

**Masters in Computer Speech, Text and Internet
Technology**

**Neural Network-based Language Model
for Conversational Telephone Speech
Recognition**

Graeme W. Blackwood

St. Catherine's College

21st of July, 2005

This dissertation is submitted for the degree of Master of Philosophy.

Preface

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

I hereby declare that my thesis does not exceed the limit of length prescribed in the Special Regulations of the M. Phil. examination for which I am a candidate. The length of my thesis is 14980 words.

Acknowledgements

I would like to thank Professor Phil Woodland for his help and guidance over the course of this project.

I would also like to thank David Mrva for providing a customized version of the `LPlex` tool capable of summing over words in a specified shortlist.

Abstract

This thesis presents a large scale neural network language model for telephone conversation transcriptions. By mapping n -gram contexts to a continuous vector space, the neural network is trained with softmax normalization to operate as a probability estimator. The smooth nature of the resulting distributions achieves consistently reduced perplexity for restricted subsets of the vocabulary.

Excessive training time is a major issue and optimized linear algebra libraries are used for an efficient implementation of feed forward and back propagation during training.

A word-class interpretation of the network inputs and outputs is demonstrated to obtain improved perplexity over the n -gram model when training data is limited.

Table of Contents

| | | |
|-------|--|----|
| 1 | Introduction..... | 7 |
| 1.1 | Neural Network Language Models..... | 7 |
| 1.2 | Aims and Objectives..... | 9 |
| 2 | Background and Theory..... | 10 |
| 2.1 | Architecture of the Neural Network Language Model..... | 10 |
| 2.1.1 | The Projection Layer..... | 11 |
| 2.1.2 | The Hidden Layer..... | 12 |
| 2.1.3 | The Output Layer..... | 13 |
| 2.2 | Cross Entropy Error Criterion..... | 14 |
| 2.3 | Perplexity Calculation..... | 14 |
| 2.4 | Neural Network Shortlist..... | 15 |
| 2.5 | Language Model Data..... | 15 |
| 2.6 | Neural Network Training Patterns..... | 16 |
| 3 | Gradient Descent and Network Convergence..... | 18 |
| 3.1 | Gradient Descent Introduction..... | 18 |
| 3.2 | Gradient Descent Optimizations..... | 18 |
| 3.2.1 | Variable Learning Rate..... | 18 |
| 3.2.2 | Momentum Term..... | 19 |
| 4 | Neural Network Language Model Implementation..... | 20 |
| 4.1 | Data Structures and Algorithms..... | 20 |
| 4.1.1 | Neural Network Layer..... | 20 |
| 4.1.2 | Neural Network Data-set..... | 21 |
| 4.1.3 | Forming the Projection..... | 22 |
| 4.2 | Optimized Training with BLAS..... | 22 |
| 4.2.1 | Neural Network Architecture with BLAS..... | 22 |
| 4.2.2 | Feed-forward using BLAS..... | 23 |
| 4.2.3 | Back Propagation using BLAS..... | 25 |
| 4.2.4 | Updating the Network Weights..... | 27 |
| 4.3 | Neural Network Training Procedure..... | 28 |
| 4.4 | Neural Network Testing Procedure..... | 29 |
| 5 | Results and Analysis..... | 30 |
| 5.1 | N-gram Perplexities..... | 30 |
| 5.2 | Model Parameters..... | 30 |
| 5.3 | Baseline Experiments..... | 32 |
| 5.4 | Neural Network Properties..... | 34 |
| 5.4.1 | Network Training Times..... | 34 |
| 5.4.2 | Initialization Experiments..... | 34 |
| 5.4.3 | Over-fitting Experiments..... | 36 |
| 5.4.4 | N-gram History Length Experiments..... | 36 |
| 5.4.5 | Topology Experiments..... | 38 |
| 5.5 | Full Model Experiments – The Shortlist..... | 40 |
| 6 | Class-based Neural Network Language Models..... | 42 |
| 6.1 | Background and Theory..... | 42 |
| 6.2 | Class-based N-gram Baseline Models..... | 43 |
| 6.3 | Class-based Neural Network Models..... | 44 |

| | | |
|-------|---|----|
| 6.4 | Class-based Results and Analysis..... | 46 |
| 6.4.1 | N-gram Results | 46 |
| 6.4.2 | Neural Network Results..... | 46 |
| 7 | Conclusions and Future Work | 48 |
| 7.1 | Conclusions..... | 48 |
| 7.2 | Future Work..... | 48 |
| | Appendix A – Program Operation | 50 |
| | Appendix B – Program Configuration..... | 51 |
| | Appendix C – Program Output..... | 53 |
| | Appendix D – Project Distribution | 55 |
| | Bibliography and References..... | 56 |

List of Tables

| | | |
|----------|--|----|
| Table 1 | – Cellular 1 and Switchboard I corpus statistics..... | 16 |
| Table 2 | – Cellular 1 and Switchboard I OOV statistics..... | 16 |
| Table 3 | – Padded training patterns for sentence initial words..... | 17 |
| Table 4 | – Training and testing n -gram perplexities for Cellular 1 and Switchboard I ... | 30 |
| Table 5 | – Model parameters comparison for n -gram and neural network..... | 31 |
| Table 6 | – Training and testing perplexities for the Switchboard I corpus..... | 33 |
| Table 7 | – Epoch training times in seconds for Cellular 1 and Switchboard I..... | 34 |
| Table 8 | – Shortlist training and testing perplexities for Cellular 1..... | 40 |
| Table 9 | – Shortlist training and testing perplexities for Switchboard I..... | 41 |
| Table 10 | – Class memberships assigned by the HLM tool Cluster | 43 |
| Table 11 | – Testing perplexity for class-based n -gram models using Cellular 1..... | 46 |
| Table 12 | – Testing perplexity for class-based neural network models using Cellular 1 .. | 47 |
| Table 13 | – Testing perplexity for class-based models using Switchboard I..... | 47 |
| Table 14 | – Neural network tool command-line options | 50 |
| Table 15 | – Neural network language model configuration file settings..... | 52 |

List of Figures

| | |
|---|----|
| Figure 1 – Connectionist language model architecture..... | 10 |
| Figure 2 – Formatting the projection vector from an n -gram context | 12 |
| Figure 3 – Matrix-vector connectionist language model architecture | 23 |
| Figure 4 – Neural network training algorithm | 28 |
| Figure 5 – Neural network testing algorithm..... | 29 |
| Figure 6 – Training perplexities for Cellular 1 and Switchboard I..... | 32 |
| Figure 7 – Testing perplexities for Cellular 1 and Switchboard I..... | 33 |
| Figure 8 – Initialization and training data convergence for Cellular 1 | 35 |
| Figure 9 – Testing perplexity for Switchboard I evaluated over 40 epochs | 36 |
| Figure 10 – Training perplexity by n -gram order for Cellular 1..... | 37 |
| Figure 11 – Testing perplexity by n -gram order for Cellular 1 | 38 |
| Figure 12 – Network topology and convergence by hidden layer size when $P = 24$ | 38 |
| Figure 13 – Network topology and convergence by hidden layer size when $P = 48$ | 39 |
| Figure 14 – Testing data perplexity by size of hidden layer for Cellular 1 | 40 |
| Figure 15 – Class membership and probabilities output by the HLM Cluster tool | 44 |
| Figure 16 – Neural network class-based model architecture | 45 |
| Figure 17 – Sample program output produced during neural network training | 54 |
| Figure 18 – Project distribution folder hierarchy..... | 55 |

List of Code Samples

| | |
|---|----|
| Code Sample 1 – Neural network layer class prototype..... | 20 |
| Code Sample 2 – Neural network data-set class prototype..... | 21 |
| Code Sample 3 – Formatting the projection vector from an n -gram context | 22 |
| Code Sample 4 – Feed forward using BLAS..... | 24 |
| Code Sample 5 – Computing error derivatives at the output layer..... | 25 |
| Code Sample 6 – Back propagation using BLAS..... | 26 |
| Code Sample 7 – Accumulating partial derivatives of the projection layer | 26 |
| Code Sample 8 – Updating the network layer weights..... | 27 |
| Code Sample 9 – Data structures for the class-based neural network language model.... | 45 |

1 Introduction

This section presents an overview of the main limitations of traditional n -gram statistical language models and the advantages to be gained by adopting a connectionist approach.

1.1 Neural Network Language Models

In automatic speech recognition (ASR) n -gram language models [Manning & Shütze 1999, Jurafsky & Martin 2000] are traditionally used to compute the prior probability of a word string corresponding to an acoustic hypothesis. They are a simple and effective way of statistically modelling the syntax and semantics of natural language and can be easily and rapidly trained. A serious issue, however, is the poor generalization observed when such models are applied to unseen testing data. Although n -gram models can be generated that accurately capture the parameters of the training data, the discrete nature of the approach does not allow for good generalization to unseen contexts. The discrete word indices have probability distributions that are not smooth. Thus, any alteration in the indices of a particular context can result in an arbitrarily large change in the word probability estimated by the model.

The standard techniques of discounting, interpolation and back-off all aim to address this issue by improving generalization. It is also possible to compensate for data sparsity and improve lexical coverage by training with a very large corpus such that there are far fewer unseen contexts when the model is applied to testing data – contemporary corpus sizes of 500 million words are not uncommon. However, for some problem domains the quantity of training data is very limited. This is especially true of telephone speech conversations for which the cost of producing accurate annotated transcriptions is very high.

A series of papers [Schwenk & Gauvain 2002, 2003, 2004a, 2004b, Schwenk 2004c] has examined a connectionist approach to statistical language modelling. The main advantage of such an approach is that the model can be trained to learn a continuous vector space representation of the word distribution in the training corpus and such continuity permits smoother generalization to unseen contexts than the discrete n -gram model of language. This can improve both the generalization and the quality of language model that can be created from a given limited training corpus.

The neural network language model seeks to improve generalization to unseen data though the use of a continuous space. The history context of the n -gram is projected onto this continuous space and the network is trained to operate as a probability estimator within it. The neural network can be trained simultaneously to learn the continuous space projection and to estimate the probabilities a particular projection represents. Such a model naturally handles unseen contexts. The architecture of the neural network language model is described in detail in section 2.1.

The simplicity of the n -gram model allows for very fast run-time integration with ASR systems. An n -gram language model probability can be retrieved by a single table lookup operation. By contrast, computation of word probabilities using the neural network model

requires a complete forward pass through the network and this can be expensive for larger network topologies. A typical ASR system using n -grams will request many language model probabilities for a given acoustic hypothesis and each of these requires a further n -gram lookup. The neural network model simultaneously computes the probabilities of all words in the vocabulary and these are then immediately available. Since many word probabilities are rarely requested in practice, it is inefficient to compute word probabilities for the entire vocabulary. It is better to specify a much smaller shortlist of high frequency words and use the neural network to compute only these shortlist words. All other word probabilities are computed with a standard n -gram back-off model. A detailed description of the use of such a shortlist is presented in section 2.4.

One of the main issues in developing a connectionist model of natural language is the time required for training [Schwenk 2004c]. The computational complexity of a network capable of modelling the characteristics of natural language is relatively high and training to convergence requires many epochs. To improve the speed of training it is necessary to leverage machine-specific optimized libraries of floating point matrix and vector subroutines. The library used in this project was the Basic Linear Algebra Subroutines (BLAS) provided by the Intel Maths Kernel Library [Intel MKL¹]. The use of BLAS to optimize neural network training is described in section 4.2.

A common approach to dealing with the n -gram issue of data sparsity is to reduce the number of equivalence classes. This can be achieved by automatically clustering vocabulary words into classes. The reduced number of equivalence classes allows n -grams with longer contexts to be captured from a fixed-size set of training data. A neural network language model trained using word classes also benefits from the significantly reduced training time of the smaller network. A class-based approach to connectionist language modelling is described in section 6.1.

¹ <http://www.intel.com/cd/software/products/asm-na/eng/perflib/mkl/index.htm>

1.2 ***Aims and Objectives***

The main aims and objectives of this project are summarized below:

1. Implement and train a neural network-based model of natural language suitable for predicting word probabilities for telephone speech transcriptions.
2. Analyze the performance of the model relative to the traditional n -gram approach to language modelling for a range of different vocabulary sizes.
3. Investigate the properties of the connectionist approach to language modelling by varying the network topology and model parameters.
4. Support efficient training of the model by integrating BLAS into the feed forward and back propagation training algorithms and by incorporating refinements to gradient descent.
5. Implement and analyze a hybrid neural network / n -gram language model by supporting a frequency based shortlist and using standard n -gram probabilities for words outside the shortlist.
6. Evaluate a class-based neural network language model relative to standard n -gram techniques by implementing a model in which both the inputs and outputs of the network are mapped to word classes and the class probabilities estimated by the network are combined with class-conditional unigram probabilities.

The following section describes the theoretical background and architecture of the neural network language model.

2 Background and Theory

This section describes the background and theory of the neural network approach to statistical language modelling.

2.1 Architecture of the Neural Network Language Model

The neural network language model is implemented as a standard fully connected multi-layer perceptron with three layers, termed the projection, hidden and output layers. Only the hidden and output layers have a non-linear activation function. The inputs to the network are the indices of the previous words that define the n -gram context²:

$$h_j = w_{j-n+1}, w_{j-n+2}, \dots, w_{j-1} \quad [4-1]$$

The outputs of the network are the posterior probabilities of all words in the vocabulary given the history:

$$P(w_j = i | h_j), \quad \forall i \in [1 \dots N] \quad [4-2]$$

The architecture of the neural network language model is shown in figure 1 below.

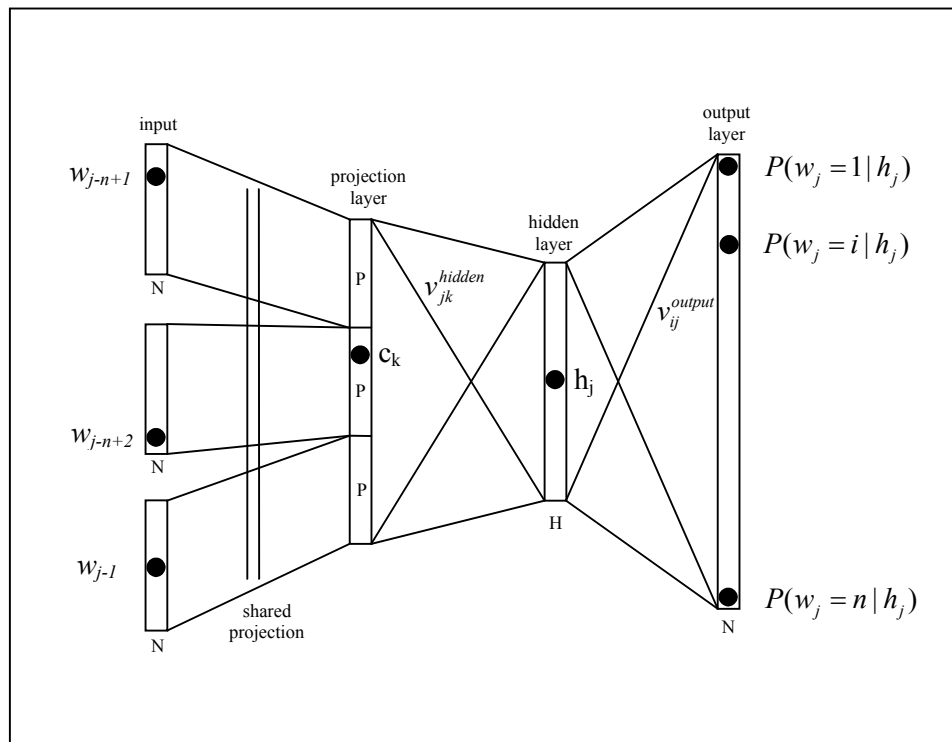


Figure 1 – Connectionist language model architecture³

² Also referred to as a history

³ After Schwenk & Gauvain 2002

Each network training pattern consists of an n -gram context h_j and a target word w_j . The output probabilities are calculated by performing a forward pass through the network with the given context. The output corresponding to the target w_j is used to calculate perplexity (section 2.3). During training, back propagation is used with cross entropy as the error criterion. The back propagation process computes the partial derivatives of the error function with respect to the weights of each layer and gradient descent (section 3.1) computes modifications to these weights that minimize this error function. The following mathematical treatment of the layer operations is adapted from the presentation in [Schwenk & Gauvain 2002].

2.1.1 The Projection Layer

The projection layer maps the discrete word indices of an n -gram context to a continuous vector space.

A standard 1-of- n encoding scheme for representing non-numeric neural network inputs is used. Each word in the context is represented by a vector of binary valued elements in which for a context word w_i only the i^{th} element of the vector is set to 1 and all other values are set to 0. Such a scheme ensures that the index value of each word in the context is not captured when the network is trained, and this is desirable since the index values assigned to each word in the vocabulary are entirely arbitrary. It also enables a highly compact representation of input contexts and training patterns – both can be represented as simple lists of word indices, i.e. single integer values.

The projection layer is shared such that for contexts containing the same word multiple times, the same set of weights is applied to form each part of the projection vector. This organization effectively increases the amount of data available for training the projection layer weights since each word of each context training pattern individually contributes changes to the weight values. Since the purpose of the projection layer is to obtain a continuous vector representation of the context word's index, the matrix of weights is common to all words in the history. This approach means that the ordering of the words that form n -gram histories is not captured by the projection layer weights. Instead, the ordering of context words is made explicit in the output of the projection layer and learned by the hidden layer.

The number of neurons in the projection layer is defined in the network topology configuration file (see Appendix B for details). Each neuron in the projection layer is represented by a number of weights equal to the size of the vocabulary. The projection layer differs from the hidden and output layers by not using a non-linear activation function. Its purpose is simply to provide an efficient means of projecting the given n -gram context onto a reduced continuous vector space for subsequent processing by hidden and output layers trained to classify such vectors. Given the one-or-zero nature of the input vector elements, the output for a particular word with index i is simply the i^{th} column of the trained matrix of projection layer weights (where each row of the matrix represents the weights of a single neuron). Figure 2 shows for a trivial topology how the output of the projection layer can be efficiently assembled by copying columns from the projection layer weights matrix.

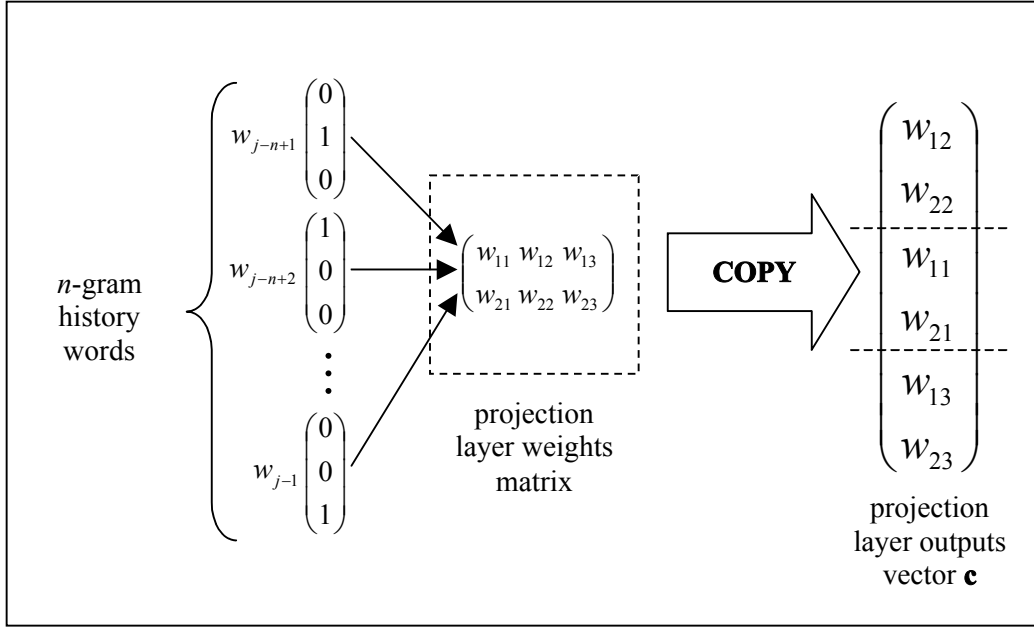


Figure 2 – Formatting the projection vector from an n -gram context

For an n -gram history of length $h = n - 1$ and a projection layer size of P neurons, the final output from the projection layer is a vector of real values of length $h * P$. The resulting projection vector is used as the input to the next layer in the network: the hidden layer.

2.1.2 The Hidden Layer

The hidden layer processes the output of the projection layer and is also created with a number of neurons specified in the topology configuration file. The optimal size for the hidden layer is a function of the size of the vocabulary and is therefore a system parameter that must be tuned to the problem domain. The number of weights associated with each hidden layer neuron is defined by the dimensionality of the output from the projection layer.

The hidden layer activities h_j are computed by applying the \tanh function to the weighted sum of the projection layer activities c_k :

$$h_j = \tanh\left(\sum_k v_{jk}^{hidden} c_k + b_j^{hidden}\right) \quad \forall k \in 1 \dots H \quad [4-3]$$

The main reason for using \tanh as an activation function instead of the more common logistic regression sigmoid is that the rate of convergence of \tanh is often faster than sigmoid [Bishop 1995]. This is likely to be particularly true of the very large network sizes and data-sets that characterize a neural network language model. The \tanh function generates continuous outputs in the range $-1 \leq h_j \leq 1$.

If x is defined as the weighted sum of the projection layer activities, then:

$$\tanh(x) = \frac{e^x + e^{-x}}{e^x - e^{-x}} \quad [4-4]$$

Differentiating the *tanh* function gives the following expression:

$$\frac{d}{dx}(\tanh(x)) = 1 - \left(\frac{e^x + e^{-x}}{e^x - e^{-x}} \right)^2 \quad [4-5]$$

If the output $g(h_j)$ of each hidden layer neuron is cached during forward propagation, then the gradient of the error function with respect to the weights of that neuron can be calculated efficiently using:

$$g'(h_j) = 1 - (g(h_j))^2 \quad [4-6]$$

A matrix-vector treatment of feed forward and back propagation for the hidden layer together with sample code showing their implementation in terms of BLAS operations can be found in section 4.2.

2.1.3 The Output Layer

The output layer processes the output from the hidden layer and is created with a number of neurons equal to the size of the vocabulary (or, if enabled, the size of the shortlist) and a number of weights equal to the dimensionality of the hidden layer output. There are also outputs for OOV and the sentence end token. When training with a shortlist (section 2.4) there is no output for OOV. The purpose of the output layer is to compute the posterior probabilities of each word w_j in the vocabulary given the n-gram context h_j that was fed forward through the network.

The output layer uses a *softmax* normalization of the weighted sum of hidden layer activities. The activities of the output layer are first computed:

$$o_i = \sum_j v_{ij}^{output} h_j + b_i^{output} \quad [4-7]$$

The normalization is achieved using the non-linear *softmax* activation function:

$$p_i = \frac{e^{o_i}}{\sum_{k=1}^N e^{o_k}} \quad \forall i = 1 \dots N, \quad \sum_{i=1}^N p_i = 1 \quad [4-8]$$

The use of *softmax* as an activation function allows a probabilistic interpretation of the neural network outputs. The output is continuous over the interval $0 \leq p_i \leq 1$. For the neural network language model, the output of the i^{th} neuron corresponds directly to the

conditional probability that the next word is the i^{th} word in the wordlist, i.e. $P(w_{j=i} | w_{j-n+1}, w_{j-n+2}, \dots, w_{j-1}) = P(w_j = i | h_j)$.

A matrix-vector treatment of the output layer with sample code showing the required BLAS operations and activation function with normalization can be found in section 4.2.

2.2 Cross Entropy Error Criterion

The use of cross entropy as an error criterion allows the partial derivatives of the errors at the output layer to be trivially computed. The total cross entropy of a neural network with N outputs (where N is the size of the vocabulary or shortlist) is defined by the expression:

$$E = \sum_{i=0}^{N-1} d_i \log p_i \quad [4-9]$$

In equation [4-9], d_i represents the value of the desired output, i.e. 1.0 if the target word given the n -gram history h_j is i , and 0.0 otherwise. The derivation of the cross-entropy error measure is defined by the expression:

$$\frac{\partial E}{\partial o_i} = p_i - d_i \quad [4-10]$$

The use of cross entropy as an error function can be shown to minimize the perplexity of the training data [Bishop 1995].

2.3 Perplexity Calculation

The primary metric for assessing the performance of a neural network applied to the task of statistical language modelling is the perplexity obtained by testing the network on previously unseen data. The advantage of perplexity as a measure of language model quality is that it depends only on the model and can thus be easily calculated without the time-consuming experiments required by integration with a full speech recognition system.

Perplexity (PP) is a measure of the average branching factor of the grammar when predicting a word from some observed n -gram context. When comparing language models of the same type, a reduction in perplexity usually confers a reduction in the word error rate (WER) of a speech recognition system. It is less certain that a neural network language model with a lower perplexity than an n -gram model will reduce the WER since the general rule that a reduction in PP leads to a reduction in WER only really applies when comparing language models of the same type.

During training and testing, perplexity is calculated over the entire epoch – i.e. after processing every training pattern in the data-set. The perplexity is calculated from the entropy of the network outputs using the following formulae:

$$H = -\sum_{j=0}^N \log_2 P(w_j = i | h_j) \quad [4-11]$$

Given this definition of entropy, the perplexity is defined by:

$$PP = 2^H \quad [4-12]$$

For a neural network language model, the probabilities are obtained directly from the output corresponding to the training pattern's target word. If the network output for a given pattern is consistently high, then the sum of log probabilities and – therefore – the perplexity will be low. The perplexity of the training data is used as the error criterion for convergence. When the perplexity ceases to decrease training has reached a (possibly local) minimum.

2.4 Neural Network Shortlist

The main problem with a language model based on a neural network is the network size required to capture the model parameters of a large vocabulary and the correspondingly long training time required to learn such parameters. Since many of the words occur relatively infrequently in real data, there is limited utility in explicitly modelling such words in the network. It is better, instead, to make use of a shortlist of interesting words and compute only a subset of the conditional probabilities using the neural network. The probabilities of the remaining words are computed using a standard n -gram back-off language model. For a specified shortlist, the conditional probability of a word given its history can be expressed as in [Schwenk 2004c]:

$$P(w_j | h_j) = \begin{cases} P_N(w_j | h_j) \cdot P_S(h_j) & \text{if } w_j \in \text{shortlist} \\ P_B(w_j | h_j) & \text{otherwise} \end{cases} \quad [4-13]$$

The term $P_S(h_j)$ is the total probability mass assigned to words in the shortlist for the given context and is defined by:

$$P_S(h_j) = \sum_{w \in \text{shortlist}} P_B(w | h_j) \quad [4-14]$$

The probability masses required to support the shortlist implementation can be calculated in advance for a given training or testing file and stored as a simple stream of values to be read by the neural network tool. This allows the values of $P_S(h_j)$ and $P_B(w_j | h_j)$ required for perplexity calculations to be retrieved by simple table lookup operations.

2.5 Language Model Data

Two separate corpora were used for training and testing the neural network language model, both published by the Linguistic Data Consortium⁴. Both are transcriptions of

⁴ <http://www ldc.upenn.edu/>

telephone conversations. The larger of the two, Switchboard I⁵ (*swbdI*) contains around 2400 telephone conversations by 543 individuals speaking on one of 70 topics. The smaller corpus, Cellular 1⁶ (*cell1*), collects data from a balanced set of GSM mobile phone users and includes 250 transcribed calls. Table 1 compares the sizes of the two corpora.

| <i>Type</i> | <i>Corpus</i> | <i>Sentences</i> | <i>Words</i> |
|-----------------|---------------------------|------------------|--------------|
| <i>Training</i> | <i>swbdI.dat</i> | 245661 | 3032638 |
| | <i>cell1.dat</i> | 31635 | 222296 |
| <i>Testing</i> | <i>h5eval03_noexp.dat</i> | 8770 | 69758 |
| | <i>h5dev_01_cell1.dat</i> | 1750 | 20683 |

Table 1 – Cellular 1 and Switchboard I corpus statistics

The Cellular 1 training data and development test set both expand contractions, i.e. “ISN’T” is expanded to “IS NOT”. Switchboard I does not expand contractions; they remain in their original unaltered form. This makes it harder to use the larger *h5eval03_noexp.dat* testing file to evaluate the Cellular 1 trained models since there is a mismatch in the way many of the most common words are transcribed and this adversely impacts perplexity.

Table 2 below shows the frequency of OOV words in the training and testing data for each of the target vocabulary sizes.

| <i>Vocabulary Size</i> | <i>Cellular 1</i> | | <i>Switchboard I</i> | |
|------------------------|-------------------|-------------|----------------------|-------------|
| | <i>Train</i> | <i>Test</i> | <i>Train</i> | <i>Test</i> |
| 50 | 96414 | 8504 | 1407356 | 31907 |
| 100 | 68046 | 6106 | 1034870 | 22874 |
| 500 | 26159 | 2498 | 459486 | 10302 |
| 1000 | 15384 | 1663 | 298326 | 7069 |
| 2000 | 7935 | 1119 | 181298 | 4678 |
| full | 0 | 487 | 0 | 811 |

Table 2 – Cellular 1 and Switchboard I OOV statistics

Comparing tables 1 and 2 shows that the OOV rate for both corpora is very high when the vocabulary size is small. The training and testing OOV rates for a 2000 word vocabulary are approximately 3.6% and 5.4% for *cell1* and 6.0% and 6.7% for *swbdI* respectively. These are acceptable OOV rates for evaluating the training and convergence experiments.

2.6 Neural Network Training Patterns

The main aim of training is to create a model that can accurately predict a target word from a context. The outputs of the network consist of the full vocabulary (or shortlist word set) and two additional outputs representing the end-of-sentence symbol *</s>* and

⁵ <http://www ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC97S62>

⁶ <http://www ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2001T14>

a special out-of-vocabulary (OOV) symbol, conventionally `!!UNK`. The inputs to the network are the full set of vocabulary words, the OOV symbol and the start symbol `<s>`.

The training corpus consists of telephone conversation transcriptions and such spontaneous speech typically comprises many short sentences. The average sentence length of the `cell1` and `swbd1` corpora is approximately seven and twelve words respectively. The short nature of the training sentences requires a sensible strategy for dealing with the start of sentences. The solution presented here prefixes additional start symbols `<s>` to each training pattern that predicts a target word at sentence position less than the length of the n -gram. To make this explicit, the full set of training patterns presented to the network for the `cell1` corpus sentence “`<s> WHERE ARE YOU NOW </s>`” is shown in table 3 below:

| <i>Context words h_j</i> | | | <i>Target word w_j</i> |
|---------------------------------------|------------------------|------------------------|-------------------------------------|
| <code><s></code> | <code><s></code> | <code><s></code> | WHERE |
| <code><s></code> | <code><s></code> | WHERE | ARE |
| <code><s></code> | WHERE | ARE | YOU |
| WHERE | ARE | YOU | NOW |
| ARE | YOU | NOW | <code></s></code> |

Table 3 – Padded training patterns for sentence initial words

The main advantage of this approach is that it compensates for the prevalence of short sentences in the training corpus. Although many training patterns include the sentence start symbol, the position of the start symbol in the context can be learned by the network and this allows it to more accurately capture the underlying characteristics of the data. Ignoring the first few words of a sentence until sufficient words to create a valid n -gram have been observed would be risky for a corpus predominantly composed of short sentences.

3 Gradient Descent and Network Convergence

This section describes gradient descent and refinements that aim to improve the rate at which the network is trained⁷.

3.1 Gradient Descent Introduction

The error values computed during back propagation represent the gradient of the error surface and can be used to calculate changes in the weights that reduce the value of the error function, a procedure known as gradient descent. Each bias and weight in the network is updated according to equation [3-1].

$$\mathbf{w}_i^{(k)}[\tau + 1] = \mathbf{w}_i^{(k)}[\tau] - \eta \left. \frac{\partial E}{\partial \mathbf{w}_i^{(k)}} \right|_{\boldsymbol{\theta}[\tau]} \quad [3-1]$$

The constant η is the learning rate and the term to which it is applied is the partial derivative of the error with respect to the i^{th} weight of layer k for model parameters $\boldsymbol{\theta}$ at time τ . Iteratively updating the weights in this manner allows training to find minimums in the error surface.

3.2 Gradient Descent Optimizations

The main issue in creating a neural network model of language is the very long training time resulting from the network complexity and size of the training data. It is therefore very important to try and improve the rate of convergence. The following sections describe the gradient descent optimizations supported by the neural network training tool.

3.2.1 Variable Learning Rate

A standard refinement to gradient descent is to use a variable learning rate that is updated after each training epoch. The advantage of this approach is that it allows learning to accelerate whilst descending through steep regions of the error surface and for stabilization and finer-grained exploration whenever the learning rate grows too large.

The learning rate η is varied according to the change between epochs in the cost function $E(\boldsymbol{\theta})$. For the neural network language model, this error function is defined by the cross entropy. If the previous learning rate decreased the value of the cost function, then the learning rate is increased by a small amount. If the previous rate increased the cost function, the learning rate is significantly decreased. This update rule can be expressed as follows:

$$\eta[\tau + 1] = \begin{cases} 1.1\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) < E(\boldsymbol{\theta}[\tau - 1]) \\ 0.5\eta[\tau]; & \text{if } E(\boldsymbol{\theta}[\tau]) > E(\boldsymbol{\theta}[\tau - 1]) \end{cases} \quad [3-2]$$

⁷ After Gales 2001

The initial learning rate is specified in the topology configuration file using the parameter `learning_rate` (see Appendix B). The default value is 0.1. During training, all decreases in the learning rate are explicitly recorded in the log file (see Appendix C).

3.2.2 Momentum Term

Including a momentum term in the update rule confers two advantages: successive updates are smoothed, and it is possible to slide over small local minima during descent. Unfortunately, the momentum term scaling constant is an additional system parameter than needs to be tuned to the problem domain and network topology. Additionally, if the descent chances to strike the minimum, the momentum term will cause it to overshoot – this can slow the final stages of descent. The update rule for the vector of weights associated with each neuron i is given by the expression:

$$\mathbf{w}_i[\tau + 1] = \mathbf{w}_i[\tau] + \Delta\mathbf{w}_i[\tau] \quad [3-3]$$

The vector of weight changes $\Delta\mathbf{w}_i$ at time τ is defined by:

$$\Delta\mathbf{w}_i[\tau] = -\eta[\tau + 1] \left. \frac{\partial E}{\partial \mathbf{w}_i} \right|_{\mathbf{0}[\tau]} + \alpha[\tau] \Delta\mathbf{w}_i[\tau + 1] \quad [3-4]$$

The value of α – the momentum term scaling constant – is specified in the topology configuration file using the parameter `momentum_alpha` (see Appendix B). The default value is 0.95. The inclusion of a momentum term can be disabled by setting α to zero.

4 Neural Network Language Model Implementation

This section describes the implementation of the neural network language model.

4.1 Data Structures and Algorithms

The descriptions of the feed forward and back propagation training process make reference to a number of implementation specific engineering and design decisions. This section provides a brief overview of the main data structures and general nature of the training and testing algorithms.

4.1.1 Neural Network Layer

The core data structure manipulated by the training algorithm represents the model parameters of a neural network layer. Each layer is defined by three properties – the number of neurons, the number of weights and the number of outputs. The reason for maintaining separate counts of neurons and outputs is to support the composite projection produced at the projection layer outputs by mapping each individual context word to the continuous vector space – for the hidden layer the number of outputs is $h * P$, where h is the length of the n -gram history and P the number of projection layer neurons. The hidden and output layers have the same number of neurons as outputs.

These three properties are used at initialization time to allocate sufficient storage to represent the network model parameters (i.e. the weights and biases) and to allow efficient run-time operation of the feed forward and back propagation algorithms by allowing BLAS operations to function “in-place” without requiring further memory management. An abbreviated class prototype for the network layer representation `NNLayer` is shown in code sample 1 below:

```
class NNLayer
{
public:
    NNFloat** weights;
    NNFloat** changesW;
    NNFloat** previousW;

    NNFloat* biases;
    NNFloat* changesB;
    NNFloat* previousB;

    NNFloat* outputs;
    NNFloat* errors;

    unsigned int cNeurons;
    unsigned int cWeights;
    unsigned int cOutputs;
};
```

Code Sample 1 – Neural network layer class prototype

The weights and biases are stored in the `weights` and `biases` data members. All matrix storage is in “row major” format – i.e. the elements of each row of the matrix are located in contiguous storage. A special allocation routine is used to enable 2-d indexing

of dynamically allocated arrays. The `changesW` and `changesB` data members store the accumulated partial derivatives for each pattern and are necessary to support batch based learning. The `previousW` and `previousB` data members store the previous changes in the weights and biases and are required to support the momentum term of gradient descent. If no previously trained network is specified on the command-line, the `weights` and `biases` data members are initialized with random floating point values in the interval defined in the configuration file (see Appendix B for a complete description of the configuration settings). The remaining members are all initialized to zero.

The `outputs` and `errors` members are used to store the vector of outputs computed during feed forward and the vector of errors computed during the back propagation stages of training. Allocating these vectors at initialization time allows BLAS to write the results of the matrix-vector operations directly to the layer's storage and ensures that cached values are available for efficient computation of errors and weight updates.

4.1.2 Neural Network Data-set

All of the data required for training and testing the neural network language model is exposed by a single object: the `NNData` class. There are three main types of data: the wordlist defining the vocabulary, the n -gram contexts for the specified data file, and the probability masses required for normalization and calculation of n -gram probabilities in shortlist mode. An abbreviated class prototype is shown in code sample 2 below:

```
class NNData
{
public:
    vector<NNWordId> contexts;           // Word indicies extracted from data file.
    map<string, NNWordId> wordlist;     // Mapping from orthography to unique word ID.
    vector<NNFloat> probMassesS;       // Masses for shortlist normalization.
    vector<NNFloat> probMassesB;       // Masses for backoff probabilities.
};
```

Code Sample 2 – Neural network data-set class prototype

The word list is represented as an STL map and provides a very simple orthography to unique integer word identifier mapping. The order in which words are added to the word list dictates their index value. Since the vocabulary file is assumed to be ordered by frequency (most frequent words first) the words with low word identifier values are the most frequent words in the corpus. This allows a very simple test for shortlist membership: if the index is less than the size of the shortlist then the word is a shortlist word. The wordlist is extended with special symbols that are not read from the vocabulary file – those that represent sentence start, sentence end, and the special unknown word symbol. The wordlist is created before training or testing begins and does not need to be accessed during network simulation. It is used purely to simplify the generation of the n -gram context / target data that constitutes the patterns processed by the network.

The training or testing data is represented as a simple array of integer word identifiers. Each entry corresponds to a single target for a given training or testing pattern, i.e. the w_j

in the expression $P(w_j | h_j)$. Thus, when reading words from the specified data file the sentence start symbol is skipped (the sentence end markers sufficiently demarcate sentences). Various statistics are collected and logged: the total number of words, the percentage of OOV words, the total number of sentences and average number of words per sentence, and the total number of patterns extracted from the data file. A method exposed by the `NNData` class allows the next context to be retrieved during training or testing. If the word is near the start of the sentence, the returned context is prefixed with additional sentence start symbols in accordance with the preceding discussion of section 2.6. After returning a context, the current position is incremented in preparation for the next context. The class effectively exposes a sliding window onto the word stream of the data file.

4.1.3 Forming the Projection

Given a list of context word indices, `context`, and a matrix of projection layer weights, `weights`, the generation of the projection layer outputs can be achieved using the code shown in code sample 3 below:

```
// Write the projection into the layerP outputs
NNFloat* p = layerP->outputs;
for (int i=0; i<historyLength; i++) {
    for (int n=0; n<layerP->cNeurons; n++) {
        *p++ = layerP->weights[n][ context[i] ];
    }
}
```

Code Sample 3 – Formatting the projection vector from an n -gram context

The code sample shows that obtaining the projection for a given context can be efficiently implemented by simply copying one column of the projection layer weights matrix for each word in the context.

4.2 Optimized Training with BLAS

This section describes how BLAS can be used to optimize the training of the neural network language model.

4.2.1 Neural Network Architecture with BLAS

The computationally intensive nature of a large neural network and data-set require a highly optimized training procedure. Since the feed forward and back propagation stages of neural network training can be elegantly formulated in terms of matrix and vector operations, it is possible to leverage linear algebra routines that optimize these operations for specific machine architectures.

The library used in this project is Intel's Maths Kernel Library (MKL) and their C interface to the Basic Linear Algebra Subprograms (BLAS) [Blackford et al⁸]. The MKL fully supports levels 1, 2 and 3 of BLAS and methods from all three of these levels are used while training the neural network language model.

⁸ BLAS Quick Reference: <http://www.netlib.org/blas/blasqr.ps>

This section describes how the calculation of network layer activities and back propagation can be expressed in terms of matrix and vector operations and integrated with BLAS. Figure 3 below recasts the network architecture in terms of matrix-vector operations.

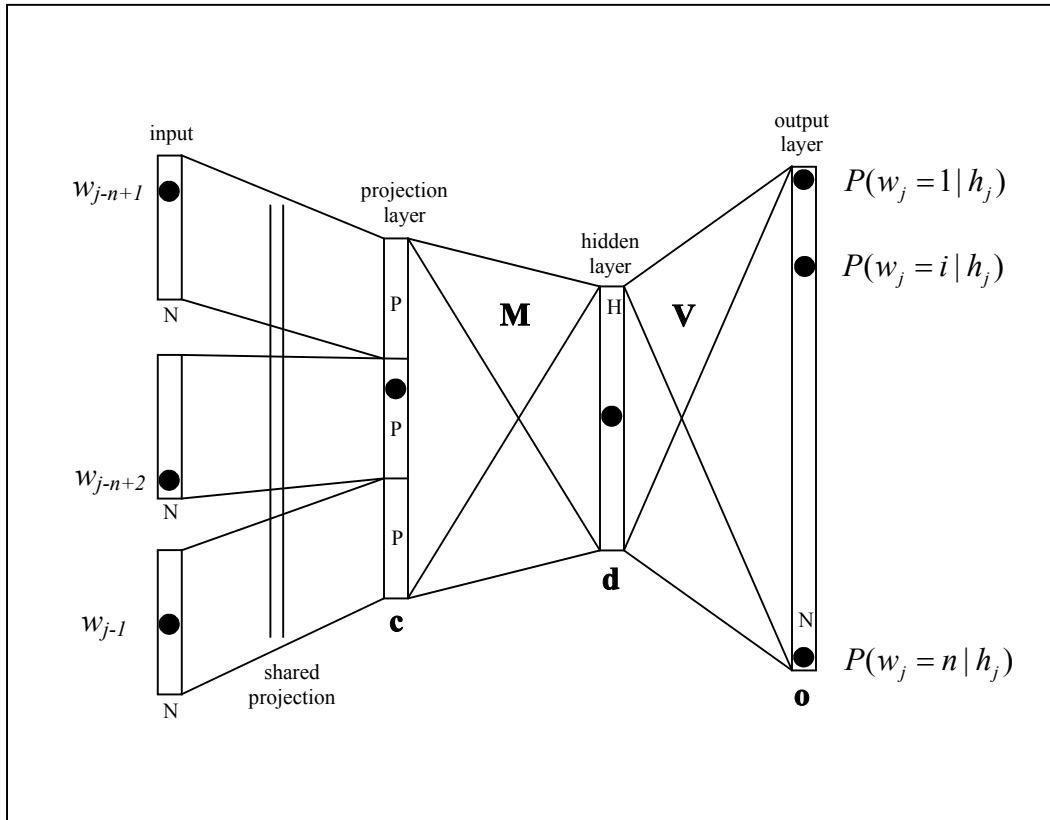


Figure 3 – Matrix-vector connectionist language model architecture⁹

4.2.2 Feed-forward using BLAS

Assuming the architecture shown in the figure, and starting from a projection layer output vector \mathbf{c} , the feed forward phase of the neural network can be expressed by the sequence of matrix and vector operations shown in equations [4-1] (this matrix-vector treatment of feed forward and back propagation follows [Schwenk 2004c]).

$$\begin{aligned}
 \mathbf{d} &= \tanh(\mathbf{M} * \mathbf{c} + \mathbf{b}) \\
 \mathbf{o} &= \mathbf{V} * \mathbf{d} + \mathbf{k} \\
 \mathbf{p} &= \exp(\mathbf{o}) / \sum_{k=1}^N e^{o_k}
 \end{aligned}
 \tag{4-1}$$

Although BLAS can be used to compute the matrix and vector operations, the *tanh* and *softmax* activations must be evaluated on a per-neuron basis. This element-wise application of the activation functions can be seen clearly in steps two and four of code

⁹ After Schwenk 2004c

sample 4 below. The listing shows how BLAS is used to optimize the feed forward process. The NN_* family of function names represent macros that selectively enable single or double precision prototypes according to the `_USE_DOUBLE_PRECISION` compiler flag in the `NNCommon.h` header file. Note that for formatting reasons the true names of the objects `layerP`, `layerH` and `layerO` are here abbreviated to `P`, `H` and `O` respectively. Please refer back to section 4.1.1 for details of the network layer representation and data members exposed by these objects.

| | | |
|--|---|---|
| $\mathbf{d} = \tanh(\mathbf{M} * \mathbf{c} + \mathbf{b})$ | <pre>// Copy hidden layer biases vector to outputs. NN_BLAS_COPY(H->cNeurons, H->biases, 1, H->outputs, 1); // Feed forward for the hidden layer. NN_BLAS_GEMV(CblasRowMajor, CblasNoTrans, H->cNeurons, H->cWeights, 1.0f, (NNFloat*)(H->weights), H->cWeights, P->outputs, 1, 1.0f, H->outputs, 1);</pre> | 1 |
| | <pre>// Apply hidden layer activation function. for (int n=0; n<H->cNeurons; n++) { H->outputs[n] = NN_TANH(H->outputs[n]); }</pre> | 2 |
| $\mathbf{o} = \mathbf{V} * \mathbf{d} + \mathbf{k}$ | <pre>// Copy output layer biases vector to outputs. NN_BLAS_COPY(O->cNeurons, O->biases, 1, O->outputs, 1); // Feed forward for the output layer. NN_BLAS_GEMV(CblasRowMajor, CblasNoTrans, O->cNeurons, O->cWeights, 1.0f, (NNFloat*)(O->weights), O->cWeights, H->outputs, 1, 1.0f, O->outputs, 1);</pre> | 3 |
| $\mathbf{p} = \exp(\mathbf{o}) / \sum_{k=1}^N e^{o_k}$ | <pre>NNFloat normalization = 0.0f; // Apply activation function. for (int n=0; n<O->cNeurons; n++) { O->outputs[n] = NN_EXP(O->outputs[n]); normalization += O->outputs[n]; } // Normalize outputs for posterior probabilities. for (int n=0; n<O->cNeurons; n++) { O->outputs[n] /= normalization; }</pre> | 4 |

Code Sample 4 – Feed forward using BLAS

The values of the outputs vector on completion of the feed forward process represent the normalized posterior probabilities of the vocabulary words for the given input context. These are the probabilities required to compute perplexity.

4.2.3 Back Propagation using BLAS

When training the network, all of the outputs save one have a target value of 0.0 in equation [4-9] and this means that the errors at the output layer can be computed in a trivial fashion. Given the vector of outputs, `outputs`, the initially empty vector of errors, `errors`, and the target word identifier, `target`, the errors at the output layer can be computed using the method shown in code sample 5:

```
// Compute output layer error derivatives.
for (int n=0; n<layerO->cOutputs; n++) {
    layerO->errors[n] = (n != target) ? layerO->outputs[n] : layerO->outputs[n] - 1.0f;
}
```

Code Sample 5 – Computing error derivatives at the output layer

During training, these derivatives of the error function at the output layer $\Delta \mathbf{k}$ are used to back propagate the errors through the sequence of matrix and vector operations shown in equations [6-2] below¹⁰. The terms λ and α represent the learning rate and weight decay regularization constants respectively. Note that the operator $\cdot *$ represents element-wise vector multiplication – i.e. a new vector is created whose elements are formed by a direct multiplication of the corresponding elements of each source vector.

$$\begin{aligned}
 \mathbf{k} &= \mathbf{k} - \lambda \Delta \mathbf{k} \\
 \Delta \mathbf{b} &= \mathbf{V}^T * \Delta \mathbf{k} \\
 \mathbf{V} &= -\lambda \Delta \mathbf{k} * \mathbf{d}^T + \alpha \mathbf{V} \\
 \Delta \mathbf{b} &= \Delta \mathbf{b} \cdot *(1 - \mathbf{d} \cdot * \mathbf{d}) \\
 \mathbf{b} &= \mathbf{b} - \lambda \Delta \mathbf{b} \\
 \Delta \mathbf{c} &= \mathbf{M}^T * \Delta \mathbf{b} \\
 \mathbf{M} &= -\lambda \Delta \mathbf{b} * \mathbf{c}^T + \alpha \mathbf{M}
 \end{aligned}
 \tag{4-2}$$

For batch based updates, equations [6-2] need to be separated into error calculation and update steps. The quantities that need to be accumulated during each stage of back propagation are listed explicitly in the following table. Code sample 6 below shows how back propagation can be efficiently implemented in terms of BLAS operations and element-wise vector multiplication (step 4):

| | | |
|--|---|---|
| $\mathbf{k} = \mathbf{k} - \lambda \Delta \mathbf{k}$ Accumulates $\Delta \mathbf{k}$ | <pre>// Accumulate bias changes at the output layer. NN_BLAS_AXPY(O->cOutputs, 1.0f, O->errors, 1, O->changesB, 1);</pre> | 1 |
| $\Delta \mathbf{b} = \mathbf{V}^T * \Delta \mathbf{k}$ | <pre>// Compute intermediate hidden layer errors. NN_BLAS_GEMV(CblasRowMajor, CblasTrans, O->cNeurons, O->cWeights, 1.0f,</pre> | 2 |

¹⁰ After Schwenk 2004c

| | | |
|--|---|---|
| | <pre>(NNFloat*)(O->weights), O->cWeights, O->errors, 1, 0.0f, H->errors, 1);</pre> | |
| $\mathbf{V} = -\lambda \Delta \mathbf{k} * \mathbf{d}^T + \alpha \mathbf{V}$ Accumulates $\Delta \mathbf{k} * \mathbf{d}^T$ | <pre>// Accumulate weights matrix changes at the output layer. NN_BLAS_GER(CblasRowMajor, O->cNeurons, O->cWeights, 1.0f, O->errors, 1, H->outputs, 1, (NNFloat*)(O->changesW), O->cWeights);</pre> | 3 |
| $\Delta \mathbf{b} = \Delta \mathbf{b} * (1 - \mathbf{d} * \mathbf{d})$ | <pre>// Compute error derivatives at the hidden layer. for (int n=0; n<H->cNeurons; n++) { H->errors[n] *= (1.0f - H->outputs[n]*H->outputs[n]); }</pre> | 4 |
| $\mathbf{b} = \mathbf{b} - \lambda \Delta \mathbf{b}$ Accumulates $\Delta \mathbf{b}$ | <pre>// Accumulate bias changes at the hidden layer. NN_BLAS_AXPY(H->cOutputs, 1.0f, H->errors, 1, H->changesB, 1);</pre> | 5 |
| $\Delta \mathbf{c} = \mathbf{M}^T * \Delta \mathbf{b}$ | <pre>// Compute error derivatives at the projection layer. NN_BLAS_GEMV(CblasRowMajor, CblasTrans, H->cNeurons, H->cWeights, 1.0f, (NNFloat*)(H->weights), H->cWeights, H->errors, 1, 0.0f, P->errors, 1);</pre> | 6 |
| $\mathbf{M} = -\lambda \Delta \mathbf{b} * \mathbf{c}^T + \alpha \mathbf{M}$ Accumulates $\Delta \mathbf{b} * \mathbf{c}^T$ | <pre>// Accumulate weights matrix changes at the hidden layer. NN_BLAS_GER(CblasRowMajor, H->cNeurons, H->cWeights, 1.0f, H->errors, 1, P->outputs, 1, (NNFloat*)(H->changesW), H->cWeights);</pre> | 7 |

Code Sample 6 – Back propagation using BLAS

The projection layer has a linear activation function, so the calculation of changes to the weights and biases is trivial. Given the vector of errors for the projection layer $\Delta \mathbf{c}$ calculated in step 6 above, careful use of modular arithmetic allows the changes to the projection layer for a given array of context words `context` to be accumulated using the minimal code of code sample 7 below:

```
// Compute changes in the biases for the projection layer.
for (int i=0; i<layerP->cOutputs; i++) {
  layerP->changesB[i % layerP->cNeurons] += layerP->errors[i];
}

// Compute changes in the weights for the projection layer.
for (int c=0; c<contextLength-1; c++) {
  for (int i=0; i<layerP->cOutputs; i++) {
    layerP->changesW[i % layerP->cNeurons][context[c]] += layerP->errors[i];
  }
}
```

Code Sample 7 – Accumulating partial derivatives of the projection layer

When a full batch of training patterns has been processed by the network, the weights and biases of each layer are updated.

4.2.4 Updating the Network Weights

Without an implementation of the matrix bunch-mode training described in [Schwenk 2004c] it is more efficient not to use BLAS for the updates. Since there is no matrix-add operation exposed by BLAS, it is necessary to loop through the layer weights, factoring in the learning rate, momentum term and weight decay. Although the update step can be slow for very large network topologies, the use of batch mode learning ensures that updates are applied relatively infrequently. The majority of processing required for training remains in the feed forward and back propagation procedures. Code sample 8 below shows the layer update methods.

```
// Update as a function of weight value, accumulated changes and previous changes.
inline void NNLayer::UpdateWeight(NNFloat& w, NNFloat& c, NNFloat& p)
{
    p = -config.learningRate * c + config.momentumAlpha * p;
    w += p;
    w *= config.weightDecay;
    c = 0.0f;
}

// Updates a layer with the changes computed during back propagation.
void NNLayer::ApplyChanges()
{
    for (int n=0; n<cNeurons; n++) {
        UpdateWeight(biases[n], changesB[n], previousB[n]);
        for (int w=0; w<cWeights; w++) {
            UpdateWeight(weights[n][w], changesW[n][w], previousW[n][w]);
        }
    }
}
```

Code Sample 8 – Updating the network layer weights

The learning rate, momentum scaling constant, and weight decay settings are all specified in the configuration file (see Appendix B for details).

4.3 Neural Network Training Procedure

Figure 4 shows a process-oriented flow diagram for the neural network training algorithm.

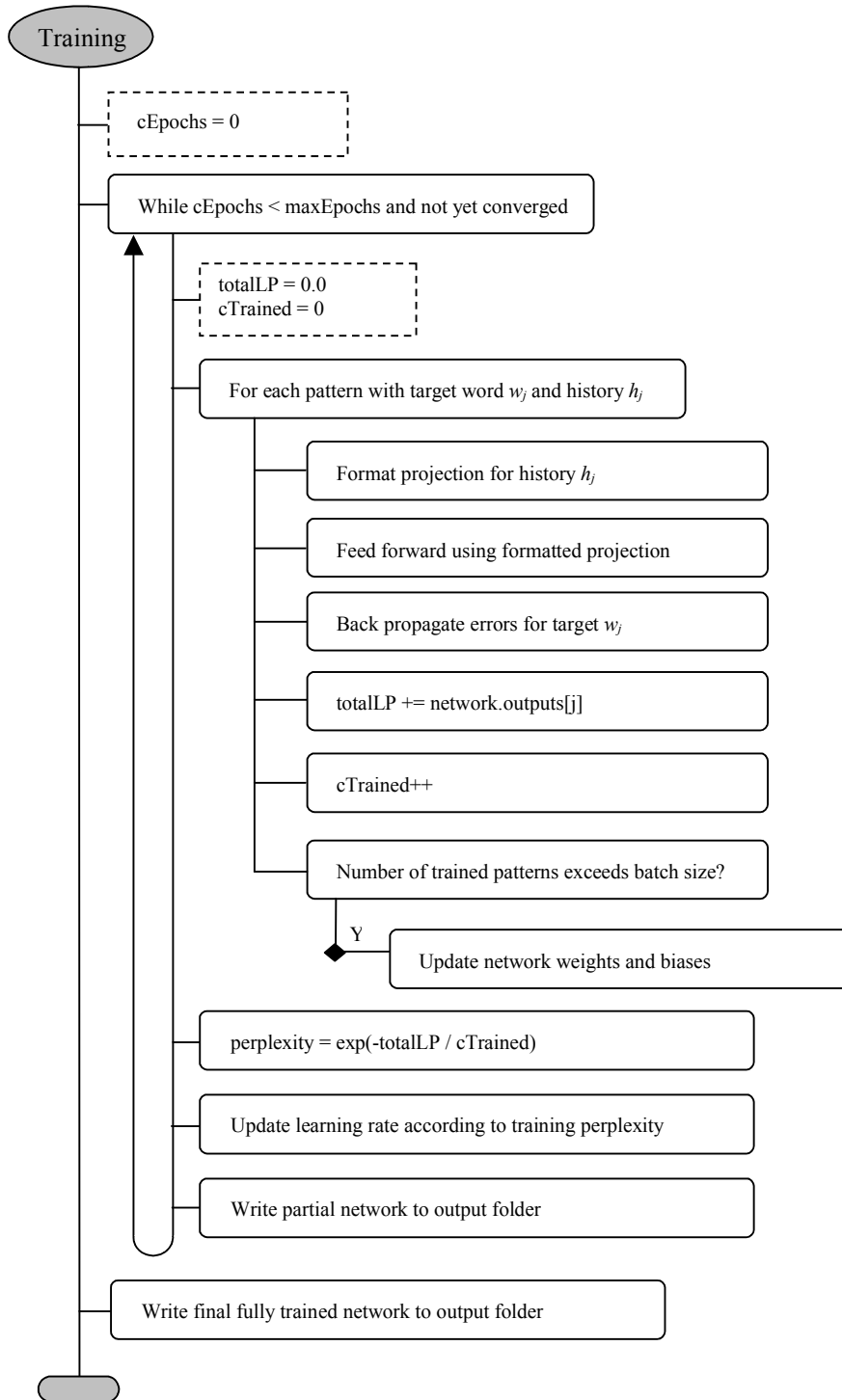


Figure 4 – Neural network training algorithm

4.4 Neural Network Testing Procedure

Figure 5 shows a process-oriented flow diagram for the neural network testing algorithm.

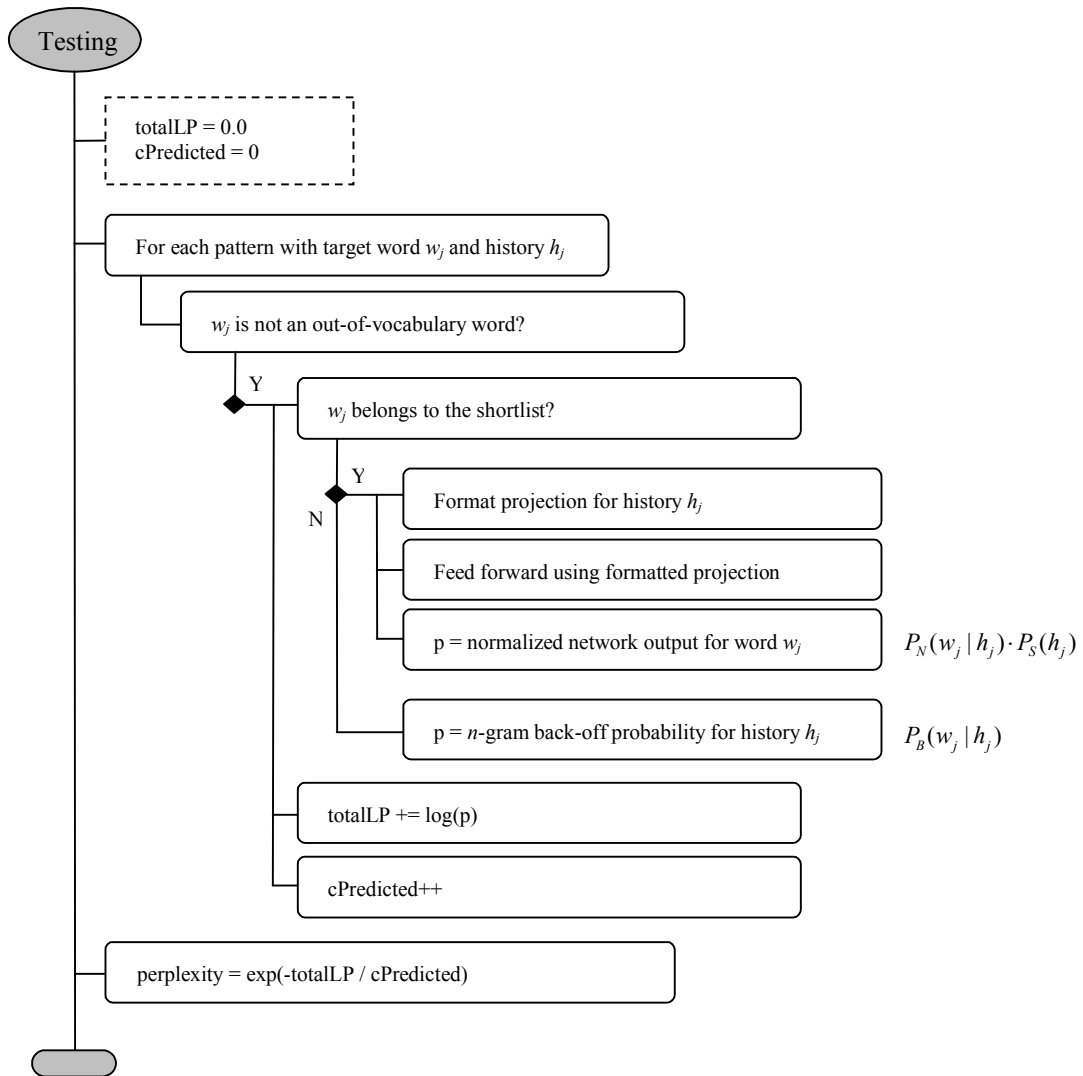


Figure 5 – Neural network testing algorithm

5 Results and Analysis

This chapter describes the experimental results and analysis that fulfil the aims and objectives of section 1.2. The source data used to generate all graphs can be found in tabular form in the `docs/` folder of the project distribution.

5.1 *N*-gram Perplexities

As a baseline for comparison with the neural network results, *n*-gram language models were built for a range of different vocabulary sizes. The *n*-gram language models were trained using the tools of the SRI-LM Language Modelling Toolkit [Stolcke 2002¹¹]. Perplexities were calculated for the full vocabulary and for the restricted sizes of 50, 100, 500, 1000, and 2000 words.

The aim of this project is to demonstrate that the continuous space representation of the neural network language model affords better generalization to unseen contexts than *n*-grams. In order to make a comparison of perplexities meaningful, the best *n*-gram model than can possibly be built was constructed from the available training data. After extracting wordlists for each target vocabulary, the wordlists were used with the training data to construct *n*-gram language models. The cut-offs were 1, 1, and 2 for bi-grams, tri-grams and 4-grams respectively.

Table 4 shows the *n*-gram perplexities calculated using models trained from the `cell11` and `swbdI` corpora.

| Vocabulary Size | Cellular 1 | | Switchboard I | |
|-----------------|-------------------------------------|--|------------------------------------|--|
| | Training <code>cell11.dat</code> | Testing <code>h5dev01_cell11.dat</code> | Training <code>swbdI.dat</code> | Testing <code>h5eval03_noexp.dat</code> |
| 50 | 30.61 | 40.10 | 45.44 | 47.42 |
| 100 | 28.44 | 40.76 | 43.15 | 46.44 |
| 500 | 23.15 | 51.09 | 35.00 | 50.94 |
| 1000 | 21.04 | 56.83 | 32.13 | 54.37 |
| 2000 | 19.45 | 62.79 | 29.96 | 60.18 |
| full | 17.47 | 73.59 | 25.22 | 79.71 |

Table 4 – Training and testing *n*-gram perplexities for Cellular 1 and Switchboard I

The results show that whilst the training perplexity is low, the perplexity when applied to unseen testing data increases significantly.

5.2 Model Parameters

This section compares the number of parameters required by the *n*-gram and neural network language models for the vocabulary sizes of the baseline experiments. The *n*-gram parameter count is simply the sum over all orders of the *n*-grams stored in the model. The neural network parameter count is the total number of weights, quoted here for a history size $h = 3$ and a fixed topology of $P=24$ and $H=48$. For such a model, the

¹¹ SRI-LM: <http://www.speech.sri.com/projects/srilm/>

hidden layer requires $N = P * h * H = 3456$ weights. The total number of network weights is a function of the number of inputs $i = v + 2$ (i.e. v + sentence-start and OOV) and number of outputs $o = v + 2$ (i.e. v + sentence-end + OOV). Given these values, the number of network weights N can be calculated using equation [5-1] below.

$$N = i * P + P * h * H + H * o \quad [5-1]$$

The parameter counts for each vocabulary size are shown in table 5 below (results are quoted for the `cell111` corpus).

| <i>Model</i> | <i>50</i> | <i>100</i> | <i>500</i> | <i>1000</i> | <i>2000</i> | <i>full</i> ¹² |
|-----------------------|-----------|------------|------------|-------------|-------------|---------------------------|
| <i>N-gram</i> | 22999 | 49740 | 128248 | 157348 | 179925 | 207352 |
| <i>Neural Network</i> | 7200 | 10800 | 39600 | 75600 | 147600 | 491328 |

Table 5 – Model parameters comparison for *n*-gram and neural network

The figures show that for smaller vocabularies, the *n*-gram representation requires more parameters but that as the vocabulary size increases, the relative number of model parameters required by the network increases faster. For the full vocabulary, the neural network requires more than twice as many parameters as the *n*-gram, caused by the very large size of the projection and output layers. Training so many neural network weights will be difficult and time consuming and larger network topologies further compound this problem. This is clear evidence in support of the need for a shortlist to make neural network language modelling a viable proposition.

¹² No full vocabulary neural network model was actually constructed due to excessive training time

5.3 Baseline Experiments

The purpose of the baseline experiments was to examine whether the neural network provides better generalization to unseen contexts. It was not possible to evaluate a network using the full vocabulary due to the excessive training time (see instead the shortlist results of section 5.5). All baseline experiments were conducted using the same fixed network size of $P=24$ and $H=48$ unless otherwise noted. Training was performed for a total of 40 epochs and OOVs were permitted in the contexts for both n -gram and neural network evaluations.

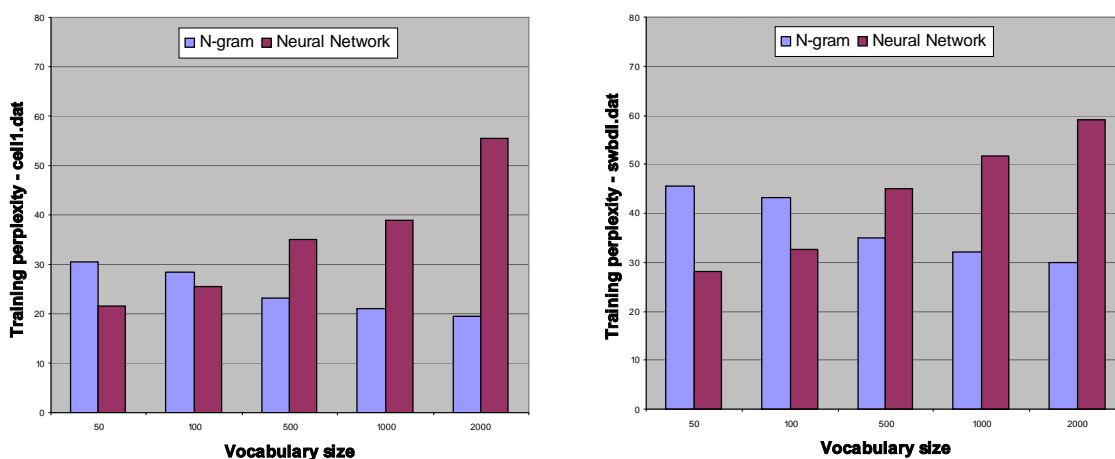


Figure 6 – Training perplexities for Cellular 1 and Switchboard I

Figure 6 compares the training perplexity calculated by the n -gram and neural network language models. For both `cell1` and `swbd1`, the n -gram obtains significantly lower perplexities and, as the size of the vocabulary increases, becomes ever better able to model the training data. This is not unexpected: the explicit nature of the n -gram language model representation allows it to capture patterns in the training data very accurately.

The opposite trend is seen in the training perplexity of the neural network. As the vocabulary size increases, the neural network perplexity rises steadily. A larger vocabulary allows the n -gram to capture more of the word relationships in the data. The neural network is also capable of learning these relationships but first requires an enlarged topology or further training. Without increasing the size of the projection and hidden layers, the neural network finds it increasingly difficult to model the larger vocabulary of the training data as accurately as the n -gram.

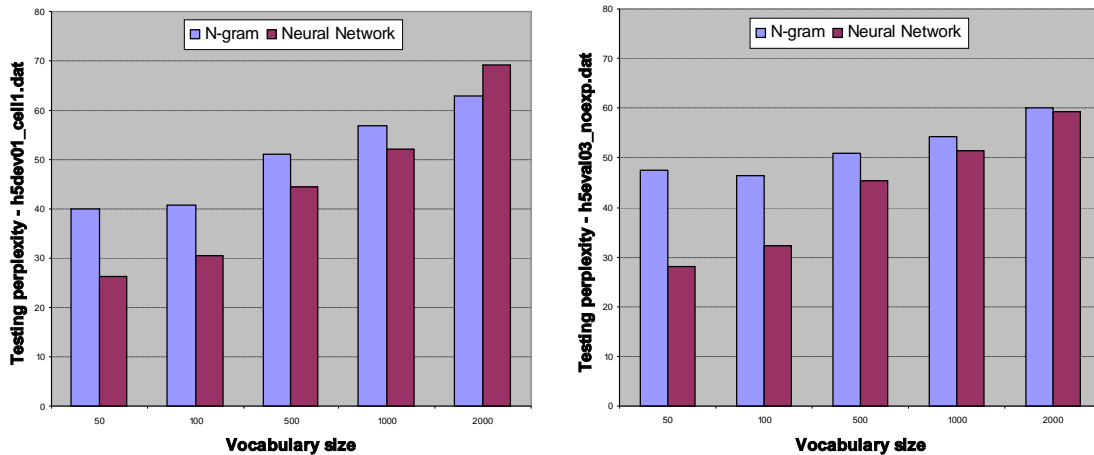


Figure 7 – Testing perplexities for Cellular 1 and Switchboard I

The testing perplexities of figure 7 show that for all vocabulary sizes except 2000 the neural network language model was successfully trained to predict words with a lower perplexity than the traditional n -gram approach. The graphs of testing perplexity indicate that whilst the network easily improves upon the perplexity of the n -gram model for small vocabulary sizes, the relative difference steadily falls as the vocabulary size increases. This fall in performance is probably due to the small fixed size of the network. A larger topology would probably enable the network to outperform the n -gram even for a vocabulary size of 2000. Section 5.4.5 describes experiments in which much larger networks were evaluated.

It is interesting to note that for the `swbdI` corpus and vocabulary sizes of 100 and 1000, the testing perplexity was observed to be marginally lower than the training perplexity. This is made clear in table 6 below.

| <i>Vocabulary Size</i> | <i>Perplexity</i> | |
|------------------------|-------------------|----------------|
| | <i>Training</i> | <i>Testing</i> |
| 100 | 32.56 | 32.36 |
| 1000 | 51.57 | 51.32 |

Table 6 – Training and testing perplexities for the Switchboard I corpus

The n -gram contexts that the network has been trained to predict well must be proportionately higher in the testing data, allowing the network to make many accurate predictions and resulting in a lower overall perplexity.

The results of these baseline experiments show clearly that the continuous nature of the neural network’s vector representation allows superior generalization to unseen testing data and results in lower perplexities than those obtainable with a well-trained back-off n -gram model of language.

5.4 Neural Network Properties

This section presents results that demonstrate the convergence and performance properties of the neural network language model.

5.4.1 Network Training Times

Table 7 compares epoch training times for the topology of the baseline experiments (i.e. n -gram order 4, $P=24$, $H=48$).

| <i>Vocabulary Size</i> | <i>Cellular 1</i> | <i>Switchboard I</i> |
|------------------------|-------------------|----------------------|
| 50 | 18 | 231 |
| 100 | 23 | 304 |
| 500 | 68 | 836 |
| 1000 | 162 | 1130 |
| 2000 | 413 | 5082 |

Table 7 – Epoch training times in seconds for Cellular 1 and Switchboard I

Although Cellular 1 training is quite manageable, a single epoch of Switchboard I using a 2000 word vocabulary takes almost 85 minutes. Given that Switchboard I is not particularly large by language modelling standards, it is easy to see that excessive training time will become a major issue as the corpus and network size increase. The most promising way of improving the speed of training is the matrix bunch-mode updates of [Schwenk 2004c].

5.4.2 Initialization Experiments

The purpose of this experiment was to examine whether the use of randomly initialized network weights affects the rate of convergence or final perplexity. Figure 8 on the following page plots the training perplexity for a network of size $P=24$ and $H=48$ over 24 epochs for eight distinct training sessions using `cell1` and a vocabulary size of 1000. The configuration parameters and learning rates used for each session were precisely identical. Only the initial network weights (set to a random value $-0.2 \leq w \leq 0.2$) were varied. Full logs can be found in the `results/initialize/` folder of the project distribution.

The graph shows that for the first 16 epochs the training perplexity of all eight training sessions was close to uniform. After epoch 16, however, the effect of random initialization becomes apparent. The step-like decrease in perplexity from 49 to 46 corresponds to a change in the learning rate that enables exploration of the error space at a finer resolution. The graph shows clearly that whilst the first session to reach this point occurs at epoch 16, the last does not occur until epoch 19. It took four additional epochs for the slowest of the sessions to catch up. This shows that random network initialization does affect the rate of convergence. For very large data-sets and network topologies, these additional epochs could take a long time.

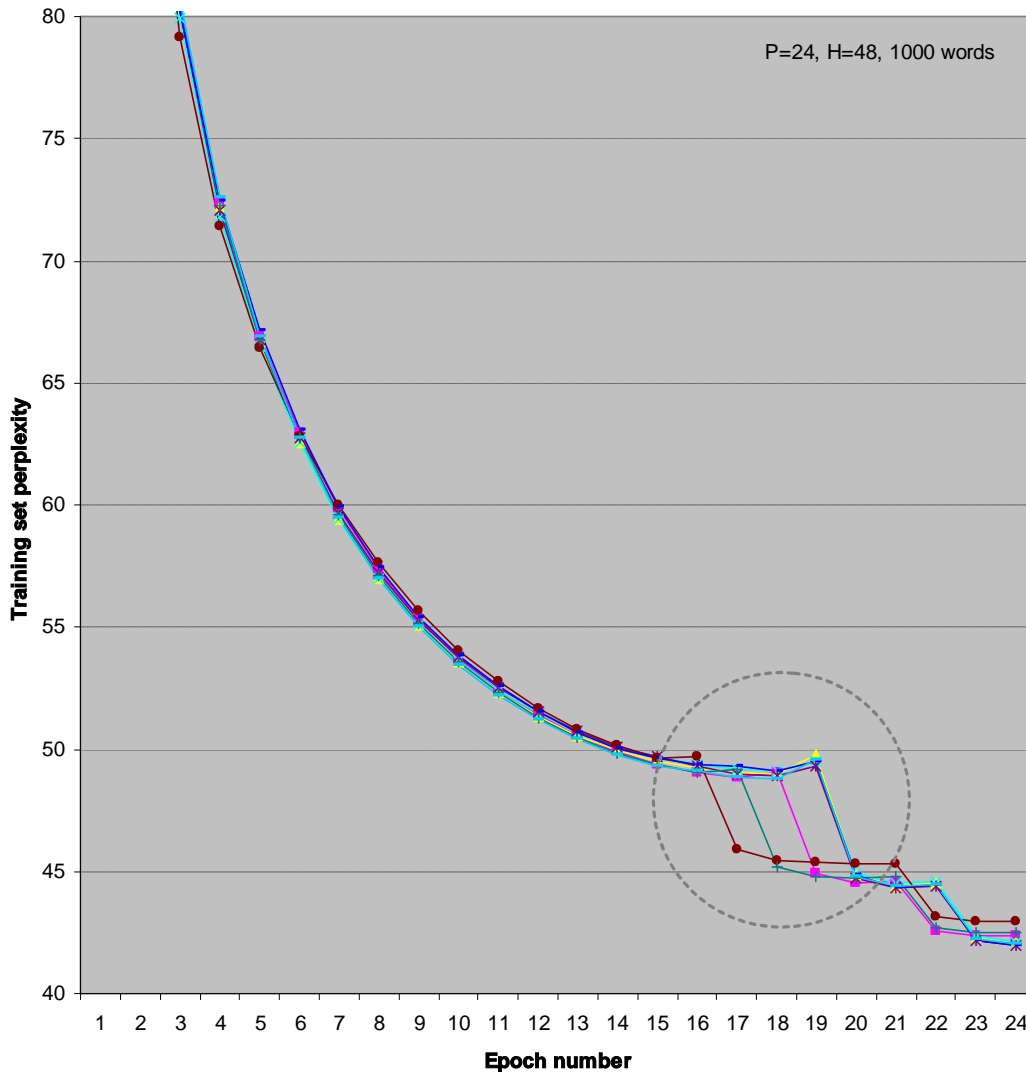


Figure 8 – Initialization and training data convergence for Cellular 1

The results show that all of the randomly initialized training sessions converged to the same minimum. This is reassuring as it demonstrates that random initialization of weights does not adversely impact the ability of the network to learn the patterns of the training data. The very large size of the data-set ensures that any initial variations are smoothed away as the weights approach their equilibrium values.

One possibility to improve the rate of convergence and to ensure uniformity of learning (i.e. that all weights reach their equilibrium values at approximately the same time) is to use an initialization scheme such as that proposed in [Duda, Hart & Stork 2000] where the initial values of the weights are layer specific and a function of the number of connected inputs. The problem of non-uniform learning is particularly acute for the reduced vocabularies examined in this project where there is a huge preponderance of

one particular category: the OOV symbol. Time constraints meant that it was not possible to examine this style of initialization.

5.4.3 Over-fitting Experiments

This section examines whether or not over-fitting occurs when training for a large number of epochs. The experiment was performed using a network size of $P=24$ and $H=48$ with the `swbdI` data and a 1000 word vocabulary trained for 40 epochs. Figure 9 below plots the training and testing perplexities evaluated on an epoch-by-epoch basis (full log files and partially trained networks can be found in the `results/overfit` folder of the project distribution).

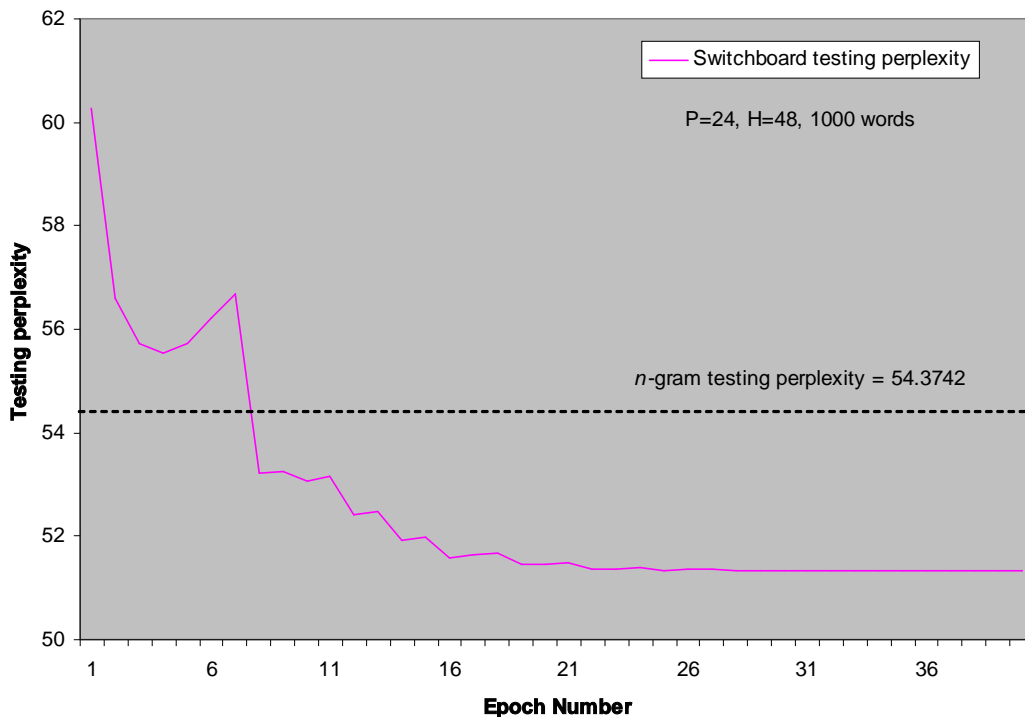


Figure 9 – Testing perplexity for Switchboard I evaluated over 40 epochs

For the above described experimental conditions, the smooth and flat testing perplexity minimum indicates that no over-fitting occurred. The temporary peak and sudden fall between epochs five and seven is a result of too large a learning rate. The absence of over-fitting suggests two possibilities: either training has reached a local minimum, or the network topology is not sufficiently complex to model the patterns of the training data. The experimental results of section 5.4.5 indicate that for larger network topologies, over-fitting may indeed occur.

5.4.4 N-gram History Length Experiments

This section examines the impact on perplexity of varying the n -gram order. For the neural network language model, varying the n -gram order is analogous to varying the number of words from which the projection vector is composed. As the n -gram order

increases, the network should be capable of learning longer distance dependencies between the words and thus obtaining a reduced perplexity. The experiments were conducted for a fixed network topology of $P=48$ and $H=64$ using the `cell1` data and a vocabulary size of 1000 words. Full logs and results can be found in the `results/history/` folder of the project distribution. Figure 10 below shows the training perplexity over 24 epochs for n -gram orders $2 \leq n \leq 6$.

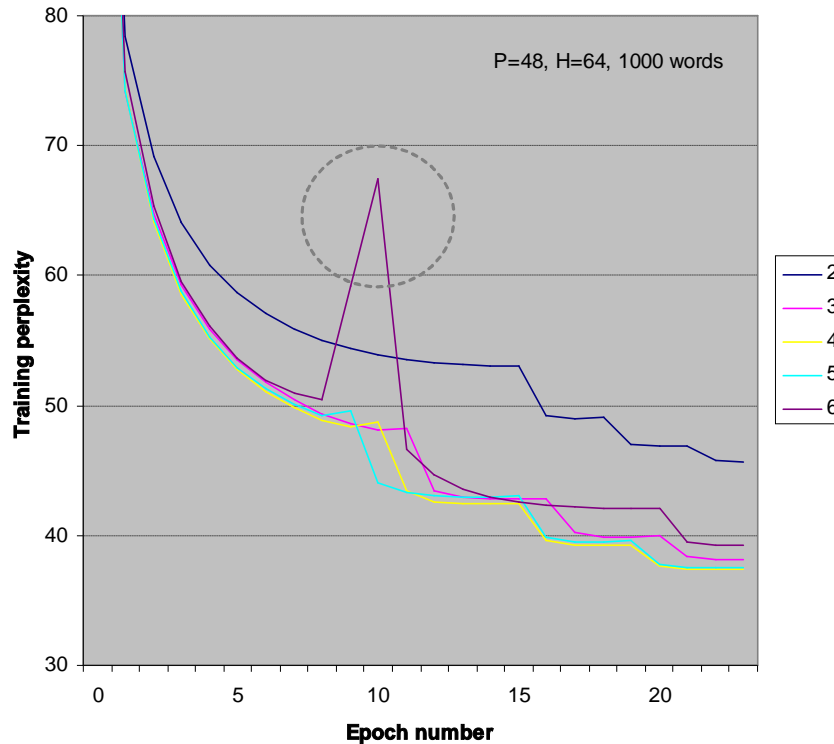


Figure 10 – Training perplexity by n -gram order for Cellular 1

The graph clearly shows that as the n -gram order increases from 2 to 5 the perplexity decreases. It is also clear that most of the gain is observed when moving from bi-grams to tri-grams. A further small improvement is observed when 4-grams are used, but there is no significant advantage to be gained using n -gram orders greater than 4. These results suggest that the most appropriate n -gram order to use for the neural network language model is 4.

The spike observed in the n -gram order 6 line of figure 10 is an example of a situation in which the learning rate grew too large and resulted in an overall increase in the error function for the epoch. The training algorithm halves the learning rate before resuming training. The results show that for this experiment gradient descent was able to continue improving the training perplexity, although after the fixed number of 24 epochs the perplexity is still worse than that for order 5. The network topology remained fixed across all n -gram orders and it is likely that the increased length and complexity of the patterns of network inputs require a corresponding increase in the hidden layer dimensionality before the training patterns can be accurately learned.

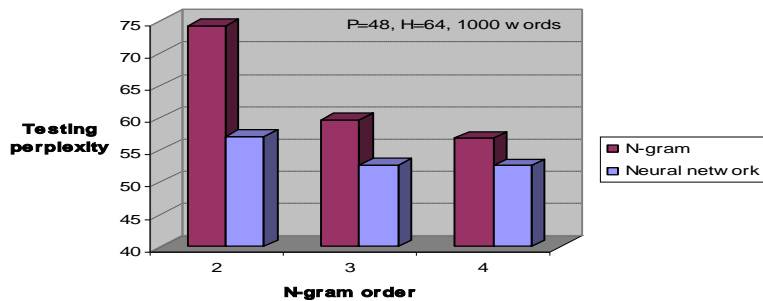


Figure 11 – Testing perplexity by n -gram order for Cellular 1

Figure 11 shows that a similar pattern is observed when neural network models of order 2, 3, and 4 are applied to the testing data `h5dev01_cell11.dat`. The perplexities obtained using the n -gram model are shown for comparison and it is clear that for both types of language model the greatest gain is in moving from bi-grams to tri-grams.

5.4.5 Topology Experiments

The experiments described here are designed to evaluate if larger network topologies can obtain lower perplexities. The experiments were performed using a range of different hidden layer sizes and two different projection layer sizes. The networks were trained using the `cell1` data and a 1000 word vocabulary. Figure 12 below shows the training perplexity calculated over 60 epochs for $P=24$ with a range of different hidden layer dimensionalities.

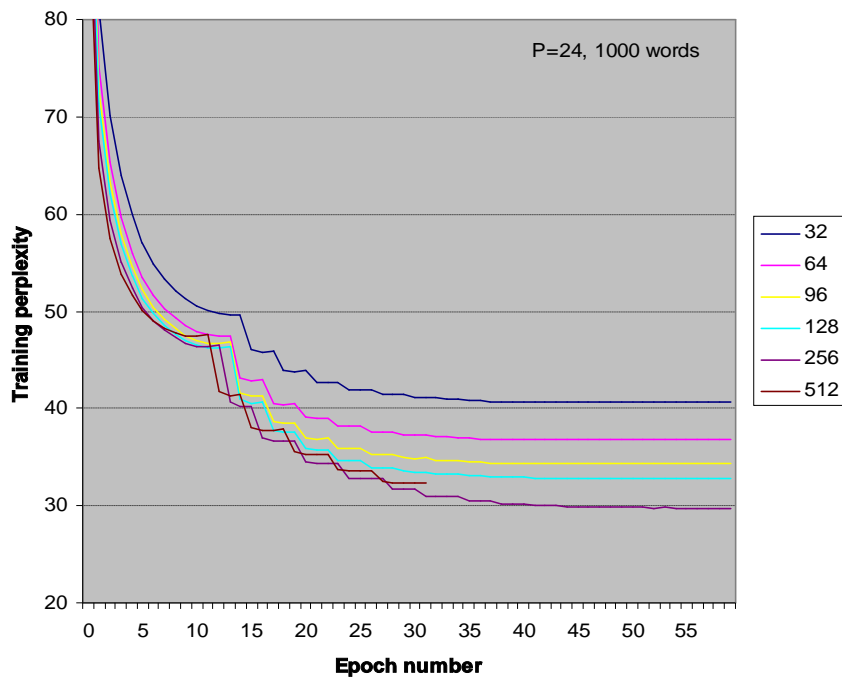


Figure 12 – Network topology and convergence by hidden layer size when $P = 24$

The results show that for a fixed projection layer size, the training perplexity decreases as the number of hidden layer neurons is increased. The increased dimensionality of the hidden layer allows a wider range of patterns to be modelled by the weights. The lowest perplexity was obtained with a hidden layer size of $H=256$. There was insufficient time for the $H=512$ experiment to run to completion but the graph suggests that the final perplexity would not be lower than that for $H=256$. Figure 13 below shows results for a similar experiment in which the dimensionality of the projection layer is increased to $P=48$.

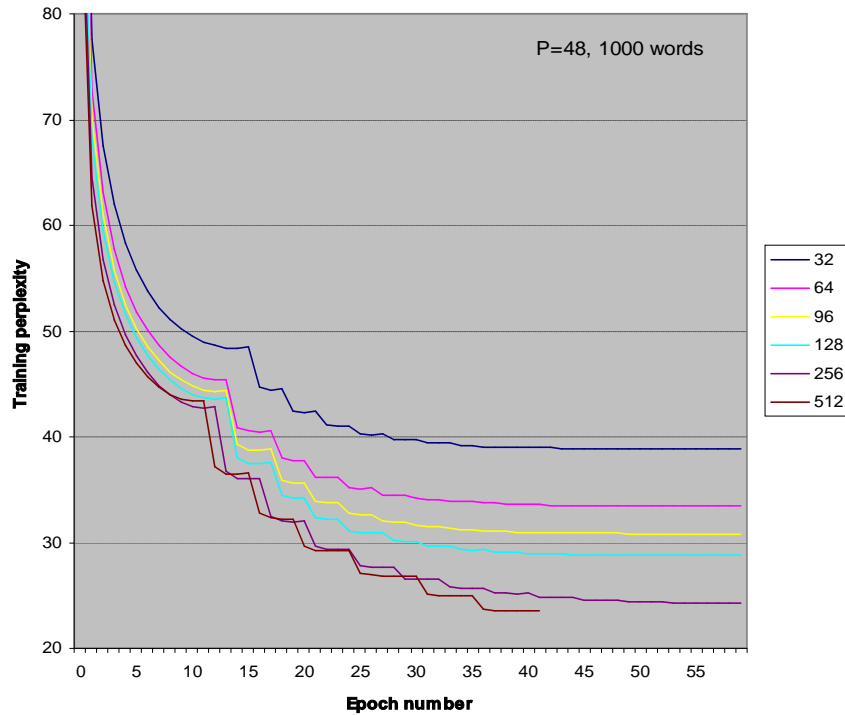


Figure 13 – Network topology and convergence by hidden layer size when $P = 48$

The graph shows that the increased size of the projection layer allows a smoother representation of the input vector. This results in overall improved training perplexities for all hidden layer dimensionalities. Comparing the two graphs shows that increasing P contributes only a small improvement when $H=32$ or 64 , a medium improvement when $H=96$ or 128 and a fairly large improvement when $H=256$ or 512 . Again, there was insufficient time to process the full experiment for the very large hidden layer of $H=512$. The perplexity for $P=48$ and $H=512$ is almost 6% lower than the corresponding perplexity for $P=24$. These figures are also much better than the baseline results of section 5.3. However, as figure 14 shows below, the significantly improved training perplexity does not extend to the testing data.

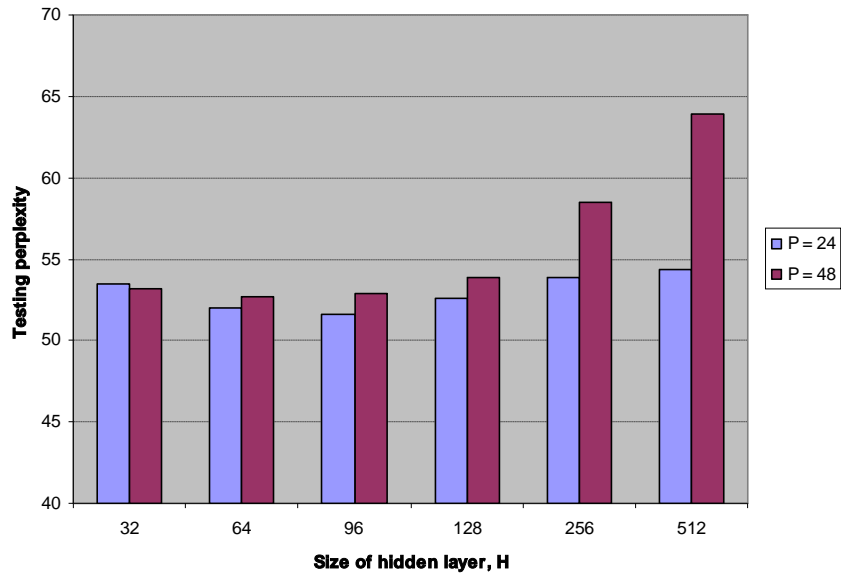


Figure 14 – Testing data perplexity by size of hidden layer for Cellular 1

The figure shows that although increasing the size of network layers leads to impressive reductions in the training data perplexity, the perplexity of the testing data does not improve. In fact, for the very large hidden layer sizes of 256 and 512 the testing perplexity degrades significantly. The poor performance on testing data suggests there may be a problem with over-fitting.

5.5 Full Model Experiments – The Shortlist

This section describes the results obtained using the shortlist approach of section 2.4. Unfortunately, it was not possible to obtain perplexities lower than the n -gram models despite extensive experimentation. There are two possible reasons for this inability to match the improved generalisation shown by [Schwenk & Gauvain 2004c]. Firstly, it is possible that the correct combination of topology and learning rate was not identified and this resulted in badly trained networks. The second possibility is that there is an algorithmic flaw in, for example, the handling of OOV symbols.

Table 8 shows perplexities obtained for `cell1` using shortlist sizes of 1000 and 2000 words. The size of the projection layer was $P=48$ and training proceeded for 24 epochs.

| Shortlist Size | $\% P_N(w_j h_j)$ | $\% P_B(w_j h_j)$ | Hidden Layer Size | Training Perplexity | Testing Perplexity |
|----------------|---------------------|---------------------|-------------------|---------------------|--------------------|
| 1000 | 89.77 | 10.23 | 64 | 31.28 | 100.95 |
| | | | 128 | 26.95 | 94.49 |
| | | | 256 | 24.48 | 98.60 |
| 2000 | 92.13 | 7.87 | 64 | 36.17 | 87.46 |
| | | | 128 | 32.40 | 91.99 |
| | | | 256 | 34.64 | 95.39 |

Table 8 – Shortlist training and testing perplexities for Cellular 1

The lack of a discernable trend as the hidden layer size increases suggests there may be a training problem and that the configuration options are such that gradient descent did not successfully find a good minimum in the error space. Further tests were performed using hidden layer sizes of 400, 500 and 600 but the testing perplexity improvements were marginal. Complete log files and resultant networks are available in the `results/shortlist` folder.

Table 9 shows results obtained using `swbdI` and a shortlist of size 1000. Excessive training time allowed for only four epochs to be processed in 24 hours. The logs suggest these models were not fully trained to convergence.

| $\% P_N(w_j h_j)$ | $\% P_B(w_j h_j)$ | <i>Hidden Layer Size</i> | <i>Training Perplexity</i> | <i>Testing Perplexity</i> |
|---------------------|---------------------|--------------------------|----------------------------|---------------------------|
| 90.29 | 9.71 | 64 | 48.84 | 87.96 |
| | | 128 | 46.81 | 85.92 |

Table 9 – Shortlist training and testing perplexities for Switchboard I

These results are more promising – they are much closer to the full vocabulary target n -gram performance of 79.71. With further training, and a larger network topology, it may be possible to improve upon the n -gram perplexity scores.

The main difference between the shortlist approach and the restricted vocabulary case is in the handling of OOVs. The restricted vocabulary allows OOV as an input and also has OOV as a network output. The shortlist includes OOV in the inputs but does not include an OOV output. One possible problem is that when the full vocabulary is used there are no OOV words in any of the n -gram patterns presented to the network during training. As a result, the projection matrix contains randomly initialized weights for these values. During testing, if OOVs are allowed in the context then these untrained weights contribute random noise to the outputs and this significantly degrades performance.

Two strategies are available for dealing with this problem. The first is to train using a subset of the vocabulary – e.g. all words that occur more than once. This ensures the weights for the OOV input to the projection matrix are trained. The second possibility is to evaluate perplexity using only contexts that do not contain an OOV symbol (similar to the `-skipooovs` option of the SRI-LM tool `ngram`). Both of these strategies were tested but neither helped to improve performance.

6 Class-based Neural Network Language Models

This section describes the main project extension: a connectionist approach to class-based language models.

6.1 Background and Theory

A common technique for dealing with the n -gram issue of data sparsity is to reduce the number of history equivalence classes by grouping words. The probability of a word given a particular history is reformulated in terms of the class n -gram probabilities and the prior class conditional unigram probabilities. If each word is assigned to a single class, the probability of word w_j given history $h_j = w_{j-n+1}, w_{j-n+2}, \dots, w_{j-1}$ can be expressed as:

$$P(w_j | w_{j-n+1}, w_{j-n+2}, \dots, w_{j-1}) = P(c_j | c_{j-n+1}, c_{j-n+2}, \dots, c_{j-1}) \cdot P(w_j | c_j) \quad [8.1]$$

The main advantage of such a class-based model is that if relatively few classes are used then the number of word histories that map to each possible class history is greatly increased and there are far fewer missing contexts for a given corpus size. Ameliorating the data sparsity issue in this manner enables the use of longer n -gram contexts. The class n -gram probabilities and class conditional unigram probabilities are estimated from counts in a standard fashion [Manning & Schütze 1999]. The assignment of words to classes can be achieved using an automatic agglomerative clustering process that maximizes the training corpus log likelihood [Jurafsky and Martin 2000].

There are two possible ways of using classes in a neural network language model:

1. The first involves mapping the input words to classes and training directly on the word outputs of the network. This approach requires minimal changes to the existing architecture – all that is required is a scheme for mapping the history words of each training pattern to an appropriate class index and then using these class indices to format a projection vector of suitable size. However, this approach is only really feasible for a greatly restricted vocabulary. To build a class-based neural network that supported the full `swbd1` vocabulary, for example, would require 26728 output neurons. Efficiently training such a network will be very difficult.
2. The second possibility is to train the network to minimize errors in classifying classes at the outputs. In this approach, input words are mapped to classes and the network has a single output for each possible class. The target word for each training pattern is also mapped to a class and this mapped class is used to minimize the cross entropy error function. To compute perplexity, the network output corresponding to the target class is multiplied by the class conditional unigram probability, the second term on the right hand side of equation [8.1]. The class memberships and unigram probabilities can be pre-calculated using language modelling tools and stored in memory. This is the approach examined in the current project.

The baseline results of section 5.3 showed that as the number of inputs and outputs increases it is harder for the neural network to achieve competitive testing perplexities. The main advantage of using classes instead of words is that the number of inputs to the projection layer is significantly decreased and this allows the network to more accurately capture the parameters of the training data. It is also possible to train networks with longer n -gram histories.

6.2 Class-based N -gram Baseline Models

In order to provide a meaningful comparison with the perplexities calculated by the neural network language model it is first necessary to create n -gram class-based language models. These models were created using the HTK¹³ Language Modelling tools [Young et al 2002]. The models were constructed using automatic clustering to group the vocabulary words into classes on the basis of their bi-gram statistics. These class memberships and unigram class probabilities were used to create class-based language models and perplexities evaluated for the training and testing data-sets.

A script to produce a class-based language model by specifying a vocabulary, the required number of classes, and a training data file can be found in the `lm/classes/` folder of the project source tree. The n -gram cut-offs for bi-grams, tri-grams and 4-grams were all set to 1. Clustering was performed for six iterations¹⁴ although perplexity was not significantly decreased by the additional iterations. Since the start of sentence, end of sentence, and unknown word tokens have a special interpretation it is conventional to assign each to its own single-word class.

Table 10 below shows two class assignments produced when `Cluster` was used to create 100 classes from the `cell1` training data restricted to a vocabulary of 2000 words.

| <i>Class 28</i> | | | <i>Class 20</i> | | |
|-----------------|---------|-----------|-----------------|---------|-----------|
| JUST | CLASS28 | -0.469009 | YEAH | CLASS20 | -0.374551 |
| ACTUALLY | CLASS28 | -1.536308 | HM | CLASS20 | -3.354867 |
| RECENTLY | CLASS28 | -4.569336 | YES | CLASS20 | -3.235427 |
| CERTAINLY | CLASS28 | -4.743689 | UH_HUH | CLASS20 | -2.535679 |
| PERSONALLY | CLASS28 | -4.792479 | MHM | CLASS20 | -1.962064 |
| ALSO | CLASS28 | -3.078681 | YEP | CLASS20 | -4.851509 |
| USUALLY | CLASS28 | -3.310875 | YUP | CLASS20 | -4.818719 |
| ALREADY | CLASS28 | -3.799228 | EW | CLASS20 | -6.001415 |
| TOTALLY | CLASS28 | -4.004022 | | | |
| MOSTLY | CLASS28 | -4.743689 | | | |

Table 10 – Class memberships assigned by the HLM tool Cluster

In order to implement a neural network class-based language model, explicit definitions of class membership and class conditional unigram probabilities are required. This information is encoded by HLM in the word-class file produced by the `Cluster` tool. The file contains an initial header followed by one class membership definition per line. Each definition consists of three pieces of information: the word, the class to which it belongs, and the log probability of that word given the class. This file can be parsed by

¹³ HTK: <http://htk.eng.cam.ac.uk/>

¹⁴ Just one iteration of clustering was performed for the full vocabulary class-based models

the neural network tool and appropriate data structures constructed to support a class interpretation of the training and testing data. The listing below shows a sample of the output produced by Cluster:

| | | |
|-------|---------|-----------|
| !!UNK | CLASS3 | 0.000000 |
| IT | CLASS5 | -0.032110 |
| IS | CLASS6 | -0.011964 |
| GOOD | CLASS7 | -1.470477 |
| FOR | CLASS8 | -0.030501 |
| </s> | CLASS2 | 0.000000 |
| SO | CLASS9 | -0.022439 |
| YOU | CLASS10 | 0.000000 |

Figure 15 – Class membership and probabilities output by the HLM Cluster tool

The next section describes the data structures and architecture of the class-based neural network language model.

6.3 Class-based Neural Network Models

Figure 16 on the following page shows the architecture of the class-based neural network language model. The training or testing data file is read from disk and stored as an array of unique integer word identifiers in exactly the same manner as the word-based network implementation. A new method maps the word identifiers in the window to their corresponding class identifiers and also returns the unigram probability of the head of the training pattern. The class identifiers are used as the inputs to the network and the outputs are trained by cross entropy to compute the probability of the target class given the class history.

The data structures required to support the architecture described above are shown in code sample 10. The `wordlist` and `classlist` maps are used to convert the text training or testing file to an internal representation using unique integer identifiers. The `contexts` data member stores the training or testing data file contents. A sliding window of size equal to the n -gram order moves along this array. The identifiers of the words in the window are used as indices into the `expansions` member which contains the class mappings and unigram probabilities for each word read from the data file. The advantage of this scheme is that almost all processing is done at the time the file is loaded, rather than during training. Mapping each word to its class and probability resolves to simple table look-up operation.

The training of the network is almost identical to the word-based version – the only difference being that mapped class identifiers are used to format the projection vector during feed forward and to specify the target output for back propagation. Only the class identifier corresponding to the n -gram head is required to train the cross entropy error criterion – the unigram probability is not used during training. This ensures that the network is trained to predict classes from histories entirely independently of the class conditional word probabilities. The unigram probabilities are used during testing: they are required to calculate the perplexity of the data-set from the average log probability of equation [8.1].

6.4 Class-based Results and Analysis

This section compares the performance of the n -gram and neural network approaches to class-based language models.

6.4.1 N-gram Results

The n -gram testing perplexities for `cell1` evaluated using the HTK tool `LPerplex` are shown in table 11 below (the OOV column shows the perplexity when OOVs are permitted in the n -gram context, i.e. the `-u` option in `LPerplex`).

| <i>Vocabulary Size</i> | <i>Testing Perplexity</i> | | |
|----------------------------|---------------------------|---------------|------------|
| | <i>Classes</i> | <i>No OOV</i> | <i>OOV</i> |
| 1000 | 50 | 66.27 | 71.93 |
| | 100 | 57.53 | 62.61 |
| 2000 | 50 | 78.59 | 82.61 |
| | 100 | 68.13 | 71.93 |
| | 150 | 67.03 | 70.86 |
| Full | 100 | 89.48 | n/a |
| | 200 | 83.50 | n/a |
| | 500 | 81.30 | n/a |

Table 11 – Testing perplexity for class-based n -gram models using Cellular 1

Comparing these results with the word-based n -gram language models of section 5.1, it is immediately apparent that across all vocabulary and class sizes the class-based perplexities are inferior. This is unsurprising – mapping specific words to general classes blurs some of the syntactic and semantic relations between the words of the training data. To effectively leverage a class-based model, it is usual practice to build an interpolated model that computes a weighted sum of word- and class-based probabilities. Perplexity calculated using such an interpolated model typically improves upon the words-only model.

6.4.2 Neural Network Results

The neural network class-based models were evaluated over the same range of vocabulary and class sizes. The model topology of $P=24$ and $H=48$ was kept constant throughout these tests. The results are shown in table 12. Full experiment logs and trained networks can be found in the `results/classes` folder of the project distribution.

| <i>Vocabulary Size</i> | <i>Testing Perplexity</i> | | |
|------------------------|---------------------------|---------------|------------|
| | <i>Classes</i> | <i>No OOV</i> | <i>OOV</i> |
| 1000 | 50 | 65.62 | 70.50 |
| | 100 | 56.37 | 61.03 |
| 2000 | 50 | 77.05 | 80.57 |
| | 100 | 66.68 | 70.28 |
| | 150 | 63.61 | 67.22 |
| Full | 100 | 91.82 | n/a |
| | 200 | 84.73 | n/a |
| | 500 | 79.65 | n/a |

Table 12 – Testing perplexity for class-based neural network models using Cellular 1

The results show that although the neural network consistently achieves better perplexities than the n -gram, the improvements are small. The greatest difference was for a 2000 word vocabulary with 150 classes in which the perplexity of the neural network has improved by approximately 5%. These results suggest that even when the n -gram has the advantage of far fewer unseen contexts, the neural network is still able to obtain better generalisation through the use of a continuous vector space to represent the class histories. The reason why the perplexity improvement is smaller for class-based than word-based models is that the main limitation of n -grams (i.e. the huge preponderance of unseen contexts caused by data sparsity) is greatly ameliorated when using classes.

The small size of the `cell1` training corpus helps the neural network to obtain better relative perplexities. When these experiments are repeated using the `swbd1` corpus, the relative performance of the n -gram increases to the point at which it is very difficult for the neural network to match. The very large number of training examples and reduced set of history equivalence classes improve the n -gram class-based model’s performance on testing data. This is confirmed by the `swbd1` results of table 13.

| <i>Vocabulary Size</i> | <i>Classes</i> | <i>N-gram</i> | <i>Neural Network</i> |
|------------------------|----------------|---------------|-----------------------|
| 1000 | 50 | 73.66 | 76.46 |
| | 100 | 59.38 | 63.93 |
| 2000 | 50 | 88.03 | 91.14 |
| | 100 | 71.41 | 76.51 |
| | 150 | 65.41 | 70.53 |
| Full | 100 | 106.64 | 115.79 |
| | 200 | 92.61 | 101.98 |
| | 500 | 85.08 | 92.45 |

Table 13 – Testing perplexity for class-based models using Switchboard I

The results of these experiments show that it is only when training data is limited that the neural network approach to class-based language models offers improved performance over that of n -grams.

7 Conclusions and Future Work

This section summarizes the conclusions of the project and suggests directions for future work.

7.1 Conclusions

This thesis described the implementation and evaluation of a large scale neural network language model. The neural network models were successfully demonstrated to achieve improved perplexity over the traditional n -gram approach for restricted vocabularies. The improved estimation of language model probabilities is enabled by the smooth interpolation of a continuous vector space representation of each n -gram context. The results show that for a domain such as telephone conversation transcriptions, in which there is limited training data, a neural network approach is better able to generalise the estimation of language model probabilities.

The results also show that an efficiently optimized training scheme is vital. Whilst an n -gram model for even a large corpus can be trained in minutes, much more time is required to train a neural network language model to convergence. This issue made extensive experimentation with the larger Switchboard I corpus problematic, and further refinements to the training procedure are clearly in need (see below for suggestions).

The results of the shortlist evaluation were unfortunately unable to demonstrate the improved perplexity of [Schwenk 2004c] despite extensive experimentation with varying topologies and system parameters. With additional time, and a review of the handling of OOV words during training and testing, it should be possible to improve the shortlist perplexity.

The evaluation of the class-based neural network language model shows that when there is limited training data, the neural network can again achieve better perplexity than a class-based n -gram. However, the results also demonstrate that as the amount of training data increases, the n -gram reduction in the number of equivalence classes compensates for the discrete nature of the word indices, allowing the n -gram to supersede the performance of the neural network.

7.2 Future Work

There are a number of interesting directions in which to build upon the work of this project.

The most interesting and useful extension would be to apply the trained neural network to the task of lattice rescoring or to perform a full integration with an ASR system. This would allow an evaluation of to what extent the reduced perplexities achieved by the neural network truly enable a reduction in the word error rate, the best test of language model quality.

The problem of excessive training time could be addressed by an implementation of the matrix bunch-mode described in [Schwenk 2004c]. This would allow a full BLAS

implementation of all stages of neural network training and confer a significant reduction in epoch training time.

It may be possible to obtain faster convergence by incorporating the Quickprop alternative to the variable learning rate [Duda, Hart & Stork 2000] or some other 2nd order gradient descent scheme. If the network can be trained in fewer epochs then this will significantly reduce the time required for training networks with large corpora.

A better network initialization – such as the layer-specific scheme described by [Duda, Hart & Stork 2000] – may allow for more uniform learning and a less biased exploration of the error surface.

There may be idiosyncrasies in the training data that affect which areas of the error space are explored during gradient descent. This problem could be resolved by randomizing the order in which the training patterns of an epoch are presented to the network. A utility to randomize the lines of a specified file was written and included in the project distribution but there was insufficient time to assess if such randomization led to a reduction in perplexity.

Appendix A – Program Operation

The source code conforms to ANSI C++ and has been successfully built and tested on Linux and Windows. Table 14 summarizes the command-line options that can be used to control program execution¹⁵.

| <i>Switch</i> | <i>Required?</i> | <i>Description</i> |
|---------------|------------------|--|
| -config | Y | Configuration file pathname. |
| -vocab | Y | Vocabulary file pathname. |
| -data | Y | Training or testing data file pathname. |
| -classes | Y | Word-class definitions pathname (required only for class-based evaluations). |
| -test | N | Configures testing mode instead of the training mode default. |
| -sizev | N | Overrides vocabulary size in the configuration file. |
| -sizes | N | Overrides the shortlist size specified in the configuration file. |
| -output | N | Folder in which to store partially trained networks |
| -network | N | In training mode, name to use for the final trained network. In testing mode, name of previously trained network to load. |
| -p | N | Overrides projection layer size in configuration file |
| -h | N | Overrides hidden layer size in configuration file. |
| -n | N | Overrides the n-gram order specified in configuration file. |
| -e | N | Overrides number of epochs specified in configuration file. |
| -ps | N | Shortlist normalization probability stream pathname – only valid if a shortlist size > 0 specified. |
| -pb | N | N-gram probability stream pathname – only valid if a shortlist size > 0 specified. |

Table 14 – Neural network tool command-line options

The following example shows how to start a training session from the command-line (this Linux example assumes the command is issued from the `project/` folder):

```
> NNTool.out -config config/network.cfg -vocab data/wlists/cell1.2000.txt \  
-data data/train/cell1.dat -network final-network.dat
```

The log will be written to standard output and the partial results and final network will be saved in the default `output/` folder. The fully trained network will be named `final-network.dat`. When training completes, the network can be evaluated using the following command-line:

```
> NNTool.out -config config/network.cfg -vocab data/wlists/cell1.2000.txt \  
-data data/test/h5dev01_cell1.dat -test -network output/final-network.dat
```

The results of the evaluation will be logged to standard output.

¹⁵ Please see the user manual in the `docs/` folder for full program operation details

Appendix B – Program Configuration

Table 15 lists the parameters used to configure training and testing. They should be specified in a text configuration file and may appear in any order. Each parameter should appear on a separate line with the parameter name and value separated by whitespace. The table shows the parameter name, the default value if left unspecified, and the command-line switch that may be used as an override where applicable.

| <i>Parameter</i> | <i>Default</i> | <i>Override</i> | <i>Description</i> |
|--------------------|----------------|-----------------|--|
| neurons_p | 16 | -p | The number of projection layer neurons. |
| neurons_h | 24 | -h | The number of hidden layer neurons. |
| ngram_size | 4 | -n | The size of n-gram to use for formatting history projections. |
| training_mode | true | -test | Program mode: training or testing. |
| vocabulary_size | 0 | -sizev | The size of vocabulary to use. A value of 0 indicates all words in the supplied wordlist should be included in the vocabulary. |
| shortlist_size | 0 | -sizes | The size of the shortlist to use. A value of 0 indicates that no shortlist should be used. |
| initial_min | -0.2 | | The initial minimum value for the weights and biases. |
| initial_max | 0.2 | | The initial maximum value for the weights and biases. |
| learning_rate | 0.0005 | | The learning rate used to compute changes in the weights and biases during gradient descent |
| momentum_alpha | 0.75 | | The momentum scaling constant to use during gradient descent. A value of 0 indicates no momentum term should be included. |
| weight_decay | 1.0 | | The weight decay regularisation constant. A value of 1.0 indicates that no weight decay should be applied. |
| batch_size_percent | 1 | | The percentage size of the data-set that should be used for batched updates. A value of 0 indicates that the batch size is a specific number defined by the following parameter. |
| batch_size_fixed | 2000 | | The fixed batch size to use for batched updates. This value is only used if the above parameter is set to 0. |
| max_epochs | 20 | -e | The maximum number of epochs to process during network training. This |

| | | | |
|-----------------|------------|--|---|
| | | | value is ignored during testing. |
| debug_level | 1 | | The tracing level to use. |
| debug_precision | 2 | | Precision of floating point output in debug mode. |
| lm_symbol_start | <s> | | Orthography of sentence start symbol. |
| lm_symbol_end | </s> | | Orthography of sentence end symbol. |
| lm_symbol_oov | !!UNK | | Orthography of unknown word symbol. |
| layer_name_p | projection | | Projection layer name – for debugging. |
| layer_name_h | hidden | | Hidden layer name – for debugging. |
| layer_name_o | output | | Output layer name – for debugging. |

Table 15 – Neural network language model configuration file settings

Sample configuration files can be found in the `config/` folder of the project distribution.

Appendix C – Program Output

Figure 17 on the following page shows sample output from a log file produced during a neural network training session.

The program first reports configuration options specified in the configuration file and those passed on the command-line. The program next shows the source of the wordlist and data file that are to be loaded from disk. After loading the required data, a data-set summary is printed showing the number of sentences, number of words, average number of words per sentence, and frequency of OOV words.

The next portion of the output shows the network topology – this is defined by the sizes of the projection and hidden layers, the size of the n -gram context, and the total number of words in the vocabulary.

The program then logs each epoch's training results. These results include the epoch number, the perplexity (PPL), the perplexity excluding sentence end tokens (PPL1), the epoch processing time, and the modified value of the learning rate. The time at which training started and ended are also recorded.

In addition to the log file, partial results are produced at the end of each epoch. The partially trained network is written to the disk location specified by the `-output` command-line switch. These files may be used to resume training at a later time. The filename uses a special format that indicates the provenance of the partial network:

```
nn-v1000-p24-h32-o1002-e3.dat
```

This network was produced using a 1000 word vocabulary, 24 projection layer neurons, 32 hidden layer neurons, and has 1002 outputs. It is the 3rd epoch of training. The files are written in binary format but may be viewed as text using the `NNViewNetwork.out` utility provided with the project distribution.

When training completes, the final network is written to disk in the same location as the partial results. The name of the network is specified by the `-network` command-line switch (see Appendix A for a complete description of the command-line switches).

```

NNLM Tool Configuration:

Configuration      = config/network.cfg
Vocabulary         = data/wlists/cell1.50.txt
Datafile          = data/train/cell1_2000.dat
Output folder     = nnlm-output
Network name      = nnlm-network.dat

Mode              = TRAIN

N-gram size       = 4
Vocabulary size   = 50
Shortlist size    = 0
Initial min       = -0.2
Initial max       = +0.2
Learning rate     = 0.0005
Momentum alpha    = 0.95
Weight decay      = 1
Quick prop        = 0
Batch size %      = 0
Batch size N      = 2000
Maximum epochs    = 8
Debug level       = 1
Debug precision   = 2

LM symbol START   = <s>
LM symbol END     = </s>
LM symbol OOV    = !!UNK

Reading wordlist file: data/wlists/cell1.50.txt DONE
Reading data-set file: data/train/cell1_2000.dat DONE

NNLM Dataset Summary:

Vocabulary size = 53 (inclusive of special symbols)
Total n-grams   = 10429
Total words     = 14577
Total OOVs      = 6148 (42.176 percent)
Total sentences = 2000
Average length  = 7.2885 words

Total patterns  = 10429

NNLM Tool Network Topology:

[      layer      neurons,      weights,      outputs]
[  projection      8,          53,          24]
[    hidden      16,          24,          16]
[    output      52,          16,          52]

Initializing network weights and biases: -0.2 <= w <= 0.2

Batch size = 1 pattern(s)

START: Sat Jul 09 15:45:16 2005

[0] [LP=  -40194.05 | PPL=47.1845 (10429) | t=0.95 ] lambda=0.000500
[1] [LP=  -38850.97 | PPL=41.4829 (10429) | t=0.95 ] lambda=0.000550
[2] [LP=  -38050.09 | PPL=38.4166 (10429) | t=0.94 ] lambda=0.000605
[3] [LP=  -37289.02 | PPL=35.7129 (10429) | t=0.94 ] lambda=0.000666
[4] [LP=  -36760.75 | PPL=33.9490 (10429) | t=0.94 ] lambda=0.000732
[5] [LP=  -36458.21 | PPL=32.9783 (10429) | t=0.94 ] lambda=0.000805
[6] [LP=  -36274.68 | PPL=32.4030 (10429) | t=0.95 ] lambda=0.000886
[7] [LP=  -36156.68 | PPL=32.0384 (10429) | t=0.97 ] lambda=0.000974

END: Sat Jul 09 15:45:26 2005

```

Figure 17 – Sample program output produced during neural network training

Appendix D – Project Distribution

The folder hierarchy for the project distribution is shown in figure 18 below.

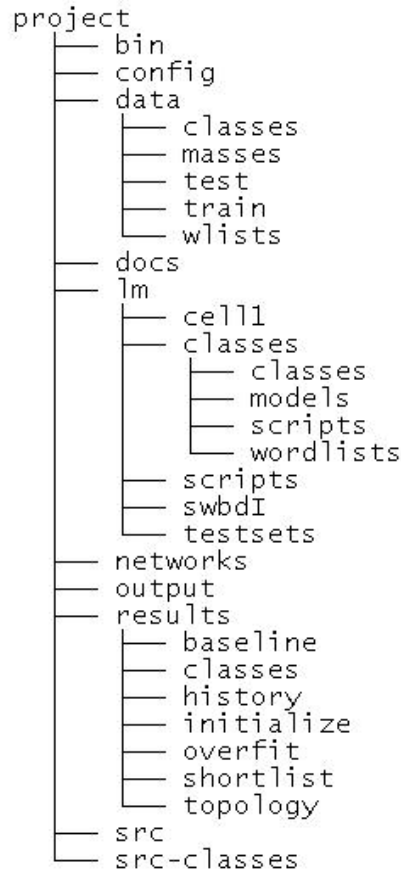


Figure 18 – Project distribution folder hierarchy

The project binaries and scripts are located in the `bin/` folder. The project should be built using the `makefile` in the `src/` folder – this copies the resulting binaries and utilities to `bin/`.

Sample configuration files can be found in the `config/` folder. The most general of these is `network.cfg`. The other configuration files were used for script-based automatic evaluation of the experiments whose logs and trained networks are stored in the `results/` folder.

The `lm/` folder contains the word- and class-based language models developed using the SRI-LM and HLM tools.

A more detailed description of the project organisation can be found in the user manual located in the `docs/` folder.

Bibliography and References

- [1] Schwenk, H. and Gauvain, J.-L. (2002), Connectionist Language Modelling for Large Vocabulary Continuous Speech Recognition, in Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, volume 1, pages 765-768.
- [2] Schwenk, H. and Gauvain, J.-L. (2003), Using Continuous Space Language Models for Conversational Speech Recognition, in IEEE Workshop on Spontaneous Speech Processing and Recognition, April 14-16, 2003, Tokyo
- [3] Schwenk, H. and Gauvain, J.-L. (2004a), Neural Network Language Models for Conversational Speech Recognition, in Proc. Int. Conf. Spoken Language Processing.
- [4] Schwenk, H. and Gauvain, J.-L. (2004b), Using Network Language models for LVCSR, DARPA's RT04F EARS Workshop.
- [5] Schwenk, H. (2004c), Efficient Training of Large Neural Networks for Language Modeling, in International Joint Conference on Neural Networks, pages 3059 – 3062.
- [6] Bishop, C. M. (1995), Neural Networks for Pattern Recognition, Clarendon Press - Oxford.
- [7] Duda, R.O., Hart, P.E., Stork, D.G. (2000), Pattern Classification, 2nd Ed., Wiley-Interscience
- [8] Knight, K. and Rich, E. (1991) Artificial Intelligence, 2nd Ed., McGraw-Hill, Inc.
- [9] Winston, P. H. (1993), Artificial Intelligence, 3rd Ed., Addison-Wesley Publishing Company.
- [10] Manning, C. D. and Schutz, H. (1999), Foundations of Statistical Natural Language Processing, MIT Press.
- [11] Jurafsky, D. and Martin, J. H. (2000), Speech and Language Processing, Prentice Hall
- [12] Charniak, E. (1994), Statistical Language Learning, MIT Press
- [13] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, An Updated Set of Basic Linear Algebra Subprograms (BLAS), Volume 28 , Issue 2, June 2002, pp 135 - 151, ACM TOMS, ISSN 0098-3500.

[14] Gales, M. (2001), Paper I10: Advanced Pattern Processing, Handouts 4 & 5: Multi-layer perceptron: introduction and training, Engineering Part IIB & EIST Part II, University of Cambridge

[15] Young, SJ and Evermann, G and Hain, T and Kershaw, D and Moore, GL and Odell, JJ and Ollason, D and Povey, D and Valtchev, V and Woodland, PC (2002), The HTK Book (Version 3.2.1)

[16] Stolcke, A., (2002), SRILM - An Extensible Language Modeling Toolkit, in Proc. Intl. Conf. Spoken Language Processing, Denver, Colorado, September 2002

[17] SRI-LM – The Language Modeling Toolkit:
<http://www.speech.sri.com/projects/srilm/>

[18] BLAS – Quick Reference:
<http://www.netlib.org/blas/blasqr.ps>

[19] Intel Maths Kernel Library:
<http://www.intel.com/cd/software/products/asm-na/eng/perflib/mkl/index.htm>

[20] Intel Maths Kernel Library – Quick Reference:
<http://www.intel.com/software/products/mkl/docs/mklqref/index.htm>

[21] Neural Network FAQ:
<ftp://ftp.sas.com/pub/neural/FAQ.html>

[22] Linguistic Data Consortium:
<http://www ldc.upenn.edu/>

[23] John J. Godfrey, and Edward Holliman, SWITCHBOARD-1 Release 2:
<http://www ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC97S62>

[24] David Graff, Kevin Walker, David Miller, Switchboard Cellular Part 1 Transcription:
<http://www ldc.upenn.edu/Catalog/catalogEntry.jsp?catalogId=LDC2001T14>

[25] Losinger, Chris, CCmdLine – Command Line Parser in C++ using STL
<http://www.codeproject.com/cpp/ccmdline.asp>

ldf;s

asdl;jas

as
dldf;s

asdl;jas

as
d