been seen at least once and changing the examples after each epoch adds noise to the training procedure, which increases the generalization performance. Using this method and a corpus of 92.5 words, the perplexity with such an NN is reduced by 3.5% compared to a baseline back-off 4-gram model and the word error rate is reduced by 0.49%[4].

3.3.2 Hierarchical Methods

In this section we present another two papers that were concerned about an NN-LM's training and testing speed. However, in a sense the methods used in them are also concerned about an even better generalization over the predicted words of the NN model at its output layer. The basic idea in these papers is to cluster similar words before computing their probability in order to only have to do one computation per word cluster at the output layer of the NN.

In Morin and Bengio, 2005 [39], the lexical resource WordNet [36] is used to do the clustering. WordNet provides an ontological graph (with IS-A, synonymy, antonymy etc. relations) over adjectives, verbs and nouns from which similarity scores (graph distances) can be calculated. An example to illustrate the approach is when one would take 10,000 values and 100 classes containing 100 words each, then only two normalizations each over 100 choices are necessary instead of 10,000 normalizations. The computation of probabilities in an NN is proportional to the number of choices (as shown above), so the clustering of words on the output layer reduces needed computations by a factor of 50. The clustering is performed via a binarization of the WordNet relational trees and the K-Means algorithm that uses the TF-IDF word cooccurrence frequency measure [44].

The authors report speed-up factors of such a hierarchical word-clustering based NN-LM of 258 at training and 193 at testing time. However, perplexity is lower for the original NN-LM by [6] (see Section 3.2), i.e. 195.3 (original NN-LM) vs. 220.7 (new hierarchical, clustering NN-LM), which is nevertheless far better than a baseline tri-gram LM at 268.7 perplexity score (over a held-out test set). In addition, information from WordNet could even be used in a more intelligent way as WordNet provides additional word sense disambiguation information that could help better linguistic generalizations over the possible words instead of the binary IS-A trees in this model.

The fact that the clustering based NN-LM performed lower than the original NN-LM in spite of using the WordNet expert knowledge is then addressed by a later work, Mnih and Hinton in 2008 [37]. This work is based on a simple feature-based algorithm that *learns* the word clustering *directly* from the NN-LM training data, *without* using external expert knowledge in order to have trees that are well-supported by the data, generalize well and result in a model that is fast to train and test.

The so-called hierarchical log-bilinear model (HLBL) is a binary tree with words at its leaves. Each word can therefore be encoded as a path from root to leave of a binary decision tree. In this model, the

[3]This is also used during NN training, reducing the time needed by a factor of 30 (leading from 47 to 1.5 hours per epoch).

[4]This training still takes 9 hours 40 minutes per epoch. The authors also experimented with 600M words corpora, but the training time rose to 12 hours per epoch without considerably better results (further reduction of the perplexity by only 0.2%).

probability of the next word w is the probability of making the sequences of binary decisions specified by the word's encoding, given its context. Since the probability of making a decision at a node depends only on the predicted feature vector, determined by the context, and the feature vector for that node, the probability can be expressed by the probability of the next word as a product of probabilities of the binary decisions:

$$P(w_n = w|w_{1:n-1}) = \prod_i P(d_i|q_i, w_{1:n-1}) \tag{22}$$

where d_i is the i-th encoding for word w_i and q_i is the feature vector for the i-th node in the path to the corresponding word encoding. The above probability definition can be extended to multiple encodings per word and a summation over all encodings, which allows better prediction of words with multiple senses in multiple contexts. The best HLBL-LM reported in [37] reduces perplexity by 11.1% compared to a baseline Kneser-Ney smoothed 5-gram LM at only 32 minutes training time per epoch.

In the next sections we report on methods using hybrid approaches (combining NNs with standard n-gram models), which is a promising and practically applicable technique to reach fast and accurate performance. Further subsections will treat recurrent neural networks, either on the level of predicting next characters or predicting next word, whereby the latter provide state-of-the-art performance for NN-LMs.

3.3.3 Hybrid Methods (NN + n-gram)

In all the methods described so far, there is a trade-off between the computational cost and the quality of the output model, but still fast NN models can improve over the state-of-the-art n-gram models. On the other hand, state-of-the-art n-gram methods are very effective and efficient despite the mentioned drawbacks of the produced model (generality, curse of dimensionality, etc.). An intuitive idea is to use an efficient neural language model in combination with the state-of-the-art n-gram model and see if there is an even better performance in terms of perplexity of the combined model.

Bengio et al. in 2003 [6] experimentally proved that a mixture of the two models improves performance i.e. reduces perplexity of the produced mixture language model (24%) compared to the perplexity of the individual ones. The authors combined the probability predictions of the neural network with those of an interpolated trigram model in several settings: with a simple fixed weight of 0.5, a learned weight based on maximum likelihood on the validation set and a set of weights that are conditional on the frequency of the context (using the same procedure that combines trigram, bigram and unigram in the back-off trigram models). In addition, the improvement achieved by the mixture of models was constant as the number of hidden units increased, which together with the fact that a simple weighted combination was used, suggests that the neural language model and the interpolated trigram make errors in different places (e.g. low probability given an observed word for an n-gram model could have higher probability in the neural language model).

Xu et al. in the same year [52] also performed experiments by combining neural networks with trigram models and achieved (8%) relative improvement in perplexity of the best n-gram result with Kneser-Ney smoothing. These two papers established the effective usage of mixture models in language modeling, with Bengio et al. demonstrating the bigger improvement over state-of-the-art n-gram models. It seems that both, the quality of the neural language model and the n-gram model play an important role on the range of reduced perplexity (different improvement in both papers). The mixture model could also be used as a quality criterion for a neural language model, for example if its combination with the state-of-the-art n-gram model does not produce a better model in terms of perplexity, then we could argue that the neural language model has learned all the strong points (i.e. high word probabilities) of the n-gram model and is superior to it, i.e. in this case the combination of those two models would be unnecessary.

A more recent study by Le et al. in 2011 [32], explored the similarly a mixture model technique called SOUL: Structure Output Layer Neural Network Language Model (NNLM) which combines neural networks with n-gram language models in a unified approach. The proposed SOUL model is based on word clustering to structure the output vocabulary and also uses the design of shortlists that are commonly used for efficiency (see previous Section). The best performance was observed by using the SOUL model with 6-grams and a shortlist of 12k most frequent words, which achieved a 23% improvement over the perplexity of a 4-gram baseline language model.

3.3.4 Related Methods

The majority of the literature deals with the neural networks applied to language modeling, but there are some variations related to language modeling that are worth to be mentioned.

Emami and Jelinek in 2004 [18] proposed a neural Structured Language Model (SLM). SLM aims at making a prediction of the next word in a given word string by making syntactical analysis of the preceding words. More precisely, SLM assigns a joint probability $P(W, T)$ to every sentence W and every possible binary parse T of W. The T nodes are the words of W with Part-of-Speech (POS) tags and they are annotated with phrase headwords and non-terminal labels [13]. The word k-prefix of a sentence is defined as $W_k = w_0 + w_1 + ... + w_k$ i.e. the words from the beginning of the sentence up to the current position k and $W_k T_k$ is the word-parse k-prefix. The language model probability assignment for the word $k + 1$ in the input sentence is made using:

$$P_{SLM}(w_{k+1}|w_k) = \sum_{T_k \in S_k} P(w_{k+1}|W_k T_k) \cdot \rho(W_k, T_k) \ ; \ \rho(W_k, T_k) = P(W_k, T_k) / \sum_{T_k \in S_k} P(W_k, T_k) \tag{23}$$

where S_k is the set of all parses present at stage k, $\rho(W_k, T_k)$ is the weight of the partial parse (W_k, T_k) at stage k. The authors use a fully connected neural network with one hidden layer which is trained with enriched contexts accordingly in order to model the SLM (the probability assignment shown above P_{SLM}). The training is done on parses built either by an external source or the baseline SLM. The other components of the baseline SLM are used as is [13], parametrized by n-gram interpolated models. The proposed approach could be considered as a mixture model (neural networks with SLM) and performed significantly better than the baseline SLM on two different datasets.

A more recent approach by Sutskever et al. in 2011 [49] uses Recurrent Neural Networks (RNNs) applied to character-level language modeling task. RNNs, albeit being powerful and able to capture non-linear dynamics via their high-dimensional hidden state, are not widely used mostly due to the difficulty of training them properly. The authors address this problem by training a variation of RNN with the Hessian-Free Optimizer (HF) [5] [33] called Multiplicative RNN. The proposed approach uses multiplicative (or 'gated') connections which allow the current input character to determine the transition matrix from one hidden state vector to the next. The intuition behind this variant is that a more powerful way for a character to affect the hidden state dynamics than standard RNNs would be to determine the entire hidden-to-hidden matrix in addition providing an additive bias. This is simply done by modifying equations of a standard RNN (see Section 2.1) so that each hidden-to-hidden weight matrix W_{hh} is a learned function of the current input x_t i.e. $W_{hh}^{(x_t)}$ is defined as a tensor:

$$W_{hh}^{(x_t)} = \sum_{m=1}^{M} x_t^{(m)} W_{hh}^{(m)} \tag{24}$$

where $x_t^{(m)}$ is the m-th coordinate of x_t. The input x_t is the 1-of-M encoding of a character and is associated with a weight matrix and $W_{hh}^{(x_t)}$ is the matrix assigned to the character x_t. This first variant the authors proposed is called Tensor RNN but it has a major drawback: the 3-way tensors are not very practical because of their size (the storage becomes prohibitive for large dimensionality of x_t). The above problem was remediated by factoring the tensor $W_{hh}^{(x_t)}$ [50] and introducing three matrices W_{fx}, W_{hf} and W_{fh} and modifying the above equation as follows:

$$W_{hh}^{(x_t)} = W_{hf} \cdot diag(W_{fx} x_t) \cdot W_{fh} \tag{25}$$

In the above equation if the dimensionality of $W_{fx} x_t$ is sufficiently large, then the factorization is as expressive as the original tensor. Finally, the Multiplicative RNN (MRNN) (Figure 5) follows the standard equations of a standard RNN by modifying the h_t equation of a hidden unit with the following:

$$f_t = diag(W_{fx} x_t) \cdot W_{fh} h_{t-1}, \ ; h_t = tanh(W_{hf} f_t) \cdot +W_{hx} x_t \tag{26}$$

[5] HF is a form of non-diagonal, second-order optimization and provides a principled solution to the vanishing gradients problem (unstable relationship between the parameters and the dynamics of the hidden states) in RNNs.

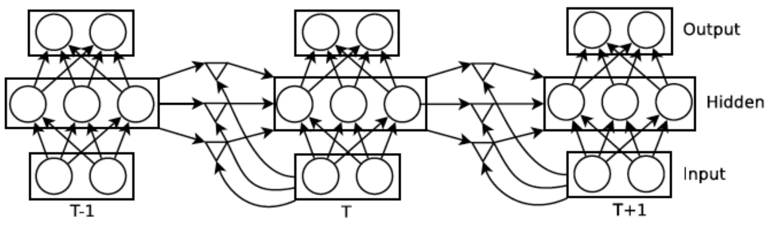

Figure 5: The architecture of a multiplicative recurrent neural network. Each of the triangle symbols represents a factor that applies a learned linear filter at each of its two input vertices. The product of those two vertices is sent by weighted connections to all the units connected to the third vertex of the triangle.

The advantage over standard RNNs was verified by experiments that showed that MRNN even when using less hidden units (350 compared to 500) achieved better performance terms of perplexity. The expressive MRNN can be used as a generative model by sampling the conditional distribution to get the next character in a generating string and provide it as the next input to the RNN. This means that the RNN is a directed non-Markov model and it resembles a sequence memorizer [51]. The authors further evaluated MRNNs in this task and showed state-of-the-art performance and even better results compared to other methods. The generative model was able to generate natural text which captures high level semantics of sentences and even generate new words that were not present in the training dataset. The training time was 5 days on 8 high-end graphic processing units, so despite the good results in application benchmarks, the training performance still remains an important issue.

3.4 Methods Using Recurrent or Deep Neural Networks

Despite the improvements brought by faster training/testing times, strong parallel implementations and the clustering of words on the output layer, a major drawback of the above methods still is that one needs to specify how much context the NN-LMs actually see during training, and, as with n-gram models, this parameter is usually set to 5-10 words only. At the cost of much more training time, improvements for complexity are rather marginal with NN-LMs. These two reasons led to current state-of-the-art methods where recurrent NNs are used for language modeling in order to include longer-range context and considerable reductions of perplexity.

In 2010, Mikolov et al. [35] claim to have reached the highest perplexity reductions (up to 50%) ever reported by using a combination of 3 recurrent NNs compared to standard Kneser-Ney smoothed 5-gram LMs. Moreover, in their work, the authors impressively show that larger perplexity reductions can be achieved with recurrent NNs using much less training data as for the 5-gram baseline model (5.4M words vs. 1.3G(!) words).

The recurrent NN-LMs are implemented as so-called 'simple recurrent neural networks' or 'Elman networks' [17]. Such an easy implementable recurrent NN has an input layer x, a hidden layer s (also called context layer or state) and an output layer y. Input to the network in time t is $x(t)$, a concatenated vector of vector w for the current word and the output vector of the context layer s at time $t-1$. The output is denoted as $y(t)$, and $s(t)$ is the state of the network (hidden layer). The output layers are then computed as follows:

$$x(t) = w(t) + s(t-1) \; ; \; s_j(t) = f(\sum_i x_i(t) u_{ji}) \; ; \; y_k(t) = g(\sum_j s_j(t) v_{kj}) \tag{27}$$

where $f(z)$ is the sigmoid activation function and $g(z)$ is the softmax function to guarantee a valid possible word distribution on the output layer:

$$f(z) = \frac{1}{1+e^{-z}} \text{ and } g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}} \tag{28}$$

For initialization, $s(0)$ is set to a vector of small values, such as 0.1. In the next time steps, $s(t+1)$ is a copy of $s(t)$ and $x(t)$ represents the word at time t encoded using 1-of-N coding and the previous context layer, whereby the size of vector x is equal to size of vocabulary V plus the size of the context layer. The size of the context (hidden) layer s was usually 30–500 hidden units (i.e. for large amounts of data, larger hidden layers are needed). The recurrent NN are trained in several epochs, in which all data from the training corpus is sequentially presented. The weights are initialized to small values (random Gaussian noise with zero mean and 0.1 variance). To train the network, a standard backpropagation algorithm with stochastic gradient descent is used with a starting learning rate $\alpha = 0.1$. Convergence is usually achieved after 10-20 epochs. At each training step, an error vector is computed according to the perplexity criterion and weights are updated with the standard backpropagation algorithm: $\text{error}(t) = \text{desired}(t) - y(t)$, where 'desired' is a vector using 1-of-N coding representing the word that should have been predicted in a particular context and $y(t)$ is the actual output from the network. Though the simple architecture leading to the big perplexity reductions are impressive, training such recurrent NN still takes several week as the authors state themselves.

Le et al., 2011 [32] continue to explore the word clustering approach as presented in Section 3.3.2 above. Their word clustering procedure before training the NN-LM is based on a straight-forward relationship between the two word spaces, the context and prediction space (see also [31]) and is much simpler as for example [37]. The clustering consists of 3 steps: In step 1, the authors train a standard NN-LM model with a 8k word shortlist as output and 3 epochs. Step 2 then reduces the dimension of the context space to 10 using PCA. Finally, in step 3, recursive K-means word clustering based on the distributed representation induced by the context space is performed (except for the words of the shortlist). The authors report a perplexity reduction of 34 with an 8k shortlist, 6-gram NN-LM compared to baseline 4-gram LM and also demonstrate the applicability of such a model to a Mandarin speech recognition task.

Recently, another work from Collobert et al. in ([15, 16]) have impressively shown the use of semi-supervised NNs to perform at the state-of-the-art of various NLP tasks (POS tagging, Chunking, Named Entity Recognition and Semantic Role Labeling). These NLP tasks are normally solved by using a cascade of classifiers, one for each task, that learns features specific to each disambiguation. Moreover, the single classifiers are often trained on costly engineered features and hand-labeled data.

It is therefore the authors' hypothesis that the features could be learned automatically and it is likely that features for one task would inform the other classifiers. This results in a joint training procedure for all tasks together (this is also referred to as 'Multi-task Learning'). The authors use an NN architecture for that, which is, in principle, based on [6, 14] (see also Section 3.2), but by extending it to a so-called *deep* neural network, or *convolutional* neural network.

The architecture of such an NN is shown in Figure 6 and relies on learned feature look-up tables that are shared between the NLP tasks. Each word $i \in D$ (finite dictionary) is embedded into a d-dimensional space using a lookup table $LT_W(\bullet)$: $LT_W(i) = W_i$, where $W \in \mathbb{R}^{d \times |D|}$ is a matrix of parameters that will be learned and $W_i \in \mathbb{R}^d$ is the i-th column of W and d is the word vector size (wsz) that is selected by the user. The parameters W of this layer are actually automatically trained during the learning process using backpropagation.

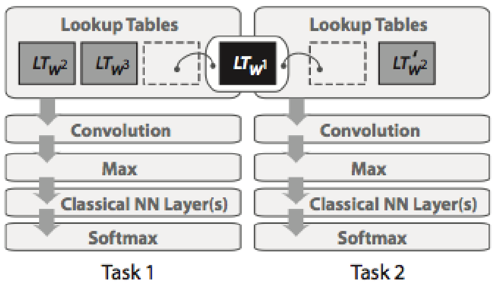

Figure 6: Architecture of a convolutional neural network for multi-task learning. The black lookup-table is shared, while the other tables and layers are task specific (following the same principle) (cf. [15]).

Instead of training the NN-LM traditionally to predict next word probabilities, Collobert et al. [15] use their architecture from Figure 6 to train an NN-LM that discriminates two classes: if the word in the middle of the input window is related to its context or not. This technique speeds-up the training process considerably (to a week on a single computer) and is much simpler than the sophisticated methods presented in Section 3.3.1. The language model was separately trained on a huge dataset (the whole English Wikipedia). It clusters semantically similar words which are then used as valuable features for the other shared NLP tasks (i.e. the word look-up-table generated with the language model is then applied as an initializer of the look-up-table for Multi-Task Learning experiments). For example, with the convolutional NN architecture, a joint Semantic-Role-Labeling+POS tagging+Chunking+Named Entity Recognition task performs at a word error rate of 16.27 (with a word vector size of 100). When adding the NN language model, the word error rate can be reduced to 14.50. The authors extended the LM architecture and size in a more recent work [16].

Even in this year's International Machine Learning Conference (ICML), there is a paper, Mnih and Teh (to appear) [38], on a further method to account for the still long training times of state-of-the-art NN-LMs. The long training times are due to, as was mentioned above, to having to consider all words in the vocabulary when computing the log-likelihood gradients. The authors propose a fast training algorithm for a neural probabilistic NN (see Section 3.2), based on a noise-contrastive estimation. This is a new technique for estimating unnormalized continuous distributions (which words basically are, if there is no lowercasing, stemming, lemmatization etc.). The algorithm requires far fewer noise samples to perform well (compared to the importance sampling ([7]), similar to NN + n-gram combination (see Section 3.3.3). The authors show the scalability by training several neural language models on a 47M-word corpus with a 80K-word vocabulary. The NNs provide state-of-the-art results in a shared sentence completion task.

4 Remaining Challenges and Conclusion

By gathering and reading through the literature that examined the use of NNs to language modeling, we felt that NN-LMs indeed are a very suitable and timely contribution to the difficulty of the task, namely to learn generalization over a highly discrete space of variables: natural language word sequences. The huge reductions in perplexity compared over n-gram models and the ability of an NN-LM to cluster and embed functionally and semantically related words has proven their usefulness. One might object that some of these NN-LMs still take weeks to train and are hardly applicable at testing times for real-time applications. But here as well, recent research on word clustering methods, shortlists, combinations with NN and n-gram models at the NN output layers, techniques with convolutional NNs as well as the increasing amount of available parallel computing power is promising for the usage of NN-LMs in any NLP real-time application in future.

However, we also felt that it is very difficult to actually and by fair means compare the different models built in recent research. A problem on the one hand are the datasets used to train the models on. Even if the same huge corpora, such as Wikipedia or the Wall Street Journal Corpus, are used, the authors apply different pre-processing steps to the data, i.e. they might normalize word forms or not, they discard rare words of different relative frequency etc. In the architecture of the NNs the amount of hidden layers varies highly among the different models and is not always reported. Moreover, some authors report training time in terms of actual hours/days needed, while others only report the number of epochs it took the NN to converge (by not stating how long an 'epoch' actually is in time).

These are several reasons why comparison between different NN-LMs and the applied techniques is difficult. Also, for a modeling task that is so computationally expensive, a unified protocol to report on computational architecture, actual training time and number of epochs would be very helpful. On top of that, depending on the languages modeled, common research should try to make use of existing (likely concatenated) text corpora to have a sort of fair comparison of different NNs to baseline models. Of course, the baselines used so far, n-gram LMs are perfectly fine and should still be used for reporting results, but a baseline NN, say, trained on the corpora X-Z, by pre-processing with X, using architecture X with X hidden layers that took X epochs (and X hours) of training would help future research to further improve and accelerate the already promising performances of Neural Network Language Models.

References

[1] A. Aizerman, E. M. Braverman, and L. I. Rozoner. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] J. K. Anlauf and M. Biehl. The AdaTron: an adaptive perceptron algorithm. *Europhys.˜Letters*, 10, 1989.

[3] L. Bahl, J. Baker, E. Jelinek, and R. Mercer. Perplexity – a measure of the difficulty of speech recognition tasks. *Program of the 94th Meeting of the Acoustical Society of America*, 62(1):63, 1977.

[4] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[5] Y. Bengio and S. Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems (NIPS 12)*, pages 400–406, Denver, CO, 2000.

[6] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3(1):1137–1155, 2003.

[7] Y. Bengio and J.-S. Sénécal. Adaptive importance sampling to accelerate training of a neural probabilistic language model. *IEEE Transactions on Neural Networks*, 19(4):713–722, 2008.

[8] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157 –166, mar 1994.

[9] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20, pages 161–168. 2008.

[10] J. S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In S. F. Fogleman and J. H. Herault, editors, *Neurocomputing: Algorithms, Architectures and Applications*, pages 227–236. Springer-Verlag, Berlin, DE, 1990.

[11] P. F. Brown, S. A. Della Pietra, V. J. Della Pietra, J. C. Lai, and R. L. Mercer. An estimate of an upper bound for the entropy of english. *Computational Linguistics*, 18(1):31–40, 1992.

[12] A. E. Bryson, Y.-C. Ho, and G. M. Siouris. Applied optimal control: Optimization, estimation, and control. *Systems, Man and Cybernetics, IEEE Transactions on*, 9(6):366–367, June 1979.

[13] C. Chelba and F. Jelinek. Structured language modeling. *Computer Speech & Language, Elsevier*, 14(4):283–332, 2000.

[14] R. Collobert and J. Weston. Fast semantic extraction using a novel neural network architecture. In *Proceedings of the 45th Annual Meeting of the ACL*, pages 560–567, Prague, CZ, 2007.

[15] R. Collobert and J. Weston. A unified architecture for natural language processing: deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 160–167, New York, NY, USA, 2008. ACM.

[16] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.

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

[18] A. Emami and F. Jelinek. Exact training of a neural syntactic language model. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 1, pages I – 245–8 vol.1, may 2004.

[19] W. A. Gale and G. Sampson. Good-turing frequency estimation without tears. *Journal of Quantitative Linguistics*, 2(3):217–237, 1995.

[20] S. I. Gallant. Perceptron-based learning algorithms. *IEEE Transactions on Neural Networks*, 1(2):179–191, 1990.

[21] G. E. Hinton. Learning distributed representations of concepts. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, pages 1–12, Amherst, MA, 1986.

[22] G. E. Hinton. Connectionist learning procedures. *Artificial Intelligence Journal*, 40(1-3):185–234, 1989.

[23] S. Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. diploma thesis. *Institute fur Informatik, Technische Universitat, Munchen*, 1991.

[24] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, Nov. 1997.

[25] J. J. Hopfleld. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of The National Academy of Sciences*, 1982.

[26] F. Jelinek and R. Mercer. Interpolated Estimation of Markov Source Parameters from Sparse Data. In E. Gelsema and L. Kanal, editors, *Pattern Recognition in Practice*, pages 381–397. North-Holland, 1980.

[27] S. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35(3):400–401, 1987.

[28] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, volume 1*, pages 181–184, Detroit, MI, 1995.

[29] P. Koehn. *Statistical Machine Translation.* Cambridge University Press, Cambridge UK, 2010.

[30] W. Krauth and Mézard. Learning algorithms with optimal stability in neural networks. *J. Phys. A*, 20:L745–L752, 1987.

[31] H. S. Le, A. Allauzen, G. Wisniewski, and F. Yvon. Training continuous space language models: Some practical issues. In *Proceedings of EMNLP*, pages 778–788, Cambridge, MA, 2010.

[32] H.-S. Le, I. Oparin, A. Allauzen, J.-L. Gauvain, and F. Yvon. Structured output layer neural network language model. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 5524 –5527, may 2011.

[33] J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 1033–1040, 2011.

[34] R. Miikkulainen and M. G. Dyer. Natural language processing with modular pdp networks and distributed lexicon. *Cognitive Science*, 15(3):343–399, 1991.

[35] T. Mikolov, M. Karafiat, L. Burget, J. H. Cernocky, and S. Khudanpur. Recurrent neural network based language model. In *Proceedings of Interspeech*, pages 1045–1048, Makuhari, Japan, 2010.

[36] G. A. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11):39–41, 1995.

[37] A. Mnih and G. E. Hinton. A scalable hierarchical distributed language model. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, volume 21, pages 1081–1088. MIT Press, Cambridge, MA, 2008.

[38] A. Mnih and Y. W. Teh. A fast and simple algorithm for training neural probabilistic language models. In *Proceedings of the International Conference on Machine Learning (ICML)*, Edinburgh, UK, 2012.

[39] F. Morin and Y. Bengio. Hierarchical probabilistic neural network language model. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, pages 246–252, Barbados, 2005.

[40] A. Paccanaro and G. Hinton. Extracting distributed representations of concepts and relations from positive and negative propositions. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks (IJCNN)*, Como, IT, 2000.

[41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. BLEU: A Method for Automatic Evaluation of Machine Translation. In *Proceedings of 40th Annual meeting of the Association for Computational Linguistics (ACL)*, pages 311–318, Philadelphia, PA, 2002.

[42] F. Rosenblatt. The perceptron: A perceiving and recognizing automaton. Technical Report 85-460-1, Project PARA, Cornell Aeronautical Laboratory, Ithaca, New York, 1957.

[43] D. E. Rumelhart and J. L. McClelland. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition.* MIT Press, Cambridge, MA, 1986.

[44] G. Salton and C. Buckley. Term weighting ap- proaches in automatic text retrieval. *Information Processing and Management*, 24(5):513–523, 1988.

[45] J. Schmidhuber and S. Heil. Sequential neural text compression. *IEEE Transactions on Neural Networks*, 7(1):142–146, 1996.

[46] H. Schwenk and J.-L. Gauvain. Training neural network language models on very large corpora. In *Proceedings of the Joint Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 201–208, Barcelona, ES, 2004.

[47] H. Schwenk and J.-L. Gauvain. Training neural network language models on very large corpora. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, pages 201–208, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics.

[48] H. Schwenk and J. luc Gauvain. Connectionist language modeling for large vocabulary continuous speech recognition. In *In International Conference on Acoustics, Speech and Signal Processing*, pages 765–768, 2002.

[49] I. Sutskever, J. Martens, and G. Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML)*, Bellevue, WA, 2011.

[50] G. W. Taylor and G. E. Hinton. Factored conditional restricted boltzmann machines for modeling motion style. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1025–1032, New York, NY, USA, 2009. ACM.

[51] F. Wood, C. Archambeau, J. Gasthaus, L. James, and Y. W. Teh. A stochastic memoizer for sequence data. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1129–1136, New York, NY, USA, 2009. ACM.

[52] P. Xu, A. Emami, and F. Jelinek. Training connectionist models for the structured language model. In *Proceedings of the 2003 conference on Empirical methods in natural language processing*, EMNLP '03, pages 160–167, Stroudsburg, PA, USA, 2003. Association for Computational Linguistics.

[53] W. Xu and A. Rudnicky. Can artificial neural networks learn language models? In *Proceedings of the Sixth International Conference on Spoken Language Processing (ICSLP)*, pages 202–205, Beijing, CN, 2000.