Technische Universität München
Fakultät für Informatik
Lehrstuhl VI: Echtzeitsysteme und Robotik

Supervised Sequence Labelling with Recurrent Neural Networks

Alex Graves

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. B. Brügge, Ph.D
Prüfer der Dissertation:	1. Univ.-Prof. Dr. H. J. Schmidhuber 2. Univ.-Prof. Dr. St. Kramer

Die Dissertation wurde am 14.01.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 19.06.2008 angenommen.

Abstract

Recurrent neural networks are powerful sequence learners. They are able to incorporate context information in a flexible way, and are robust to localised distortions of the input data. These properties make them well suited to sequence labelling, where input sequences are transcribed with streams of labels. *Long short-term memory* is an especially promising recurrent architecture, able to bridge long time delays between relevant input and output events, and thereby access long range context. The aim of this thesis is to advance the state-of-the-art in supervised sequence labelling with recurrent networks in general, and long short-term memory in particular. Its two main contributions are (1) a new type of output layer that allows recurrent networks to be trained directly for sequence labelling tasks where the alignment between the inputs and the labels is unknown, and (2) an extension of long short-term memory to multidimensional data, such as images and video sequences. Experimental results are presented on speech recognition, online and offline handwriting recognition, keyword spotting, image segmentation and image classification, demonstrating the advantages of advanced recurrent networks over other sequential algorithms, such as hidden Markov Models.

Acknowledgements

I would like to thank my supervisor Jürgen Schmidhuber for his guidance and support. I would also like to thank my co-authors Santi, Tino, Nicole and Doug, and everyone else at IDSIA for making it a stimulating and creative place to work. Thanks to Tom Schaul for proofreading the thesis, and Marcus Hutter for his mathematical assistance during the connectionist temporal classification chapter. I am grateful to Marcus Liwicki and Horst Bunke for their expert collaboration on handwriting recognition. A special mention goes to Fred and Matteo and all the other Idsiani who helped me find the good times in Lugano. Most of all, I would like to thank my family and my wife Alison for their constant encouragement, love and support.

This research was supported in part by the Swiss National Foundation, under grants 200020-100249, 200020-107534/1 and 200021-111968/1.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

In machine learning, the term *sequence labelling* encompasses all tasks where sequences of data are transcribed with sequences of labels. Well-known examples include speech and handwriting recognition, protein secondary structure prediction and part-of-speech tagging. *Supervised* sequence labelling refers specifically those cases where a set of inputs with target transcriptions is provided for algorithm training. What distinguishes such problems from the traditional framework of supervised pattern classification is that the individual data points cannot be assumed to be independent and identically distributed (i.i.d.). Instead, both the inputs and the labels form strongly correlated sequences. In speech recognition for example, the input (a speech signal) is produced by the continuous motion of the vocal tract, while the labels (a sequence of words) are constrained by the laws of syntax and grammar. A further complication is that in many cases the alignment between inputs and labels is unknown. This requires the use of algorithms able to determine the location as well as the identity of the output labels.

Recurrent neural networks (RNNs) have several properties that make them an attractive choice for sequence labelling. They are able to incorporate contextual information from past inputs (and future inputs too, in the case of bidirectional RNNs), which allows them to instantiate a wide range of sequence-to-sequence maps. Furthermore, they are robust to localised distortions of the input sequence along the time axis. The purpose of this thesis is to extend and apply RNNs to real-world tasks in supervised sequence labelling.

One shortcoming of RNNs is that the range of contextual information they have access to is in practice quite limited. Long Short-Term Memory (LSTM) is an RNN architecture specifically designed to overcome this limitation. In various synthetic tasks, LSTM has been shown capable of bridging very long time lags between relevant input and target events. Moreover, it has also proved advantageous in real-world domains such as speech processing and bioinformatics. For this reason, LSTM will be the RNN of choice throughout the thesis, and a major subgoal is to analyse, apply and extend the LSTM architecture.

1.1 Contributions

In the original formulation of LSTM (Hochreiter and Schmidhuber, 1997) an approximate form of the error gradient was used for training. We present equations for the exact gradient, evaluated with backpropagation through time (Williams and Zipser, 1995). As well as being more accurate, the exact gradient has the advantage of being easier to debug. We also combine LSTM with bidirectional RNNs (Schuster and Paliwal, 1997) to give bidirectional LSTM (BLSTM), thereby providing access to long range context in both input directions. We compare the performance of BLSTM to other neural network architectures on the task of framewise phoneme classification.

Framewise phoneme classification requires the alignment between the speech signal and the output labels (the phoneme classes) to be known in advance, e.g. using hand labelled data. However, for full speech recognition, and many other real-world sequence labelling tasks, the alignment is unknown. So far, the most effective way to apply RNNs to such tasks has been to combine them with hidden Markov models (HMMs) in the so-called hybrid approach (Bourlard and Morgan, 1994). We investigate the application of HMM-LSTM hybrids to phoneme recognition.

More importantly, we introduce an output layer, known as *connectionist temporal classification* (CTC), that allows RNNs to be trained directly for sequence labelling tasks with unknown input-label alignments, without the need for HMM hybrids. For some tasks, we want to constrain the output label sequence to obey some probabilistic grammar. In speech recognition, for example, we usually want the output to be a sequence of dictionary words, and may wish to weight the probabilities of different word sequences according to a probabilistic language model. We present an efficient decoding algorithm that constrains CTC outputs according to a dictionary and, if necessary, a bigram language model. In several experiments on speech and handwriting recognition, we find that a BLSTM network with a CTC output layer outperforms HMMs and HMM-RNN hybrids, often by a substantial margin. In particular, we obtain excellent results in online handwriting recognition, both with and without task-specific preprocessing.

RNNs were originally designed for one-dimensional time series. However, some of their advantages in time-series learning, such as robustness to input distortion, and the use of contextual information, are also desirable in multidimensional domains, such as image and video processing. Multidimensional RNNs (MDRNNs), a special case of *directed acyclic graph RNNs* (DAG-RNN; Baldi and Pollastri, 2003), are a generalisation of RNNs to multidimensional data. MDRNNs have a separate recurrent connection for every spatio-temporal dimension in the data, thereby matching the structure of the network to the structure of the input. In particular, multidirectional MDRNNs give the network access to context information in all input directions. We introduce multidimensional LSTM by adding extra recurrent connections to the internal structure of the LSTM memory blocks, thereby bringing the benefits of long range context to MDRNNs. We apply multidimensional LSTM to two image segmentation tasks, and find that it is more robust to distortion than a state-of-the-art image classification algorithm.

1.2 Overview of Thesis

The chapters are roughly grouped into three parts: background material is presented in Chapters 2–4, Chapters 5 and 6 are primarily experimental, and new methods are introduced in Chapters 7 and 8.

Chapter 2 briefly reviews supervised learning in general, and supervised pattern classification in particular. It also provides a formal definition of supervised sequence labelling, and presents a taxonomy of sequence labelling tasks that arise under different assumptions about the relationship between the input and label sequences. Chapter 3 provides background material for feedforward and recurrent neural networks, with emphasis on their application to classification and labelling tasks. It also describes the use of the *sequential Jacobian*, which is the matrix of partial derivatives of a particular output with respect to a sequence of inputs, as a tool for analysing the use of contextual information by RNNs. Chapter 4 describes the LSTM architecture, presents equations for the full LSTM error gradient, and introduces bidirectional LSTM (BLSTM). Chapter 5 contains an experimental comparison of BLSTM to other neural network architectures on the task of framewise phoneme classification. Chapter 6 investigates the use of LSTM in hidden Markov model-neural network hybrids. Chapters 7 and 8 introduce connectionist temporal classification and multidimensional RNNs respectively, and concluding remarks and directions for future work are given in Chapter 9.

Chapter 2

Supervised Sequence Labelling

This chapter provides the background material and literature review for supervised sequence labelling. Section 2.1 briefly reviews supervised learning in general. Section 2.2 covers the classical, non-sequential framework of supervised pattern classification. Section 2.3 defines supervised sequence labelling, and presents a taxonomy of sequence labelling tasks that arise under different assumptions about the label sequences, discussing algorithms and error measures suitable for each one.

2.1 Supervised Learning

Machine learning problems where a set of input-target pairs is provided for training are referred to as *supervised learning* tasks. This is distinct from *reinforcement learning*, where only a positive or negative reward value is provided for training, and *unsupervised learning*, where no task-specific training signal exists at all, and the algorithm attempts to uncover the structure of the data by inspection alone.

The nature and degree of supervision provided by the targets varies greatly between supervised learning tasks. For example, training a supervised learner to correctly label every pixel corresponding to an aeroplane in an image requires a much more informative target than simply training it recognise whether or not an aeroplane is present. To distinguish these extremes, people sometimes refer to *weakly* and *strongly* labelled data. Although we will not use this terminology in the thesis, the output layer introduced in Chapter 7 can be thought of

A standard supervised learning task consists of a *training set* S of input-target pairs (x, z), where x is an element of the input space $\mathcal{X}$ and z is an element of the target space $\mathcal{Z}$, and a disjoint *test set* S', both drawn independently from the same input-target distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Z}}$. In some cases an extra *validation set* is drawn from the training set to validate the performance of the learning algorithm. The goal of the task is to use the training set to minimise some task-specific error measure E evaluated on the test set. For example, in a regression task, the usual error measure is the *sum-of-squares*, or squared Euclidean distance between the algorithm outputs and the target vectors. For parametric algorithms (such as neural networks) the usual approach to error minimisation is to incrementally adjust the algorithm parameters to optimise an *objective function* O on the training set, where O is related, but not necessarily identical, to E. The ability of an algorithm to transfer performance from the training set to the test set is referred to as *generalisation*, and is discussed in the context of neural networks in Section 3.3.2.

2.2 Pattern Classification

Pattern classification, also known as pattern recognition, has been extensively studied in the machine learning literature (Bishop, 2006; Duda et al., 2000), and various pattern classification algorithms, such as multilayer perceptrons (Rumelhart et al., 1986; Bishop, 1995) and support vector machines (Vapnik, 1995), are routinely used for commercial tasks.

Although pattern classification deals with non-sequential data, much of the practical and theoretical framework underlying it carries over to the sequential case. It is therefore instructive to briefly review this framework before we turn to sequence labelling proper.

Since we will deal exclusively with *supervised* pattern classification, we assume the presence of a training set S of input-target pairs (x, z), where each input x is a real-valued vector of some fixed length M, and each target z represents a single class drawn from a set of K classes. A fundamental assumption underlying most work on supervised pattern classification is that the input-target pairs are independent and identically distributed (i.i.d.). Then let h be a pattern classifier that maps from input vectors onto class labels, and S' be a *test set* of input-target pairs that are not contained in S. For pattern classification tasks where all misclassifications are equally bad, the aim is to use S to train a classifier in such a way that minimises *classification error rate* E^{class} on S'

$$E^{class}(h, S') = \frac{1}{|S'|} \sum_{(x,z)\in S'} \begin{cases} 0 \text{ if } h(x) = z \\ 1 \text{ otherwise} \end{cases} \tag{2.1}$$

In the more general case where different mistakes have different weights, we can define a *loss matrix* L whose elements l_{ij} are the losses incurred by assigning a pattern with true class C_i to class C_j. In this situation we seek instead to minimise the *classification loss* E^{loss} on S'

$$E^{loss}(h, L, S') = \frac{1}{|S'|} \sum_{(x,z)\in S'} L_{h(x)z}, \tag{2.2}$$

Note that (2.1) can be recovered as a special case of (2.2), where all diagonal elements of L are 0, and all others are 1.

2.2.1 Probabilistic Classification

One approach to pattern classification is to to train a classifier that directly maps from input patterns onto class labels. This type of classifier, of which support vector machines are a well known example, is referred to as a *discriminant function*. Another approach is *probabilistic classification*, where the conditional probabilities $p(C_k|x)$ of the K classes given the input pattern x are first determined, and the most probable is then chosen as the classifier output $h(x)$:

$$h(x) = \arg\max_k p(C_k|x) \tag{2.3}$$

There are numerous advantages to the probabilistic approach. One of these is that retaining the probabilities of the different labels allows different classifiers to be combined in a consistent way. Another is that the probabilities can be used to account for the different losses associated with different types of misclassification. In the latter case Eqn. (2.3) is modified by the addition of the loss matrix L:

$$h(x) = \arg\min_j \sum_k L_{kj} p(C_k|x) \tag{2.4}$$

2.2.2 Training Probabilistic Classifiers

A further benefit of probabilistic classification is that it provides a consistent methodology for algorithm training. If a probabilistic classifier h_w with adjustable parameters w yields a particular conditional probability distribution $p(C_k|x, w)$ over the class labels C_k, we can take a product over the i.i.d. input-target pairs in the training set S to get

$$p(S|w) = \prod_{(x,z)\in S} p(z|x, w), \tag{2.5}$$

which we can invert with Bayes' theorem to obtain

$$p(w|S) = \frac{p(S|w)p(w)}{p(S)} \tag{2.6}$$

In theory, the posterior distribution over classes for some new input x can then be found by integrating over all possible values of w:

$$p(C_k|x, S) = \int_w p(C_k|x, w)p(w|S)dw \tag{2.7}$$

In practice though, w is usually very high dimensional and calculating the above distribution, known as the *predictive distribution* of h, is intractable. A common approximation, known as the maximum a priori (MAP) approximation, is to find the single parameter vector w_{MAP} that maximises (2.6) and use this to make predictions:

$$p(C_k|x) = p(C_k|x, w_{MAP}) \tag{2.8}$$

Since $p(S)$ is independent of w, we have

$$w_{MAP} = \arg\max_w p(S|w)p(w) \tag{2.9}$$

The parameter prior $p(w)$ is usually referred to as a *regularisation term.* Its effect is to weight the classifier towards those parameter values which are deemed *a priori* more probable. In accordance with Occam's razor, we usually assume that more complex parameters (where 'complex' is typically interpreted as 'requiring more information to accurately describe') are inherently less probable. For this reason $p(w)$ is sometimes referred to as an *Occam factor* or *complexity penalty.* In the particular case of a Gaussian parameter prior, where $p(w) \propto |w|^2$, the $p(w)$ term is referred to as *weight decay.* If, on the other hand, we assume a uniform prior over parameters, we can drop the $p(w)$ term from (2.9) to obtain the maximum likelihood (ML) parameter vector w_{ML}

$$w_{ML} = \arg\max_w p(S|w) = \arg\max_w \prod_{(x,z)\in S} p(z|x, w) \tag{2.10}$$

The standard procedure for finding w_{ML} is to minimise an objective function O defined as the negative logarithm of $p(S|w)$

$$O = -\ln \prod_{(x,z)} p(z|x, w) = -\sum_{(x,z)} \ln p(z|x, w), \tag{2.11}$$

where ln is the logarithm to base e. Note that, since the logarithm is monotonically increasing, minimising $-\ln p(S|w)$ is equivalent to maximising $p(S|w)$.

We will return to the maximum likelihood approximation in Chapters 3 and 7, where we will use it to derive objective functions for neural networks.

2.2.3 Generative and Discriminative Models

Algorithms that directly calculate the class probabilities $p(C_k|x)$ (often referred to as the posterior class probabilities) are called *discriminative models.* In some cases however, it is preferable to first calculate the class conditional densities $p(x|C_k)$ and then use Bayes' theorem, together with the prior class probabilities $p(C_k)$ to find the posterior values

$$p(C_k|x) = \frac{p(x|C_k)p(C_k)}{p(x)}, \tag{2.12}$$

where

$$p(x) = \sum_k p(x|C_k)p(C_k), \tag{2.13}$$

Algorithms following this approach are known as *generative models*, because the input priors $p(x)$ can be used to generate artificial input data. One advantage of the generative approach is that each class can be trained independently of the others, whereas discriminative models have to be retrained every time a new class is added. However, discriminative methods typically give better results for classification tasks, because they concentrate all their modelling effort on finding the correct class boundaries.

This thesis focuses on discriminative models for sequence labelling. However, we will refer at various points to the well known generative model *hidden Markov models* (HMMs; Rabiner, 1989; Bengio, 1999).

2.3 Sequence Labelling

The goal of sequence labelling is to assign a sequence of labels, drawn from a fixed and finite alphabet, to a sequence of input data. For example, one might wish to label a sequence of acoustic features with spoken words (speech recognition), or a sequence of video frames with hand gestures (gesture recognition). Although such tasks commonly arise when analysing time series, they are also found in domains with non-temporal sequences, such as protein secondary structure prediction.

For some problems the precise alignment of the labels with the input data must also be determined by the learning algorithm. In this thesis however, we limit our attention to tasks where the alignment is either predetermined, by some manual or automatic preprocessing, or it is unimportant, in the sense that we require only the final *sequence* of labels, and not the times at which they occur.

Sequence labelling differs from pattern classification in that the inputs are sequences $\mathbf{x}$ of fixed size, real-valued vectors, while the targets are sequences $\mathbf{z}$ of discrete labels, drawn from some finite alphabet L. (From now on we will use a bold typeface to denote sequences). As with pattern classification, we assume that the input-target pairs $(\mathbf{x}, \mathbf{z})$ are i.i.d. Of course we do *not* assume the individual data-points within each sequence to be i.i.d. While the i.i.d. assumption may not be entirely correct (e.g. when the input sequences represent turns in a spoken dialogue, or lines in a handwritten form) it is reasonable as long as the sequence boundaries are sensibly

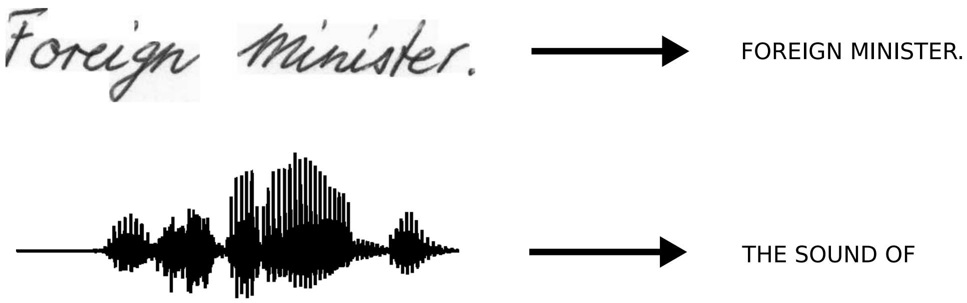


Figure 2.1: **Sequence labelling.** The algorithm receives a stream of input data, and outputs a sequence of discrete labels.

chosen. We further assume that the label sequences are at most as long as the corresponding input sequences. With these restrictions in mind we can formalise the task of sequence labelling as follows:

Let S be a set of training examples drawn independently from a fixed distribution $\mathcal{D}_{\mathcal{X}\times\mathcal{Z}}$. The input space $\mathcal{X} = (\mathbb{R}^M)^*$ is the set of all sequences of size M real-valued vectors. The target space $\mathcal{Z} = L^*$ is the set of all sequences over the (finite) alphabet L of labels. We refer to elements of L^* as *label sequences* or *labellings*. Each element of S is a pair of sequences $(\mathbf{x}, \mathbf{z})$. The target sequence $\mathbf{z} = (z_1, z_2, ..., z_U)$ is at most as long as the input sequence $\mathbf{x} = (x_1, x_2, ..., x_T)$, i.e. $U \leq T$. Regardless of whether the data is a time series, we refer to the distinct points in the input sequence as *timesteps*.

The task is then to use S to train a sequence labelling algorithm $h : \mathcal{X} \mapsto \mathcal{Z}$ to label the sequences in a test set $S' \subset \mathcal{D}_{\mathcal{X}\times\mathcal{Z}}$, disjoint from S, in a way that minimises some task-dependent error measure.

2.3.1 A Taxonomy of Sequence Labelling Tasks

So far we have described sequence labelling in its most general form. However, in some cases we can apply further constraints to the type of label sequences required. These constraints affect both the choice of algorithm and the error measures used to assess performance. In what follows, we provide a taxonomy of three classes of sequence labelling task, and discuss algorithms and error measures suitable for each class. The taxonomy is outlined in Figure 2.2

2.3.2 Sequence Classification

In the most restrictive case, the label sequences are constrained to be length one. We refer to this as *sequence classification*, since each input sequence is assigned to a single class.

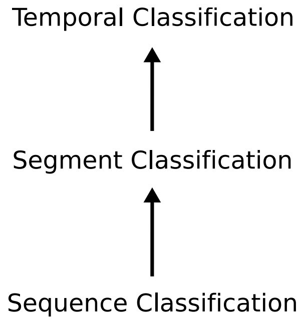


Figure 2.2: **Taxonomy of sequence labelling tasks.** Sequence classification, where each input sequence is assigned a single class, is a special case of segment classification, where each of a predefined set of input segments is given a label. Segment classification is a special case of temporal classification, where any alignment between input and label sequences is allowed.

Examples of sequence classification tasks include the identification of a single face and the recognition of an individual handwritten letter. A key feature of sequence classification is that the entire sequence can be processed before the classification is made.

If the input sequences are of fixed length, or can be easily padded to a fixed length, they can be collapsed into a single input vector and any of the standard pattern classification algorithms mentioned in Section 2.2 can be applied. A prominent testbed for fixed-length sequence classification is the MNIST isolated digits dataset (LeCun et al., 1998a). Numerous pattern classification algorithms have been applied to MNIST, including convolutional neural networks (LeCun et al., 1998a; Simard et al., 2003) and support vector machines (LeCun et al., 1998a; Decoste and Schölkopf, 2002).

However, even if the input length is fixed, algorithm that are inherently sequential may be beneficial, since they are better able to adapt to translations and distortions in the input data. This is the rationale behind our application of multidimensional recurrent neural networks to MNIST in Chapter 8.

Since one classification is made for each sequence, the error measures used for pattern classification can also be applied here, e.g.

$$E^{seq}(h, S') = \frac{1}{|S'|} \sum_{(\mathbf{x},\mathbf{z}) \in S'} \begin{cases} 0 \text{ if } h(\mathbf{x}) = \mathbf{z} \\ 1 \text{ otherwise} \end{cases} , \tag{2.14}$$

which we refer to as the *sequence error rate*.

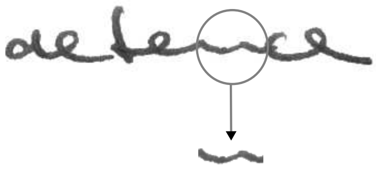


Figure 2.3: **Importance of context in segment classification.** The word 'defence' is clearly legible. However the letter 'n' in isolation is ambiguous.

2.3.3 Segment Classification

In many cases however, multiple classifications are required for a single input sequence. For example, we might wish to classify each letter in a line of handwriting, or each phoneme in a spoken utterance. If the positions of the input segments for which the classifications are required are known in advance, we refer to such tasks as *segment classification.*

A crucial element of segment classification, missing from sequence classification, is the use of *context* — i.e. data on either side of the segments to be classified. The effective use of context is vital to the success of segment classification algorithms, as illustrated in Figure 2.3. This presents a problem for standard pattern classification algorithms, which are designed to process only one input at a time. A simple solution is to collect the data on either side of the segment into a *time-window*, and use this as an input pattern. However, as well as the aforementioned issue of shifted or distorted data, the time-window approach suffers from the fact that the range of useful context (and therefore the required time-window size) is generally unknown, and may vary widely from segment to segment. Consequently the case for sequential algorithms is stronger here than in sequence classification.

An obvious error measure for segment classification tasks is the *segment error rate* $E^{seg}(h, S')$, which simply counts the number of misclassified segments

$$E^{seg}(h, S') = \frac{1}{Z} \sum_{(\mathbf{x},\mathbf{z}) \in S'} HD(h(\mathbf{x}), \mathbf{z}) \tag{2.15}$$

Where Z is the total length of all target sequences in the test set, and $HD(\mathbf{p}, \mathbf{q})$ is the hamming distance between two equal length sequences $\mathbf{p}$ and $\mathbf{q}$ (i.e. the number of places in which they differ).

In speech recognition, the classification of each acoustic frame as a separate segment is often known as *framewise phoneme classification.* In this context the segment error rate is usually referred to as the *frame error rate.* We apply various neural network architectures to framewise phoneme classification in Chapter 5. In image processing, the classification of each pixel, or block of pixels, as a separate segment is known as *image segmentation.* We apply multidimensional recurrent neural networks to image segmentation in Chapter 8.

2.3.4 Temporal Classification

In the most general case, nothing can be assumed about the label sequences except that their length is less than or equal to that of the input sequences. They may even be empty. We refer to this situation as *temporal classification* (Kadous, 2002).

The key distinction between temporal classification and segment classification is that the former requires an algorithm that can decide *where* in the input sequence the classifications should be made. This in turn requires an implicit or explicit model of the global structure of the sequence.

For temporal classification, the segment error rate is inapplicable, since the alignment between the input and label sequences is unknown. Instead we measure the number of label substitutions, insertions and deletions that would be required to turn one sequence into the other, giving us the label error rate $E^{lab}(h, S')$:

$$E^{lab}(h, S') = \frac{1}{Z} \sum_{(\mathbf{x},\mathbf{z}) \in S'} ED(h(\mathbf{x})), \tag{2.16}$$

Where Z is the total length of all target sequences in the test set, and $LEV(\mathbf{p}, \mathbf{q})$ is the *edit* or*Levenshtein* distance between the two sequences $\mathbf{p}$ and $\mathbf{q}$ (i.e. the minimum number of insertions, substitutions and deletions required to change $\mathbf{p}$ into $\mathbf{q}$). See for example (Navarro, 2001) for efficient edit distance calculation algorithms.

A family of similar error measures can be defined by introducing other types of edit operation, such as transpositions (caused by e.g. typing errors), or by weighting the relative importance of the operations. For the purposes of this thesis however, the label error rate is sufficient. We will usually refer to the label error rate according to the type of label in question, e.g. phoneme error rate, word error rate etc.

For some temporal classification tasks a completely correct labelling is required (for example when reading a postcode from an envelope) and the degree of error is unimportant. In this cases the sequence error rate (2.14) should be used to assess performance.

We investigate the use of hidden Markov model-recurrent neural network hybrids for temporal classification in Chapter 6, and introduce a neural network only approach to temporal classification in Chapter 7.

Chapter 3

Neural Networks

This chapter provides the background material and literature review for neural networks, with particular emphasis on their application to classification and labelling tasks. Section 3.1 reviews multilayer perceptrons and their application to pattern classification. Section 3.2 reviews recurrent neural networks and their application to sequence labelling. It also describes the sequential Jacobian, an analytical tool for studying the use of context information by RNNs. Section 3.3 discusses various issues, such as generalisation and input data representation, that are essential to effective network training.

3.1 Multilayer Perceptrons

Artificial neural networks (ANNs) were originally developed as mathematical models of the information processing capabilities of biological brains (McCulloch and Pitts, 1988; Rosenblatt, 1963; Rumelhart et al., 1986). Although it is now clear that ANNs bear little resemblance to real biological neurons, they enjoy continuing popularity as pattern classifiers.

The basic structure of an ANN is a network of small processing units, or nodes, which are joined to each other by weighted connections. In terms of the original biological model, the nodes represent neurons, and the connection weights represent the strength of the synapses between the neurons. The network is activated by providing an input to some or all of the nodes, and this activation then spreads throughout the network along the weighted connections. The electrical activity of biological neurons typically follows a series of sharp 'spikes', and the activation of an ANN node was originally intended to model the average firing rate of these spikes.

Many varieties of ANNs have appeared over the years, with widely varying properties. One important distinction is between ANNs whose connections form cycles, and those whose connections are acyclic. ANNs with cycles are referred to as feedback, recursive, or recurrent, neural networks,

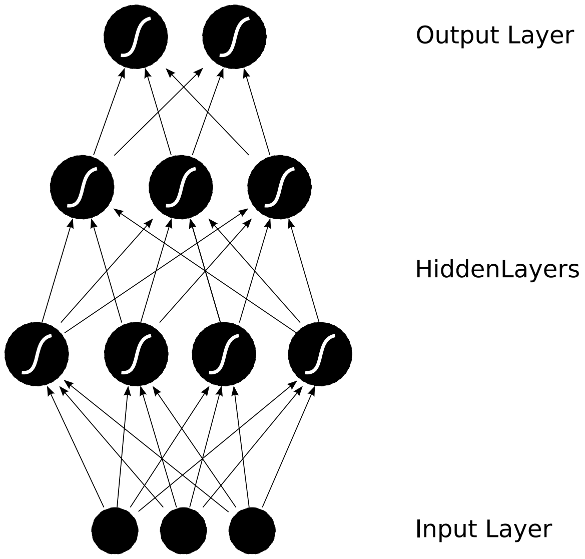
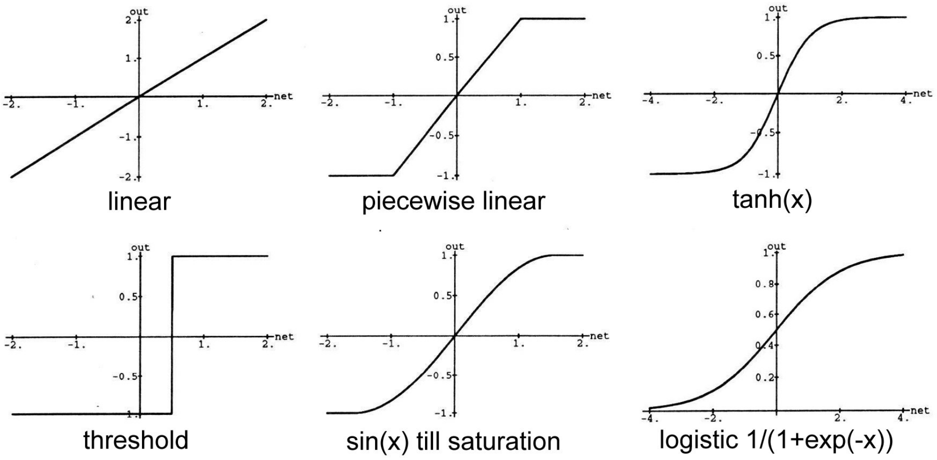


Figure 3.1: **Multilayer perceptron**

and are dealt with in Section 3.2. ANNs without cycles are referred to as feedforward neural networks (FNNs). Well known examples of FNNs include perceptrons (Rosenblatt, 1958), radial basis function networks (Broomhead and Lowe, 1988), Kohonen maps (Kohonen, 1989) and Hopfield nets (Hopfield, 1982). The most widely used form of FNN, and the one we focus on in this section, is the *multilayer perceptron* (MLP; Rumelhart et al., 1986; Werbos, 1988; Bishop, 1995).

The units in a multilayer perceptron are arranged in layers, with connections feeding forward from one layer to the next (Figure 3.1). Input patterns are presented to the input layer, and the resulting unit activations are propagated through the hidden layers to the output layer. This process is known as the *forward pass* of the network. The units in the hidden layers have (typically nonlinear) *activation functions* that transform the summed activation arriving at the unit. Since the output of an MLP depends only on the current input, and not on any past or future inputs, MLPs are more suitable for pattern classification than for sequence labelling. We will discuss this point further in Section 3.2.

An MLP can be thought of as a function that maps from input to output vectors. Since the behaviour of the function is parameterised by the connection weights, a single MLP is capable of instantiating many different functions. Indeed it has been proven (Hornik et al., 1989) that an MLP with a single hidden layer containing a sufficient number of nonlinear units can approximate *any* continuous function on a compact input domain to arbitrary precision. For this reason MLPs are said to be *universal function*



linear

piecewise linear

tanh(x)

threshold

sin(x) till saturation

logistic 1/(1+exp(-x))

Figure 3.2: **Neural network activation functions**

approximators.

3.1.1 Forward Pass

Consider an MLP with I input units, activated by input vector x. Each unit in the first hidden layer calculates a weighted sum of the input units. For hidden unit h, we refer to this sum as the *network input* to unit h, and denote it a_h. The *activation function* θ_h is then applied, yielding the final activation b_h of the unit. Denoting the weight from unit i to unit j as w_{ij}, we have

$$a_h = \sum_{i=1}^{I} w_{ih} x_i \tag{3.1}$$

$$b_h = \theta_h(a_h) \tag{3.2}$$

The two most common choices of activation function are the hyperbolic tangent

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}, \tag{3.3}$$

and the logistic sigmoid

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.4}$$

Note that these functions are related by the linear transform $tanh(x) = 2\sigma(2x) - 1$, and are therefore essentially equivalent (any function computed by a network with a hidden layer of σ units can be computed by another network with $tanh$ units and vice-versa). Although there are differences in the way a network learns with the two functions (e.g. because *tanh* has a steeper slope) we have found that they give the same results in practice, and the two are used interchangeably throughout the thesis.

One vital feature of σ and *tanh* is their nonlinearity. Nonlinear neural networks are far more powerful than linear ones since they can, for example, find nonlinear classification boundaries and model nonlinear equations. Moreover, any combination of linear operators is itself a linear operator, which means that any MLP with multiple linear hidden layers is exactly equivalent to some other MLP with a single linear hidden layer. This contrasts with nonlinear networks, which can gain considerable power by using successive hidden layers to re-represent the input data (Hinton et al., 2006; Bengio and LeCun, 2007).

Another key property is that both functions are differentiable, which allows the network to be trained with gradient descent. Their first derivatives are

$$tanh'(a) = 1 - tanh(a)^2 \tag{3.5}$$

$$\sigma'(a) = \sigma(a)(1 - \sigma(a)) \tag{3.6}$$

Because of the way they squash an infinite input domain into a finite output range, neural network activation functions are sometimes referred to as *squashing functions.*

Having calculated the activations of the units in the first hidden layer, the process of summation and activation is then repeated for the rest of the hidden layers in turn, e.g. for unit h in hidden layer l

$$a_h = \sum_{h'=1}^{H_{l-1}} w_{h'h} b_{h'} \tag{3.7}$$

$$b_h = \theta_h(a_h), \tag{3.8}$$

Where H_l is the number of units in hidden layer l.

3.1.2 Output Layers

The output vector y of an MLP is given by the activation of the units in the output layer. The network input a_k to each output unit k is calculated by summing over the units connected to it, exactly as for a hidden unit, i.e.

$$a_k = \sum_{h=1}^{H_m} w_{hk} b_h, \tag{3.9}$$

for a network with m hidden layers.

Both the number of units in the output layer and the choice of output activation function depend on the task. For binary classification tasks, the standard configuration is a single unit with a logistic sigmoid activation

(Eqn. (3.4)). Since the range of the logistic sigmoid is the open interval $(0, 1)$, the activation of the output unit can be interpreted as an estimate of the probability that the input vector belongs to the first class (and conversely, one minus the activation estimates the probability that it belongs to the second class)

$$\begin{aligned} p(C_1|x) &= y = \sigma(a) \\ p(C_2|x) &= 1 - y \end{aligned} \tag{3.10}$$

The use of the logistic sigmoid as a binary probability estimator is referred to in the statistics literature as *logistic regression*, or a *logit* model. If we use a coding scheme for the target vector z where $z = 1$ if the correct class is C_1 and $z = 0$ if the correct class is C_2, we can combine the above expressions to write

$$p(z|x) = y^z(1 - y)^{1-z} \tag{3.11}$$

For classification problems with $K > 2$ classes, the convention is to have K output units, and normalise the output activations with the *softmax* function (Bridle, 1990) to obtain estimates of the class probabilities:

$$p(C_k|x) = y_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}}, \tag{3.12}$$

which is termed a *multinomial logit* model by statisticians. A *1-of-K* coding scheme represent the target class z as a binary vector with all elements equal to zero except for element k, corresponding to the correct class C_k, which equals one. For example, if $K = 5$ and the correct class is C_2, z is represented by $(0, 1, 0, 0, 0)$. Using this scheme we obtain the following convenient form for the target probabilities:

$$p(z|x) = \prod_{k=1}^{K} y_k^{z_k} \tag{3.13}$$

Given the above definitions, the use of MLPs for pattern classification is straightforward. Simply feed in an input vector, activate the network, and choose the class label corresponding to the most active output unit.

3.1.3 Objective Functions

The derivation of objective functions for MLP training follows the steps outlined in Section 2.2.2. Although attempts have been made to approximate the full predictive distribution of Eqn. (2.7) for neural networks (MacKay, 1995; Neal, 1996), we will here focus on objective functions derived using maximum likelihood. Note that maximum likelihood can be trivially extended to the MAP approximation of Eqn. (2.9) by adding a regularisation term, such as weight-decay, which corresponds to a suitably chosen prior over the weights. For binary classification, substituting (3.11) into the maximum likelihood objective function defined in (2.11), we have

$$O = -\sum_{(x,z)\in S} z \ln y + (1-z)\ln(1-y) \tag{3.14}$$

Similarly, for problems with multiple classes, substituting (3.13) into (2.11) gives

$$O = -\sum_{(x,z)\in S}\sum_{k=1}^{K} z_k \ln y_k, \tag{3.15}$$

Collectively, the above objective functions are referred to as *cross-entropy error*. See (Bishop, 1995, chap. 6) for more information on these and other MLP objective functions.

3.1.4 Backward Pass

Since MLPs are, by construction, differentiable operators, they can be trained to minimise any differentiable objective function using *gradient descent*. The basic idea of gradient descent is to find the derivative of the objective function with respect to each of the network weights, then adjust the weights in the direction of the negative slope. Gradient descent methods for training neural networks are discussed in more detail in Section 3.3.1.

Note that the cross-entropy error terms defined above are a sum over the input-target pairs in the training set. Therefore their derivatives are also a sum of separate terms. From now on, when we refer to the derivatives of an objective function, we implicitly mean the derivatives for one particular input-target pair. The derivatives for the whole training set can then be calculated by summing over the pairs. However, it is often advantageous to consider the derivatives of the pairs separately, since doing so allows the use of online gradient-descent (see Section 3.3.1).

To efficiently calculate the gradient, we use a technique known as *backpropagation* (Rumelhart et al., 1986; Williams and Zipser, 1995; Werbos, 1988). This is often referred to as the *backward pass* of the network.

Backpropagation is simply a repeated application of chain rule for partial derivatives. The first step is to calculate the derivatives of the objective function with respect to the output units. For a binary classification network, differentiating the objective function defined in (3.14) with respect to the network outputs gives

$$\frac{\partial O}{\partial y} = \frac{y-z}{y(1-y)}. \tag{3.16}$$

The chain rule informs us that

$$\frac{\partial O}{\partial a} = \frac{\partial O}{\partial y}\frac{\partial y}{\partial a}, \tag{3.17}$$

and we can then substitute (3.6) and (3.16) into (3.17) to get

$$\frac{\partial O}{\partial a} = y - z \tag{3.18}$$

For a multiclass network, differentiating (3.15) gives

$$\frac{\partial O}{\partial y_k} = -\frac{z_k}{y_k} \tag{3.19}$$

Bearing in mind that the activation of each unit in a softmax layer depends on the network input to every unit in the layer, the chain rule gives us

$$\frac{\partial O}{\partial a_k} = \sum_{k'=1}^{K} \frac{\partial O}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k} \tag{3.20}$$

Differentiating (3.12) we obtain

$$\frac{\partial y_{k'}}{\partial a_k} = y_k \delta_{kk'} - y_k y_{k'}, \tag{3.21}$$

and we can then substitute (3.21) and (3.19) into (3.20) to get

$$\frac{\partial O}{\partial a_k} = y_k - z_k, \tag{3.22}$$

where we have the used the fact that $\sum_{k=1}^{K} z_k = 1$. Note the similarity to (3.18). In (Schraudolph, 2002) the objective function is said to *match* the output layer activation function when the output derivative has this form.

We now continue to apply the chain rule, working backwards through the hidden layers. At this point it is helpful to introduce the following notation:

$$\delta_j \overset{\text{def}}{=} \frac{\partial O}{\partial a_j}, \tag{3.23}$$

where j is any unit in the network. For the units in the last hidden layer, we have

$$\delta_h = \frac{\partial O}{\partial b_h} \frac{\partial b_h}{\partial a_h} = \frac{\partial b_h}{\partial a_h} \sum_{k=1}^{K} \frac{\partial O}{\partial a_k} \frac{\partial a_k}{\partial b_h}, \tag{3.24}$$

where we have used the fact that any objective function O depends only on unit h through its influence on the output units. Differentiating (3.9) and (3.2) and substituting into (3.24) gives

$$\delta_h = \theta'(a_j) \sum_{k=1}^{K} \delta_k w_{hk} \tag{3.25}$$

The δ terms for each hidden layer l before the last one can then be calculated recursively:

$$\delta_h = \theta'(a_h) \sum_{h'=1}^{H_{l+1}} \delta_{h'} w_{hh'} \tag{3.26}$$

Once we have the δ terms for all the hidden units, we can use (3.1) to calculate the derivatives with respect to each of the network weights:

$$\frac{\partial O}{\partial w_{ij}} = \frac{\partial O}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = \delta_j b_i, \tag{3.27}$$

Numerical Gradient

When implementing backpropagation, it is strongly recommended to check the weight derivatives numerically. This can be done by adding positive and negative perturbations to each weight and calculating the changes in the objective function:

$$\frac{\partial O}{\partial w_{ij}} = \frac{O(w_{ij} + \epsilon) - O(w_{ij} - \epsilon)}{2\epsilon} + \bigcirc(\epsilon^2) \tag{3.28}$$

This technique is known as *symmetrical finite differences.* Note that setting ϵ too small leads to numerical underflows and decreased accuracy. The optimal value therefore depends on the computer architecture and floating point accuracy of a given implementation. For the systems we used, $\epsilon = 10^{-5}$ generally gave best results.

3.2 Recurrent Neural Networks

In the previous section we considered ANNs whose connections did not form cycles. If we relax this condition, and allow cyclical connections as well, we obtain *recurrent neural networks* (RNNs). As with FNNs, many varieties of RNN have been proposed, such as Elman networks (Elman, 1990), Jordan networks (Jordan, 1990), time delay neural networks (Lang et al., 1990) and echo state networks (Jaeger, 2001). In this section, we focus on a simple RNN containing a single, self connected hidden layer, as shown in Figure 3.3

While the extension from MLPs to RNNs may seem trivial, the implications for sequence learning are far-reaching. An MLP can only map from input to output vectors, whereas an RNN can in principle map from the entire *history* of previous inputs to each output. Indeed, the equivalent result to the universal approximation theory for MLPs is that an RNN with a sufficient number of hidden units can approximate any measurable sequence-to-sequence mapping to arbitrary accuracy (Hammer, 2000). The key point is that the recurrent connections allow a 'memory' of previous inputs to persist in the network's internal state, which can then be used to influence the network output.

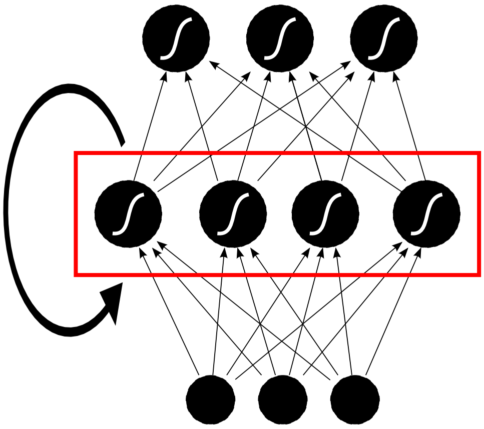


Figure 3.3: **Recurrent neural network**

3.2.1 Forward Pass

The forward pass of an RNN is the same as that of an MLP with a single hidden layer, except that activations arrive at the hidden layer from both the current external input and the hidden layer activations one step back in time. Consider a length T input sequence $\mathbf{x}$ presented to an RNN with I input units, H hidden units, and K output units. Let x_i^t be the value of input i at time t, and let a_j^t and b_j^t be respectively the network input to unit j at time t and the activation of unit j at time t. For the hidden units we have

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1} \tag{3.29}$$

Nonlinear, differentiable activation functions are then applied exactly as for MLPs

$$b_h^t = \theta_h(a_h^t) \tag{3.30}$$

The complete sequence of hidden activations can be calculated by starting at $t = 1$ and recursively applying (3.29) and (3.30), incrementing t at each step. Note that this requires initial values b_i^0 to be chosen for the hidden units, corresponding to the network's state before it receives any information from the data sequence. In this thesis, b_i^0 is always set to zero. However, other researchers have found that in some cases, RNN stability and robustness to noise can be improved by using nonzero initial values (Zimmermann et al., 2006a).

The network inputs to the output units can be calculated at the same time as the hidden activations:

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t \tag{3.31}$$

For sequence classification and segment classification tasks (see Section 2.3) the same output activation functions can be used for RNNs as for MLPs (i.e. softmax and logistic sigmoid), with the classification targets typically presented at the ends of the sequences or segments. It follows that the same objective functions can be used too. Chapter 7 introduces an output layer specifically designed for temporal classification with RNNs.

3.2.2 Backward Pass

Given the partial derivatives of the objective function with respect to the network outputs, we now need the derivatives with respect to the weights. Two well-known algorithms have been devised to efficiently calculate weight derivatives for RNNs: real time recurrent learning (RTRL; Robinson and Fallside, 1987) and backpropagation through time (BPTT; Williams and Zipser, 1995; Werbos, 1990). We focus on BPTT since it is both conceptually simpler and more efficient in computation time (though not in memory).

Like standard backpropagation, BPTT consists of a repeated application of the chain rule. The subtlety is that, for recurrent networks, the objective function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next timestep, i.e.

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{H}\delta_{h'}^{t+1} w_{hh'}\right), \tag{3.32}$$

where

$$\delta_j^t \overset{\text{def}}{=} \frac{\partial O}{\partial a_j^t} \tag{3.33}$$

The complete sequence of δ terms can be calculated by starting at $t = T$ and recursively applying (3.32), decrementing t at each step. (Note that $\delta_j^{T+1} = 0 \; \forall j$, since no error is received from beyond the end of the sequence). Finally, bearing in mind that the weights to and from each unit in the hidden layer are the same at every timestep, we sum over the whole sequence to get the derivatives with respect to each of the network weights

$$\frac{\partial O}{\partial w_{ij}} = \sum_{t=1}^{T}\frac{\partial O}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T}\delta_j^t b_i^t \tag{3.34}$$

3.2.3 Bidirectional RNNs

For many sequence labelling tasks, we would like to have access to future as well as past context. For example, when classifying a particular written letter, it is helpful to know the letters coming after it as well as those before. However, since standard RNNs process sequences in temporal order, they

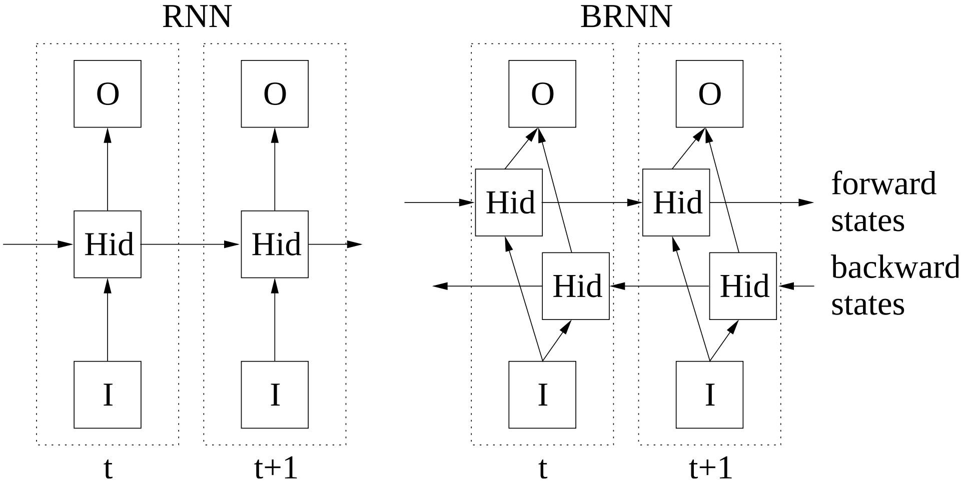


Figure 3.4: **Standard and bidirectional RNNs**

ignore future context. One obvious solution is to add a time-window of future context to the network input. However, as well as unnecessarily increasing the number of input weights, this suffers from the same problems as the time-window approaches discussed in Sections 2.3.2 and 2.3.3, namely intolerance of distortions, and a fixed range of context. Another possibility is to introduce a delay between the inputs and the targets, thereby giving the network a few timesteps of future context. This method retains the RNN's robustness to distortions, but it still requires the range of future context to be determined by hand. Furthermore it places an unnecessary burden on the network by forcing it to 'remember' the original input, and its previous context, throughout the delay. In any case, neither of these approaches remove the asymmetry between past and future information.

Bidirectional recurrent neural networks (BRNNs; Schuster and Paliwal, 1997; Schuster, 1999; Baldi et al., 1999) offer a more elegant solution. The basic idea of BRNNs is to present each training sequence forwards and backwards to two separate recurrent hidden layers, both of which are connected to the same output layer (Figure 3.4). This provides the network with complete, symmetrical, past and future context for every point in the input sequence, without displacing the inputs from the relevant targets. BRNNs have previously given improved results in various domains, notably protein secondary structure prediction (Baldi et al., 2001; Chen and Chaudhari, 2004) and speech processing (Schuster, 1999; Fukada et al., 1999). In this thesis we find that BRNNs consistently outperform unidirectional RNNs on real-world sequence labelling tasks.

The forward pass for the BRNN hidden layers is the same as for a unidirectional RNN, except that the input sequence is presented in opposite directions to the two hidden layers, and the output layer is not updated until both hidden layers have processed the entire input sequence:

```
for $t = 1$ to $T$ do
    Do forward pass for the forward hidden layer, storing activations at each timestep
for $t = T$ to 1 do
    Do forward pass for the backward hidden layer, storing activations at each timestep
for $t = 1$ to $T$ do
    Do forward pass for the output layer, using the stored activations from both hidden layers
```

Algorithm 3.1: BRNN Forward Pass

Similarly, the backward pass proceeds as for a standard RNN trained with BPTT, except that all the output layer δ terms are calculated first, then fed back to the two hidden layers in opposite directions:

```
for $t = T$ to 1 do
    Do BPTT backward pass for the output layer only, storing $\delta$ terms at each timestep
for $t = T$ to 1 do
    Do BPTT backward pass for the forward hidden layer, using the stored $\delta$ terms from the output layer
for $t = 1$ to $T$ do
    Do BPTT backward pass for the backward hidden layer, using the stored $\delta$ terms from the output layer
```

Algorithm 3.2: BRNN Backward Pass

BRNNs and Causal Tasks

A common objection to BRNNs is that they violate causality. Clearly, for tasks such as financial prediction or robot navigation, an algorithm that requires access to future inputs is unfeasible. However, there are many problems for which causality is unnecessary. Most obviously, for tasks where the input sequences are spatial and not temporal (e.g. protein secondary structure prediction), there should be no distinction between past and future inputs. This is perhaps why bioinformatics is the domain where BRNNs have been most widely adopted. However BRNNs can also be applied to temporal tasks, as long as the network outputs are only needed at the end of some input segment. For example, in speech and handwriting recognition, the data is usually divided up into sentences, lines, or dialogue turns, each of which is completely processed before the output labelling is required. Furthermore, even for online temporal tasks, such as automatic dictation, bidirectional algorithms can be used as long as it is acceptable to wait for some natural break in the input, e.g. a pause in speech, before processing a section of the data.

3.2.4 Sequential Jacobian

It should be clear from the preceding discussions that the ability to make use of contextual information is vitally important for sequence labelling. It therefore seems desirable to have a way of analysing exactly where and how an algorithm uses context during a particular data sequence. For RNNs, we can take a big step towards this by measuring the sensitivity of the network outputs to the network inputs.

For feedforward neural networks, the *Jacobian* matrix J is defined as having elements equal to the derivatives of the network outputs with respect to the network inputs

$$J_{ki} = \frac{\partial y_k}{\partial x_i} \tag{3.35}$$

These derivatives measure the relative sensitivity of the outputs to small changes in the inputs, and can therefore be used, for example, to detect irrelevant inputs. The Jacobian can be extended to recurrent neural networks by specifying the timesteps at which the input and output variables are measured

$$J_{ki}^{tt'} = \frac{\partial y_k^t}{\partial x_i^{t'}} \tag{3.36}$$

We refer to the resulting four-dimensional matrix as the *sequential Jacobian.*

Figure 3.5 provides a sample plot of a slice through the sequential Jacobian. In general we are interested in observing the sensitivity of an output at one time (for example, at the point when the network makes a label prediction) to the inputs at all times in the sequence. Note that the absolute magnitude of the derivatives is not important. What matters is the *relative* magnitudes of the derivatives to each other, since this determines the degree to which the output is influenced by each input.

Slices like that shown in Figure 3.5 can be calculated with a simple modification of the RNN backward pass described in Section 3.2.2. First, all output delta terms are set to zero except some δ_k^t, corresponding to the time t and output k we are interested to. This term is set equal to its own activation during the forward pass, i.e. $\delta_k^t = y_k^t$. The backward pass is then carried out as usual, and the resulting delta terms at the input layer correspond to the sensitivity of the output to the inputs over time. The other derivatives calculated, e.g. for individual weights or the units in the hidden layer, are also interesting, because they show the response of the output to different parts of the network over time.

We will use similar plots of the sequential Jacobian at various points throughout the thesis as a means of analysing the use of context by RNNs.

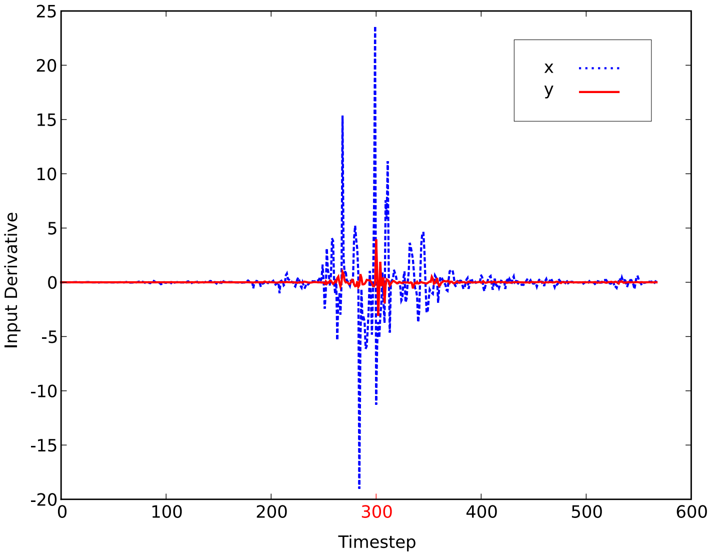


Figure 3.5: **Sequential Jacobian for a bidirectional RNN during an online handwriting recognition task.** The derivatives of a single output unit at time $t = 300$ are evaluated with respect to the two inputs (corresponding to the x and y coordinates of the pen) at all times throughout the sequence. For bidirectional networks, the magnitude of the derivatives typically forms an 'envelope' centred on t. In this case the derivatives remains large for about 100 timesteps before and after t. The magnitudes are greater for the input corresponding to the x co-ordinate (blue line) because this has a smaller normalised variance than the y input (x tends to increase steadily as the pen moves from left to right, whereas y fluctuates about a fixed baseline).

However it should be noted that sensitivity does not correspond directly to contextual importance. For example, the sensitivity may be very large towards an input that never changes, such as a corner pixel in a set of images with a fixed colour background, since the network does not 'expect' to see any change there. However, this pixel will not provide any context information. Also, as shown in Figure 3.5, the sensitivity will be larger for inputs with lower variance, since the network is tuned to smaller changes. But this does not mean that these inputs are more important than those with larger variance.

3.3 Network Training

So far we have discussed how various neural networks can be differentiated with respect to suitable objective functions, and thereby trained with gradient descent. However, to ensure that network training is both effective and tolerably fast, and that it generalises successfully to unseen data, several issues must be considered.

3.3.1 Gradient Descent Algorithms

Most obviously, we need to decide which algorithm to use when following the error gradient. The simplest such algorithm, generally known as *steepest descent* or just *gradient descent*, is to repeatedly take a small, fixed-size step in the direction of the negative error gradient

$$\Delta \mathbf{w}(n) = -\alpha \frac{\partial O}{\partial \mathbf{w}(n)} \tag{3.37}$$

where $0 \leq \alpha \leq 1$ is the *learning rate* and $\mathbf{w}(n)$ represents the vector of weights after the n^{th} weight update.

A major problem with steepest descent is that it easily gets stuck in local minima. This can be mitigated by the addition of a *momentum* term (Plaut et al., 1986; Bishop, 1995), which effectively adds inertia to the motion of the algorithm through weight space, thereby speeding up convergence and helping to escape from local minima

$$\Delta \mathbf{w}(n) = m \Delta \mathbf{w}(n-1) - \alpha \frac{\partial O}{\partial \mathbf{w}(n)} \tag{3.38}$$

where $0 \leq m \leq 1$ is the momentum parameter.

A large number of more sophisticated gradient descent algorithms have been developed (e.g. RPROP; Riedmiller and Braun, 1993, quickprop; Fahlman, 1989, conjugate gradients; Hestenes and Stiefel, 1952; Shewchuk, 1994 and L-BFGS; Byrd et al., 1995) that are generally observed to outperform steepest descent, if the gradient of the entire training set is calculated at once. Such methods are known as *batch learning* algorithms. However, one advantage of steepest descent is that it lends itself naturally to *online learning* (also known as *sequential learning*), where the weight updates are made after each pattern or sequences in the training set. Online steepest descent is often referred to *sequential gradient descent* or *stochastic gradient descent*.

Online learning tends to be more efficient than batch learning when large datasets containing significant redundancy or regularity are used (LeCun et al., 1998b). In addition, the stochasticity of online learning can help to escape from local minima (LeCun et al., 1998b), since the stationary points of the objective function will be different for each training example. The stochasticity can be further increased by randomising the order of the sequences in the training set before each pass through the training set (often referred to as a training *epoch*). The technique of training set randomisation is used throughout this thesis

A recently proposed alternative for online learning is *stochastic meta-descent* (Schraudolph, 2002), which has been shown to give faster convergence and improved results for a variety of neural network tasks. However our attempts to train RNNs with stochastic meta-descent were unsuccessful, and all experiments in this thesis were carried out using online steepest descent with momentum.

3.3.2 Generalisation

Although the objective functions for network training are, of necessity, defined on the training set, the final goal is to achieve the best performance on some previously unseen test data. The issue of whether training set performance carries over to the test set is referred to as *generalisation*, and is of fundamental importance to machine learning (see e.g. Vapnik, 1995; Bishop, 2006). Many methods for improved generalisation have been proposed over the years, usually derived from statistical principles. In this thesis, however, only two of the simplest, empirical methods for generalisation are used: early stopping and training with noise.

Early Stopping

For early stopping, part of the training set is extracted for use as a *validation set.* The objective function is evaluated at regular intervals during training on the validation set, and the weight values giving the lowest validation error are stored as the 'best' for that network. For our experiments we typically evaluate the validation error after every five training epochs, and wait until ten such validations have passed without a new lowest value for the validation error before stopping the network training. The network is never trained on the validation set. In principle, the network should not be evaluated on the test set at all until all training is complete.

During training, the error typically decreases at first on both sets, but after a certain point it begins to rise on the validation set. This behaviour is referred to as *overfitting* on the training data, and is illustrated in Figure 3.6.

Early stopping is perhaps the simplest and most universally applicable method for improved generalisation. However, one drawback of early stopping is that some of the training set has to be sacrificed for the validation set, which can lead to reduced performance, especially if the training set is small. Another problem is that there is no way of determining *a priori* how big the validation set should be. For the experiments in this thesis, we typically use five to ten percent of the training set for validation. Note that the validation set does not have to be an accurate predictor of test set

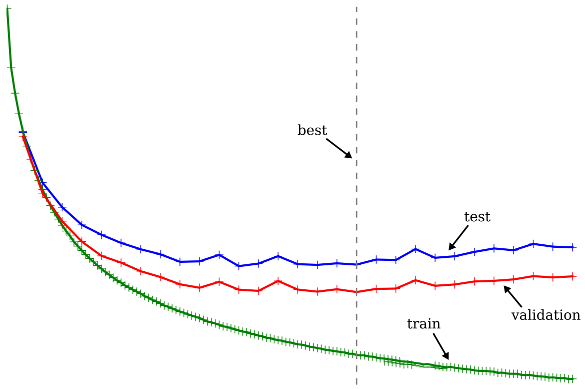


Figure 3.6: **Overfitting on training data.** Initially, network error decreases rapidly on all datasets. Soon however it begins to level off and gradually rise on the validation and test sets. The dashed line indicates the point of best performance on the validation set, which is close, but not identical to the optimal point for the test set. These learning curves were recorded during an offline handwriting recognition task.

performance; it is only important that overfitting begins at approximately the same time on it as on the test set.

Training with Noise

Adding Gaussian noise to the network inputs during training (also referred to as *training with jitter*) is a well-established method for improved generalisation (An, 1996; Koistinen and Holmström, 1991; Bishop, 1995), closely related to statistical regularisation techniques such as ridge regression. Its effect is to artificially enhance the size of the training set by generating new input values sufficiently close to the original ones that a mapping from inputs to targets can still be learned. This tends to decrease performance on the training set, but increase it on the validation and test sets. However, if the variance of the noise is too great, the network fails to learn at all.

Determining exactly how large the variance should be is a major difficulty with training with noise, and appears to be heavily dependent on the dataset. Although various rules of thumb exist, the most reliable method is to set the variance empirically on the validation set. For some of our experiments on speech recognition, we added noise with a variance of fifty to sixty percent of the variance of the original data. For other datasets, however, we found that adding any amount of noise led to reduced performance.

3.3.3 Input Representation

Choosing a suitable representation for the input data is a vital part of any machine learning task. Indeed, in many cases the preprocessing needed to get the data in the required form is more important to the final performance than the algorithm itself. Neural networks tend to be relatively robust to the choice of input representation: for example, in previous work on speech recognition, RNNs were shown to perform almost equally well using a wide range of preprocessing methods (Robinson et al., 1990). Similarly, in Chapter 7 we report excellent results in online handwriting recognition using two very different input representations.

Nonetheless, for acceptable results, the input data presented to an ANN should be both complete (in the sense that it contains all the information required to successfully predict the outputs) and reasonably compact. Although irrelevant inputs are not as much of a problem for neural networks as they are for algorithms suffering from the so-called *curse of dimensionality* (see e.g. Bishop, 2006), having a very high dimensional input space, such as raw images, leads to an excessive number of input weights and poor performance. Beyond that, the choice of input representation is something of a black art. In particular, since neural networks do not make explicit assumptions about the form of the input data, there are no absolute requirements for which representation to choose. However, best performance is generally obtained when the relationship between the inputs and targets is as simple and localised as possible.

One procedure that should be carried out for all input data used for neural network training is to standardise the components of the input vectors to have mean 0 and standard deviation 1 over the training set. That is, first calculate the mean

$$m_i = \frac{1}{|S|}\sum_{x\in S} x_i, \tag{3.39}$$

and standard deviation

$$\sigma_i = \sqrt{\frac{1}{|S|-1}\sum_{x\in S}(m_i - x_i)^2}, \tag{3.40}$$

of each component of the input vector, then calculate the standardised input vectors $\hat{x}$ as follows

$$\hat{x}_i = \frac{x_i - m_i}{\sigma_i} \tag{3.41}$$

This procedure does not alter the information in the training set, but it improves performance by putting the input values in a range more suitable for the standard activation functions (LeCun et al., 1998b). Note that test and validation sets should be standardised with the mean and standard deviation of the training set.

Input standardisation can have a considerable effect on network performance, and was carried out for all experiments in this thesis.

3.3.4 Weight Initialisation

All standard gradient descent algorithms require the network to have small, random, initial weights. For the experiments in this thesis, we initialised the weights with either a flat random distribution in the range $[-0.1, 0.1]$ or a Gaussian distribution with mean 0, standard deviation 0.1. However, we did not find our results to be very sensitive to either the distribution or the range. A consequence of having random initial conditions is that multiple repetitions of each experiment must be carried out in order to determine significance.

Chapter 4

Long Short-Term Memory

As discussed in the previous chapter, an important benefit of recurrent networks is their ability to use contextual information when mapping between input and output sequences. Unfortunately, for standard RNN architectures, the range of context that can be accessed is limited. The problem is that the influence of a given input on the hidden layer, and therefore on the network output, either decays or blows up exponentially as it cycles around the network's recurrent connections. In practice this shortcoming (referred to in the literature as the *vanishing gradient problem*; Hochreiter, 1991; Hochreiter et al., 2001; Bengio et al., 1994) makes it hard for an RNN to learn tasks containing delays of more than about 10 timesteps between relevant input and target events (Hochreiter et al., 2001). The vanishing gradient problem is illustrated schematically in Figure 4.1

Numerous attempts were made in the 1990s to address the problem of vanishing gradients for RNNs. These included non-gradient based training algorithms, such as simulated annealing and discrete error propagation (Bengio et al., 1994), explicitly introduced time delays (Lang et al., 1990; Lin et al., 1996; Plate, 1993) or time constants (Mozer, 1992), and hierarchical sequence compression (Schmidhuber, 1992). However, the most effective solution so far is the Long Short Term Memory (LSTM) architecture (Hochreiter and Schmidhuber, 1997).

This chapter reviews the background material for LSTM, which will be the main RNN architecture used in this thesis. Section 4.1 describes the basic structure of LSTM and explains how it overcomes the vanishing gradient problem. Section 4.3 discusses an approximate and an exact algorithm for calculating the LSTM error gradient. Section 4.4 describes some enhancements to the basic LSTM architecture. Section 4.2 discusses the effect of preprocessing on long range dependencies. Section 4.5 provides all the equations required to train and activate LSTM networks.

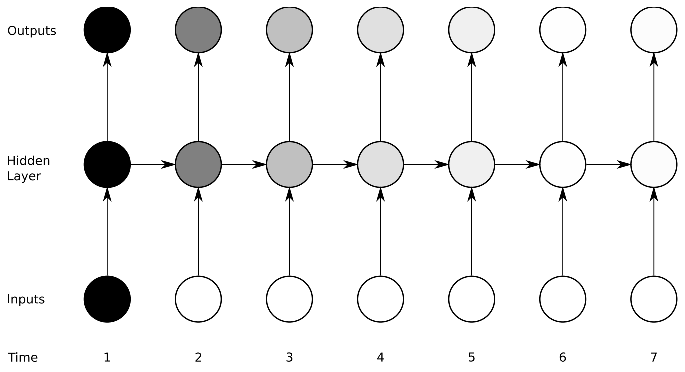


Figure 4.1: **Vanishing gradient problem for RNNs.** The shading of the nodes indicates the sensitivity over time of the network nodes to the input at time one (the darker the shade, the greater the sensitivity). The sensitivity decays exponentially over time as new inputs overwrite the activation of hidden unit and the network 'forgets' the first input.

4.1 The LSTM Architecture

The LSTM architecture consists of a set of recurrently connected subnets, known as memory blocks. These blocks can be thought of as a differentiable version of the memory chips in a digital computer. Each block contains one or more self-connected memory cells and three multiplicative units — the input, output and forget gates — that provide continuous analogues of write, read and reset operations for the cells.

Figure 4.2 provides an illustration of an LSTM memory block with a single cell. An LSTM network is formed exactly like a simple RNN, except that the nonlinear units in the hidden layer are replaced by memory blocks. Indeed, LSTM blocks may be mixed with simple units if required — although this is typically not necessary. Also, as with other RNNs, the hidden layer can be attached to any type of differentiable output layer, depending on the required task (regression, classification etc.).

The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby avoiding the vanishing gradient problem. For example, as long as the input gate remains closed (i.e. has an activation close to 0), the activation of the cell will not be overwritten by the new inputs arriving in the network, and can therefore be made available to the net much later in the sequence, by opening the output gate. The preservation over time of gradient information by LSTM is illustrated in Figure 4.3.

Over the past decade, LSTM has proved successful at a range of synthetic tasks requiring long range memory, including learning context free languages (Gers and Schmidhuber, 2001), recalling high precision real num-

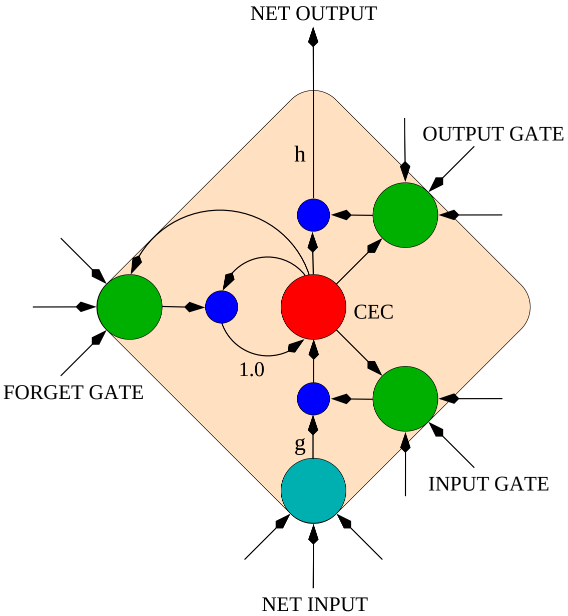


Figure 4.2: **LSTM memory block with one cell.** The internal state of the cell is maintained with a recurrent connection of fixed weight 1.0. The three gates collect activations from inside and outside the block, and control the cell via multiplicative units (small circles). The input and output gates scale the input and output of the cell while the forget gate scales the internal state. The cell input and output activation functions (g and h) are applied at the indicated places.

bers over extended noisy sequences (Hochreiter and Schmidhuber, 1997) and various tasks requiring precise timing and counting (Gers et al., 2002). In particular, it has solved several artificial problems that remain impossible with any other RNN architecture.

Additionally, LSTM has been applied to various real-world problems, such as protein secondary structure prediction (Hochreiter et al., 2007; Chen and Chaudhari, 2005), music generation (Eck and Schmidhuber, 2002), reinforcement learning (Bakker, 2002) and speech recognition (Graves and Schmidhuber, 2005b; Graves et al., 2006) and handwriting recognition (Liwicki et al., 2007; Graves et al., 2008a). As would be expected, its advantages are most pronounced for problems requiring the use of long range contextual information.

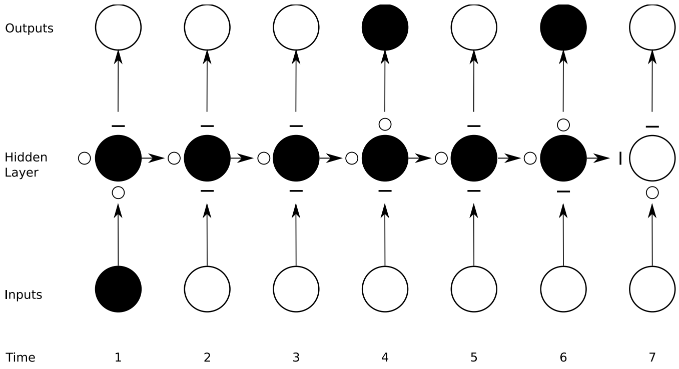


Figure 4.3: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the input unit at time one. The state of the input, forget, and output gate states are displayed below, to the left and above the hidden layer node, which corresponds to a single memory cell. For simplicity, the gates are either entirely open ('O') or closed ('—'). The memory cell 'remembers' the first input as long as the forget gate is open and the input gate is closed, and the sensitivity of the output layer can be switched on and off by the output gate without affecting the cell.

4.2 Influence of Preprocessing

Th above discussion raises an important point about the influence of preprocessing. If we can find a way to transform a task containing long range contextual dependencies into one containing only short-range dependencies before presenting it to a sequence learning algorithm, then architectures such as LSTM become somewhat redundant. For example, a raw speech signal typically consists of 16000 amplitude values per second. Clearly, a great many timesteps would have to be spanned by a sequence learner attempting to label or model an utterance presented in this form. However, by carrying out a series of transforms, averages, rescalings and decorrelations, it is possible to turn such a sequence into one that can be reasonably with modelled by decorrelated, localised function approximators such as the mixtures of diagonal Gaussians used for standard HMMs.

Nonetheless, if such a transform is difficult or unknown, or if we simply wish to get a good result without having to design task-specific preprocessing methods, algorithms capable of learning long time dependencies are useful. In Chapter 7 we evaluate LSTM on an online handwriting task using both raw data and data carefully preprocessed to reduce the need for long range context. We find that LSTM is only slightly more accurate with the hand crafted data.

4.3 Gradient Calculation

Like the ANNs discussed in the last chapter, LSTM is a differentiable function approximator that is typically trained with gradient descent. (Recently, non gradient-based versions of LSTM have also been developed; Wierstra et al., 2005; Schmidhuber et al., 2007, but they are outside the scope of this thesis).

The original LSTM training algorithm (Hochreiter and Schmidhuber, 1997) used an approximate error gradient calculated with a combination of Real Time Recurrent Learning (RTRL; Robinson and Fallside, 1987) and Backpropagation Through Time (BPTT; Williams and Zipser, 1995). The BPTT part was truncated after one timestep, because it was felt that long time dependencies would be dealt with by the memory blocks, and not by the (vanishing) flow of activation around the recurrent connections. Truncating the gradient has the benefit of making the algorithm completely online, in the sense that weight updates can be made after every timestep. This is an important property for tasks such as continuous control or time-series prediction. Additionally, it could only be proven with the truncated gradient that the error flow through the memory cells was constant (Hochreiter and Schmidhuber, 1997).

However, it is also possible to calculate the exact LSTM gradient with BPTT (Graves and Schmidhuber, 2005b). As well as being more accurate than the truncated gradient, the exact gradient has the advantage of being easier to debug, since it can be checked numerically using the technique described in Section 3.1.4. Only the exact gradient is used in this thesis, and the equations for it are provided in Section 4.5.

4.4 Architectural Enhancements

In its original form, LSTM contained only input and output gates. The forget gates (Gers et al., 2000), along with additional peephole weights (Gers et al., 2002) connecting the gates to the memory cell were added later. The purpose of the forget gates was to provide a way for the memory cells to reset themselves, which proved important for tasks that required the network to 'forget' previous inputs. The peephole connections, meanwhile, improved the LSTM's ability to learn tasks that require precise timing and counting of the internal states.

For the purposes of this thesis, we consider only the extended form of LSTM (Gers, 2001) with the forget gates and peepholes added.

Since LSTM is entirely composed of simple multiplication and summation units, and connections between them, it would be straightforward to create further variations of the block architecture. However the current setup seems to be a good general purpose structure for sequence learning tasks.

A further enhancement introduced in this thesis is bidirectional LSTM (BLSTM; Graves and Schmidhuber, 2005a,b), which consists of the standard bidirectional RNN architecture with LSTM used in the hidden layers. BLSTM provides access to long range context in both input directions and, as we will see, consistently outperforms other neural network architectures on sequence labelling tasks. It has proved especially popular in the field of bioinformatics (Chen and Chaudhari, 2005; Thireou and Reczko, 2007).

4.5 LSTM Equations

This section provides the equations for the activation (forward pass) and gradient calculation (backward pass) of an LSTM hidden layer within a recurrent neural network. Only the exact error gradient (Graves and Schmidhuber, 2005b), calculated with backpropagation through time, is presented here. The approximate error gradient (Gers, 2001) is not used in this thesis.

As before, w_{ij} is the weight of the connection from unit i to unit j, the network input to some unit j at time t is denoted a_j^t and the value of the same unit after the activation function has been applied is b_j^t. The LSTM equations are given for a single memory block only. For multiple blocks the calculations are simply repeated for each block, in any order. The subscripts ι, ϕ and ω refer to the input gate, forget gate and output gate respectively. The subscripts c refers to one of the C memory cells. s_c^t is the *state* of cell c at time t (i.e. the activation of the linear cell unit). f is the activation function of the gates, and g and h are respectively the cell input and output activation functions.

Let I be the number of inputs, K be the number of outputs and H be the number of cells in the hidden layer. Note that only the *cell outputs* b_c^t are connected to the other blocks in the layer. The other LSTM activations, such as the states, the cell inputs, or the gate activations, are only visible within the block. We use the index h to refer to cell outputs from other blocks in the hidden layer, exactly as for standard hidden units (indeed normal units and LSTM blocks could be mixed in the same hidden layer, if desired). As with standard RNNs the forward pass is calculated for a length T input sequence $\mathbf{x}$ by starting at $t = 1$ and recursively applying the update equations while incrementing t, and the BPTT backward pass is calculated by starting at $t = T$, and recursively calculating the unit derivatives while decrementing t (see Section 3.2 for details). The final weight derivatives are found by summing over the derivatives at each timestep, as expressed in Eqn. (3.34). Recall that

$$\delta_j^t \overset{\text{def}}{=} \frac{\partial O}{\partial a_j^t}, \tag{4.1}$$

where O is the objective function used for training.

The order in which the equations are calculated during the forward and backward passes is important, and should proceed as specified below. As with standard RNNs, all states and activations are set to zero at $t = 0$, and all δ terms are zero at $t = T + 1$.

4.5.1 Forward Pass

Input Gates

$$a_\iota^t = \sum_{i=1}^{I} w_{i\iota} x_i^t + \sum_{h=1}^{H} w_{h\iota} b_h^{t-1} + \sum_{c=1}^{C} w_{c\iota} s_c^{t-1} \tag{4.2}$$

$$b_\iota^t = f(a_\iota^t) \tag{4.3}$$

Forget Gates

$$a_\phi^t = \sum_{i=1}^{I} w_{i\phi} x_i^t + \sum_{h=1}^{H} w_{h\phi} b_h^{t-1} + \sum_{c=1}^{C} w_{c\phi} s_c^{t-1} \tag{4.4}$$

$$b_\phi^t = f(a_\phi^t) \tag{4.5}$$

Cells

$$a_c^t = \sum_{i=1}^{I} w_{ic} x_i^t + \sum_{h=1}^{H} w_{hc} b_h^{t-1} \tag{4.6}$$

$$s_c^t = b_\phi^t s_c^{t-1} + b_\iota^t g(a_c^t) \tag{4.7}$$

Output Gates

$$a_\omega^t = \sum_{i=1}^{I} w_{i\omega} x_i^t + \sum_{h=1}^{H} w_{h\omega} b_h^{t-1} + \sum_{c=1}^{C} w_{c\omega} s_c^t \tag{4.8}$$

$$b_\omega^t = f(a_\omega^t) \tag{4.9}$$

Cell Outputs

$$b_c^t = b_\omega^t h(s_c^t) \tag{4.10}$$

4.5.2 Backward Pass

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial b_c^t} \qquad \epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial O}{\partial s_c^t} \tag{4.11}$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck}\delta_k^t + \sum_{h=1}^{H} w_{ch}\delta_h^{t+1} \tag{4.12}$$

Output Gates

$$\delta_\omega^t = f'(a_\omega^t)\sum_{c=1}^{C} h(s_c^t)\epsilon_c^t \tag{4.13}$$

States

$$\epsilon_s^t = b_\omega^t h'(s_c^t)\epsilon_c^t + b_\phi^{t+1}\epsilon_s^{t+1} + w_{c\iota}\delta_\iota^{t+1} + w_{c\phi}\delta_\phi^{t+1} + w_{c\omega}\delta_\omega^t \tag{4.14}$$

Cells

$$\delta_c^t = b_\iota^t g'(a_c^t)\epsilon_s^t \tag{4.15}$$

Forget Gates

$$\delta_\phi^t = f'(a_\phi^t)\sum_{c=1}^{C} s_c^{t-1}\epsilon_s^t \tag{4.16}$$

Input Gates

$$\delta_\iota^t = f'(a_\iota^t)\sum_{c=1}^{C} g(a_c^t)\epsilon_s^t \tag{4.17}$$

Chapter 5

Framewise Phoneme Classification

This chapter presents an experimental comparison between various neural network architectures on a framewise phoneme classification task (Graves and Schmidhuber, 2005a,b). Framewise phoneme classification is an example of a segment classification task (see Section 2.3.3). It tests an algorithm's ability to segment and recognise the constituent parts of a speech signal, and requires the use of contextual information. However, phoneme classification alone is not sufficient for continuous speech recognition. As such, the work in this chapter should be regarded as a preliminary study for the work on temporal classification with RNNs presented in Chapter 7.

Context is of particular importance in speech recognition due to phenomena such as *co-articulation*, where the human articulatory system blurs together adjacent sounds in order to produce them rapidly and smoothly. In many cases it is difficult to identify a particular phoneme without knowing the phonemes that occur before and after it. The main result of this chapter is that network architectures capable of accessing more context give better performance in phoneme classification, and are therefore more suitable for speech recognition.

Section 5.1 describes the experimental data and task. Section 5.2 gives an overview of the various neural network architectures and Section 5.3 describes how they are trained, while Section 5.4 presents the experimental results.

5.1 Experimental Setup

The data for the experiments came from the TIMIT corpus (Garofolo et al., 1993) of prompted speech, collected by Texas Instruments. The utterances in TIMIT were chosen to be phonetically rich, and the speakers represent a wide variety of American dialects. The audio data is divided into sentences, each of which is accompanied by a phonetic transcript.

The task was to classify every timestep, or *frame* in audio parlance, according to the phoneme it belonged to. For consistency with the literature, we used the complete set of 61 phonemes provided in the transcriptions. In continuous speech recognition, it is common practice to use a reduced set of phonemes (Robinson, 1991), by merging those with similar sounds, and not separating closures from stops.

The standard TIMIT corpus comes partitioned into training and test sets, containing 3,696 and 1,344 utterances respectively. In total there were 1,124,823 frames in the training set, and 410,920 in the test set. No speakers or sentences exist in both the training and test sets. We used 184 of the training set utterances (chosen randomly, but kept constant for all experiments) as a validation set and trained on the rest. All results for the training and test sets were recorded at the point of lowest error on the validation set.

The following preprocessing, which is standard in speech recognition (Young and Woodland, 2002) was used for the audio data. The data was characterised as a sequence of vectors of 26 coefficients, consisting of twelve Mel-frequency cepstral coefficients (MFCC) plus energy and first derivatives of these magnitudes. First the coefficients were computed every 10ms over 25 ms-long windows. Then a Hamming window was applied, a Mel-frequency filter bank of 26 channels was computed and, finally, the MFCC coefficients were calculated with a 0.97 pre-emphasis coefficient.

5.2 Network Architectures

We used the following five neural network architectures in our experiments (henceforth referred to by the abbreviations in brackets):

- Bidirectional LSTM, with two hidden LSTM layers (forwards and backwards), both containing 93 memory blocks of one cell each (BLSTM)
- Unidirectional LSTM, with one hidden LSTM layer, containing 140 one-cell memory blocks, trained backwards with no target delay, and forwards with delays from 0 to 10 frames (LSTM)
- Bidirectional RNN with two hidden layers containing 185 sigmoid units each (BRNN)
- Unidirectional RNN with one hidden layer containing 275 sigmoid units, trained with target delays from 0 to 10 frames (RNN)
- MLP with one hidden layer containing 250 sigmoid units, and symmetrical time-windows from 0 to 10 frames (MLP)

The hidden layer sizes were chosen to ensure that all networks had roughly the same number of weights W ($\approx 100,000$), thereby providing a fair comparison. Note however that for the MLPs the network grew with the time-window size, and W ranged from 22,061 to 152,061. All networks contained an input layer of size 26 (one for each MFCC coefficient), and an output layer of size 61 (one for each phoneme). The input layers were fully connected to the hidden layers and the hidden layers were fully connected to the output layers. For the recurrent networks, the hidden layers were also fully connected to themselves. The LSTM blocks had the following activation functions: logistic sigmoids in the range $[-2, 2]$ for the input and output activation functions of the cell (g and h in Figure 4.2), and in the range $[0, 1]$ for the gates. The non-LSTM networks had logistic sigmoid activations in the range $[0, 1]$ in the hidden layers. All units were biased.

Figure 5.1 illustrates the behaviour of the different architectures during classification.

5.2.1 Computational Complexity

For all networks, the computational complexity was dominated by the $O(W)$ feedforward and feedback operations. This means that the bidirectional networks and the LSTM networks did not take significantly more time per training epoch than the unidirectional or RNN or (equivalently sized) MLP networks.

5.2.2 Range of Context

Only the bidirectional networks had access to the complete context of the frame being classified (i.e. the whole input sequence). For MLPs, the amount of context depended on the size of the time-window. The results for the MLP with no time-window (presented only with the current frame) give a baseline for performance without context information. However, some context is implicitly present in the window averaging and first-derivatives included in the preprocessor.

Similarly, for unidirectional LSTM and RNN, the amount of future context depended on the size of target delay. The results with no target delay (trained forwards or backwards) give a baseline for performance with context in one direction only.

5.2.3 Output Layers

For the output layers, we used the cross entropy error function and the softmax activation function, as discussed in Sections 3.1.2 and 3.1.3. The softmax function ensures that the network outputs are all between zero and one, and that they sum to one on every timestep. This means they can be interpreted as the posterior probabilities of the phonemes at a given frame, given all the inputs up to the current one (with unidirectional networks) or all the inputs in the whole sequence (with bidirectional networks).

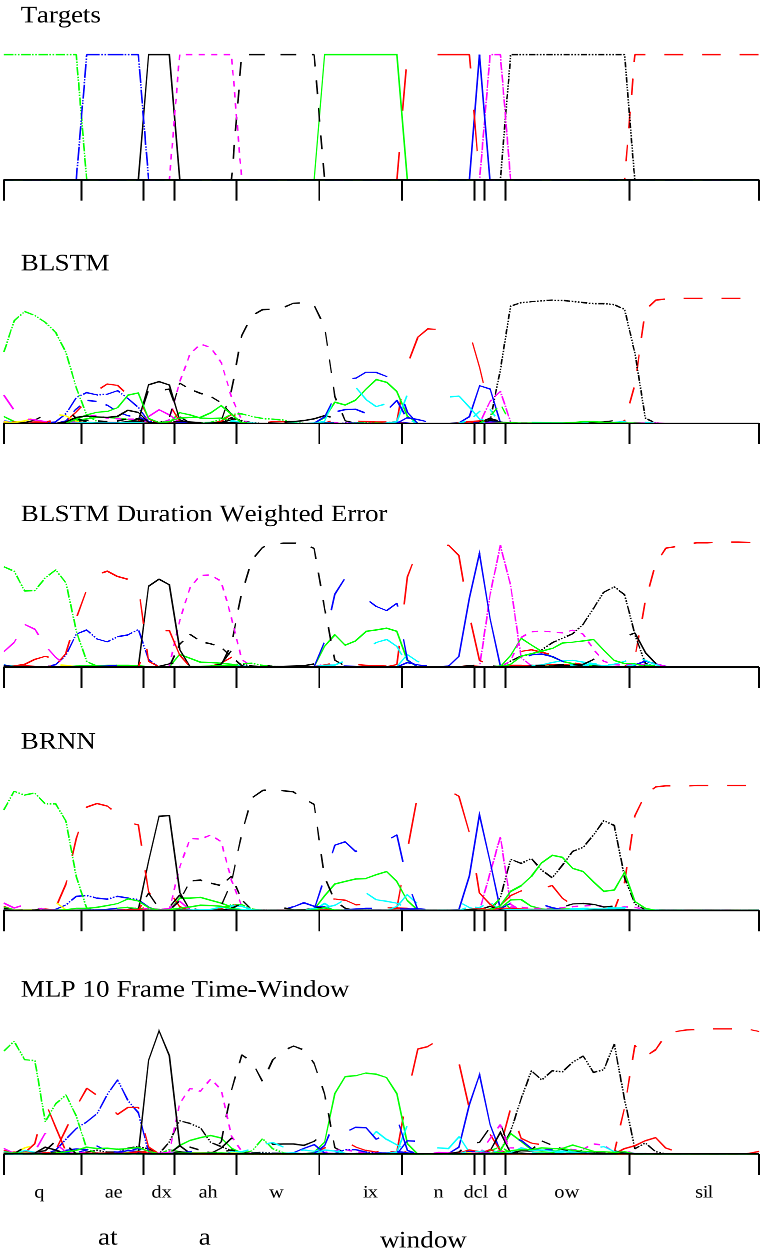


Figure 5.1: **Various networks classifying the excerpt "at a window" from TIMIT.** In general, the networks found the vowels more difficult than the consonants, which in English are more distinct. Adding duration weighted error to BLSTM tends to give better results on short phonemes, (e.g. the closure and stop 'dcl' and 'd'), and worse results on longer ones ('ow'), as expected. Note the jagged trajectories for the MLP; this is presumably because it lacks recurrency to smooth the outputs.

Several alternative error functions have been studied for this task (Chen and Jamieson, 1996). One modification in particular has been shown to have a positive effect on continuous speech recognition. This is to weight the error according to the duration of the current phoneme, ensuring that short phonemes are as significant to training as longer ones. We will return to the issue of weighted errors in the next two chapters.

5.3 Network Training

For all architectures, we calculated the full error gradient using BPTT for each utterance, and trained the weights using online steepest descent with momentum. We kept the same training parameters for all experiments: initial weights chosen from a flat random distribution with range $[-0.1, 0.1]$, a learning rate of 10^{-5} and a momentum of 0.9. As usual, weight updates were carried out at the end of each sequence, and the order of the training set was randomised at the start of each training epoch.

Keeping the training algorithm and parameters constant allowed us to concentrate on the effect of varying the architecture. However it is possible that different training methods would be better suited to different networks.

5.3.1 Retraining

For the experiments with varied time-windows or target delays, we iteratively retrained the networks, instead of starting again from scratch. For example, for LSTM with a target delay of 2, we first trained with delay 0, then took the best network and retrained it (without resetting the weights) with delay 1, then retrained again with delay 2. To find the best networks, we retrained the LSTM networks for 5 epochs at each iteration, the RNN networks for 10, and the MLPs for 20. It is possible that longer retraining times would have given improved results. For the retrained MLPs, we had to add extra (randomised) weights from the input layers, since the input size grew with the time-window.

Although primarily a means to reduce training time, we have also found that retraining improves final performance (Graves et al., 2005a; Beringer, 2004). Indeed, the best result in this paper was achieved by retraining (on the BLSTM network trained with a weighted error function, then retrained with normal cross-entropy error). The benefits presumably come from escaping the local minima that gradient descent algorithms tend to get caught in.

The ability of neural networks to benefit from this kind of retraining touches on the more general issue of transferring knowledge between different tasks (usually known as *meta-learning*, *learning to learn*, or *inductive transfer*) which has been widely studied in the neural network and general machine learning literature (see e.g. Giraud-Carrier et al., 2004).

Table 5.1: **Phoneme classification error rate on TIMIT.** BLSTM result is a mean over seven runs ± std. err.

Network	Training Set	Test Set	Epochs
MLP (no time-window)	46.4%	48.6%	835
MLP (10 frame time-window)	32.4%	36.9%	990
RNN (0 frame delay)	30.1%	35.5%	120
LSTM (0 frame delay)	29.1%	35.4%	15
LSTM (backwards, 0 frame delay)	29.9%	35.3%	15
RNN (3 frame delay)	29.0%	34.8%	140
LSTM (5 frame delay)	22.4%	34.0%	35
BLSTM (Weighted Error)	24.3%	31.1%	15
BRNN	24.0%	31.0%	170
BLSTM	22.6±0.2%	30.2±0.1%	20.1±0.5
BLSTM (retrained)	21.4%	29.8%	17

5.4 Results

Table 5.1 summarises the performance of the different network architectures. For the MLP, RNN and LSTM networks we give both the best results, and those achieved with least contextual information (i.e. with no target delay or time-window). The complete set of results is presented in Figure 5.2.

The most obvious difference between LSTM and the RNN and MLP networks was the number of epochs required for training, as shown in Figure 5.3. In particular, BRNN took more than 8 times as long to converge as BLSTM, despite having more or less equal computational complexity per timestep (see Section 5.2.1). There was a similar time increase between the unidirectional LSTM and RNN networks, and the MLPs were slower still (990 epochs for the best MLP result). A possible explanation for this is that the MLPs and RNNs require more fine-tuning of their weights to access long range contextual information.

As well as being faster, the LSTM networks were also slightly more accurate. However, the final difference in score between BLSTM and BRNN on this task is quite small (0.8%). The fact that the difference is not larger could mean that very long time dependencies are not required for this task.

It is interesting to note how much more prone to overfitting LSTM was than standard RNNs. For LSTM, after only 15-20 epochs the performance on the validation and test sets would begin to fall, while that on the training set would continue to rise (the highest score we recorded on the training set with BLSTM was 86.4%). With the RNNs on the other hand, we never observed a large drop in test set score. This suggests a difference in the way

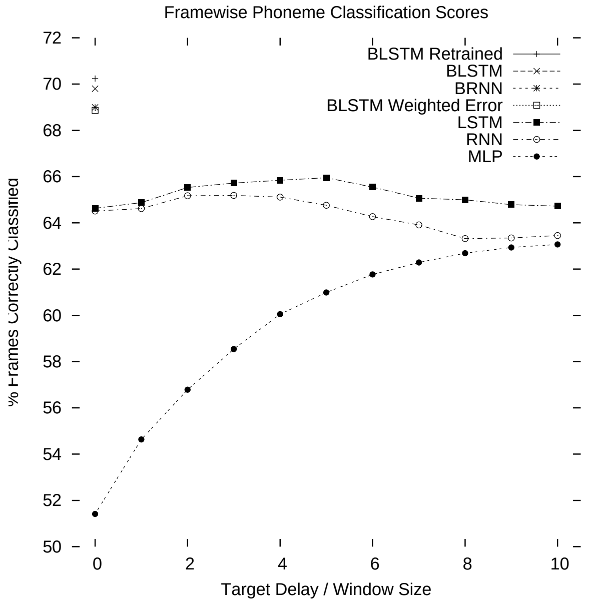


Figure 5.2: **Framewise phoneme classification results on the TIMIT test set.** The number of frames of introduced context (time-window size for MLPs, target delay size for unidirectional LSTM and RNNs) is plotted along the x axis. Therefore the results for the bidirectional networks (clustered around 70%) are plotted at x=0.

the two architectures learn. Given that in the TIMIT corpus no speakers or sentences are shared by the training and test sets, it is possible that LSTM's overfitting was partly caused by its better adaptation to long range regularities (such as phoneme ordering within words, or speaker specific pronunciations) than normal RNNs. If this is true, we would expect a greater distinction between the two architectures on tasks with more training data.

5.4.1 Comparison with Previous Work

Table 5.2 shows how BLSTM compares with the best neural network results previously recorded for this task. Note that Robinson did not quote framewise classification scores; the result for his network was recorded by Schuster, using the original software.

Overall BLSTM outperformed all networks found in the literature, apart from one result quoted by Chen and Jamieson for an RNN (Chen and Jamieson, 1996). However this result is questionable as Chen and Jamieson quote a substantially lower error rate on the test set than on the training set (25.8% and 30.1% respectively). Moreover we were unable to reproduce, or even approach, their scores in our own experiments. For these reasons we

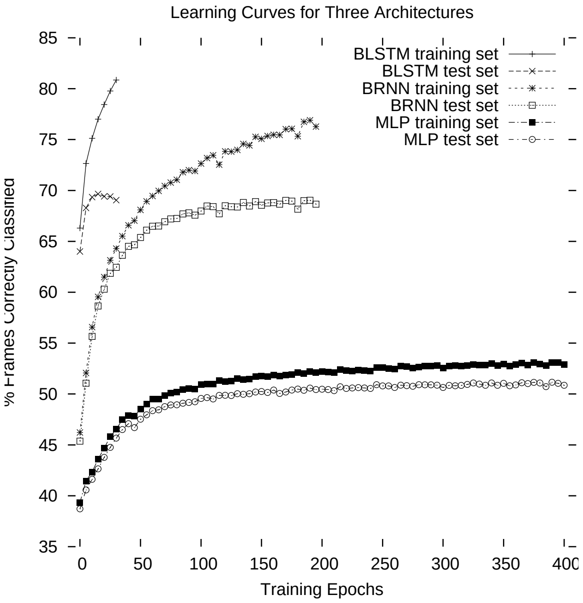


Figure 5.3: **Learning curves on TIMIT for BLSTM, BRNN and MLP with no time-window.** For all experiments, LSTM was much faster to converge than either the RNN or MLP architectures.

have not included their result in the table.

In general, it is difficult to compare network architectures to previous work on TIMIT, owing to the many variations in network training (different gradient descent algorithms, error functions etc.) and in the task itself (different training and test sets, different numbers of phoneme labels, different input preprocessing etc.). That is why we reimplemented all the architectures ourselves.

5.4.2 Effect of Increased Context

As is clear from Figure 5.2 networks with access to more contextual information tended to get better results. In particular, the bidirectional networks were substantially better than the unidirectional ones. For the unidirectional networks, LSTM benefited more from longer target delays than RNNs; this could be due to LSTM's greater facility with long time-lags, allowing it to make use of the extra context without suffering as much from having to remember previous inputs.

Interestingly, LSTM with no time delay returns almost identical results whether trained forwards or backwards. This suggests that the context in both directions is equally important. Figure 5.4 shows how the forward and backward layers work together during classification.

Table 5.2: **Comparison of BLSTM with previous neural network results**

Network	Training Set	Test Set
BRNN (Schuster, 1999)	17.9%	34.9%
RNN (Robinson, 1994)	29.4%	34.7%
BLSTM (retrained)	21.4%	29.8%

For the MLPs, performance increased with time-window size, and it appears that even larger windows would have been desirable. However, with fully connected networks, the number of weights required for such large input layers makes training prohibitively slow.

5.4.3 Weighted Error

The experiment with a weighted error function gave slightly inferior framewise performance for BLSTM (68.9%, compared to 69.7%). However, the purpose of this weighting is to improve overall phoneme recognition, rather than framewise classification. As a measure of its success, if we assume a perfect knowledge of the test set segmentation (which in real-life situations we cannot), and integrate the network outputs over each phoneme, then BLSTM with weighted errors gives a phoneme correctness of 74.4%, compared to 71.2% with normal errors.

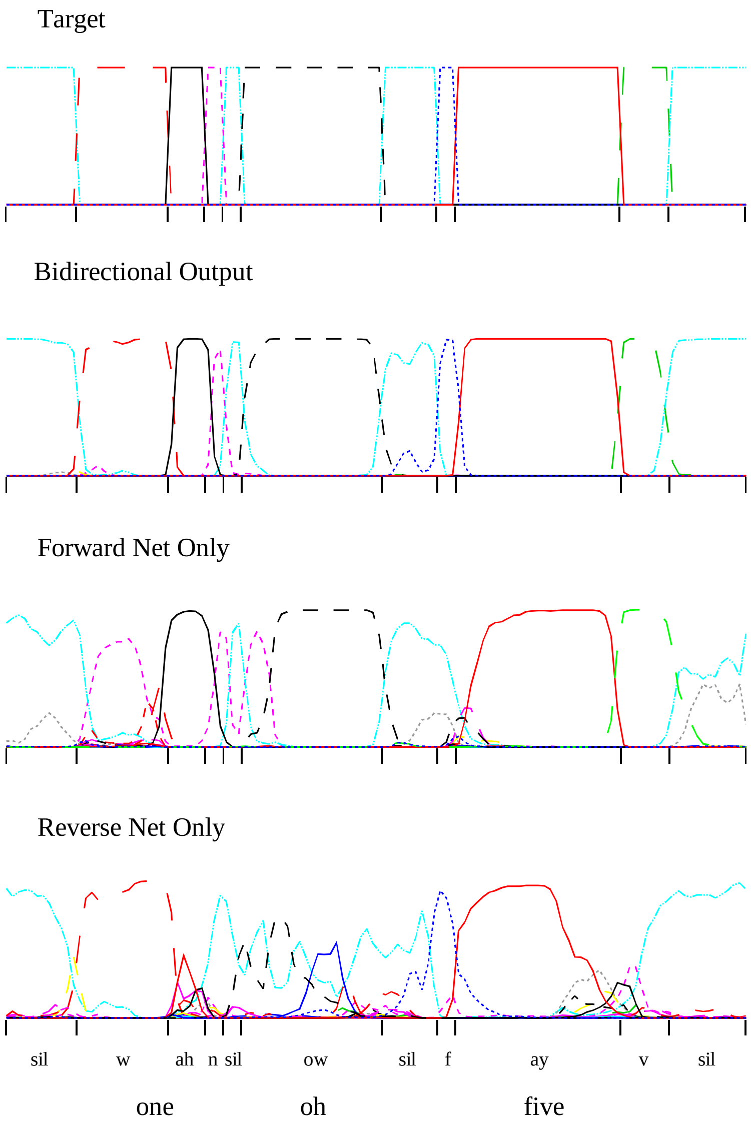


Figure 5.4: **BLSTM network classifying the utterance "one oh five".** The bidirectional output combines the predictions of the forward and backward hidden layers; it closely matches the target, indicating accurate classification. To see how the layers work together, their contributions to the output are plotted separately. As we might expect, the forward layer is more accurate. However there are places where its substitutions ('w'), insertions (at the start of 'ow') and deletions ('f') are corrected by the reverse layer. In addition, both are needed to accurately locate phoneme boundaries, with the reverse layer tending to find the starts and the forward layer tending to find the ends (e.g. 'ay').

Chapter 6

Hidden Markov Model Hybrids

In this chapter LSTM is combined with hidden Markov models to form a hybrid sequence labelling system (Graves et al., 2005b). HMM-ANN hybrids have been extensively studied in the literature, usually with MLPs as the neural network component. The basic idea is to use HMMs to model the long range sequential structure of the data, and neural networks to provide localised classifications. The HMM is able to automatically segment the input sequences during training, and it also provides a principled method for transforming network classifications into label sequences. Unlike the networks described in previous chapters, HMM-ANN hybrids can therefore be directly applied to temporal classification tasks (see Section 2.3.4), such as speech recognition. However, the use of hidden Markov models introduces unnecessary assumptions about the data, and fails to exploit the full potential of RNNs for modelling sequential data. A more powerful technique for temporal classification with RNNs is introduced in the next chapter.

We evaluate the performance of a HMM-BLSTM hybrid on a phoneme recognition experiment, and find that it outperforms both a standard HMM and a hybrid using unidirectional LSTM. This suggests that the advantages of using network architectures with increased access to context carry over to temporal classification.

Section 6.1 reviews the previous work on hybrid HMM-ANN systems. Section 6.2 presents experimental results on a phoneme recognition task.

6.1 Background

Hybrids of hidden Markov models (HMMs) and artificial neural networks (ANNs) were proposed by several researchers in the 1990s as a way of overcoming the drawbacks of HMMs (Bourlard and Morgan, 1994; Bengio, 1993; Renals et al., 1993; Robinson, 1994; Bengio, 1999). The introduction of

ANNs was intended to provide more discriminant training, improved modelling of phoneme duration, richer, nonlinear function approximation, and perhaps most importantly, increased use of contextual information.

In their simplest form, hybrid methods used HMMs to align the segment classifications provided by the ANNs into a temporal classification of the entire label sequence (Renals et al., 1993; Robinson, 1994). In other cases ANNs were used to estimate transition or emission probabilities for HMMs (Bourlard and Morgan, 1994), to re-score the N-best HMM labellings according to localised classifications (Zavaliagkos et al., 1993), and to extract observation features that can be more easily modelled by an HMM (Bengio et al., 1995, 1992). In this chapter we focus on the simplest case, since there is no compelling evidence that the more complex systems give better performance.

Although most hybrid HMM-ANN research focused on speech recognition, the framework is equally applicable to other sequence labelling tasks, such as online handwriting recognition (Bengio et al., 1995).

The two components in a the hybrid can be trained independently, but many authors have proposed methods for combined optimisation (Bengio et al., 1992; Bourlard et al., 1996; Hennebert et al., 1997; Trentin and Gori, 2003) which typically yields improved results. In this chapter we follow an iterative approach, where the alignment provided by the HMM is used to successively retrain the neural network (Robinson, 1994).

A similar, but more general, framework for combining neural networks with other sequential algorithms is provided by *graph transformer networks* (LeCun et al., 1997, 1998a; Bottou and LeCun, 2005). The different modules of a graph transformer network perform distinct tasks, such as segmentation, recognition and imposing grammatical constaints. The modules are connected by *transducers*, which provide differentiable sequence to sequence maps, and allow for global, gradient based learning.

Most hybrid HMM-ANN systems use multilayer perceptrons, typically with a time-window to provide context, for the neural network component. However there has also been considerable interest in the use of RNNs (Robinson, 1994; Neto et al., 1995; Kershaw et al., 1996; Senior and Robinson, 1996). Given that the main purpose of the ANN is to introduce contextual information, RNNs seem a natural choice. However, their advantages over MLPs remained inconclusive in early work (Robinson et al., 1993).

Despite a flurry of initial interest, hybrid HMM-ANN approaches never achieved a great improvement over conventional HMMs, especially considering the added complexity of designing and training them. As a consequence, most current speech and handwriting recognition systems are based on conventional HMMs. We believe this is in part due to the limitations of the standard neural network architectures. In what follows we evaluate the influence on hybrid performance of using LSTM and bidirectional LSTM as the network architectures.

6.2 Experiment: Phoneme Recognition

The experiment was carried out on the TIMIT speech corpus (Garofolo et al., 1993). The data preparation and division into training, test and validation sets was identical to that described in Section 5.1. However the task was different. Instead of classifying phonemes at every frame, the goal was to output the complete phonetic transcription of the input sequences. Correspondingly, the error measure was the phoneme error rate (label error rate with phonemes as labels — see Section 2.3.4) instead of the frame error rate.

We evaluate the performance of standard HMMs with and without context dependent phoneme models, and hybrid systems using BLSTM, LSTM and BRNNs. We also evaluate the effect of using a weighted error signal (Chen and Jamieson, 1996), as described in Section 5.2.3.

6.2.1 Experimental Setup

Traditional HMMs were developed with the HTK Speech Recognition Toolkit (http://htk.eng.cam.ac.uk/). Both context independent (mono-phone) and context dependent (triphone) models were trained and tested. Both were left-to-right models with three states. Models representing silence (h#, pau, epi) included two extra transitions, from the first to the final state and vice-versa, to make them more robust. Observation probabilities were modelled by eight Gaussian mixtures.

Sixty-one context-independent models and 5491 tied context-dependent models were used. Context-dependent models for which the left/right context coincide with the central phoneme were included since they appear in the TIMIT transcription (e.g. “my eyes” is transcribed as /m ay ay z/). During recognition, only sequences of context-dependent models with matching context were allowed.

In order to make a fair comparison of the acoustic modelling capabilities of the traditional and hybrid systems, no linguistic information or probabilities of partial phoneme sequences were included in the system.

For the hybrid systems, the following networks were used: unidirectional LSTM, BLSTM, and BLSTM trained with weighted error. 61 models of one state each with a self-transition and an exit transition probability were trained using Viterbi-based forced-alignment. The training set transcription was used to provide initial estimates of transition and prior probabilities. The training set transcription was also used to initially train the networks. In fact the networks trained for the framewise classification experiments in Chapter 5 were simply re-used; therefore the network architectures and parameters were identical to those described in Sections 5.2 and 5.3. The HMM was trained using the network output probabilities divided by prior probabilities to obtain observation likelihoods. The alignment provided by

Table 6.1: **Phoneme error rate (PER) on TIMIT.** Hybrid results are means over 5 runs, ± standard error. All differences were significant ($p < 0.01$).

System	Parameters	PER
Context-independent HMM	80 K	38.85%
Context-dependent HMM	>600 K	35.21%
HMM-LSTM	100 K	39.6 ± 0.08%
HMM-BLSTM	100 K	33.84 ± 0.06%
HMM-BLSTM (weighted error)	100 K	31.57 ± 0.06%

the trained HMM was then used to define a new framewise training signal for the neural networks, and the whole process was repeated until convergence.

For both the traditional and hybrid system, an insertion penalty was estimated on the validation set and applied during recognition.

6.2.2 Results

From Table 6.1, we can see that HMM-BLSTM hybrids outperformed both context-dependent and context-independent HMMs. We can also see that BLSTM gave better performance than unidirectional LSTM, in agreement with the results in Chapter 5. The best result was achieved with the HMM-BLSTM hybrid using a weighted error signal. This is what we would expect, since the effect of error weighting is to make all phonemes equally significant, as they are to the phoneme error rate.

Note that the hybrid systems had considerably fewer free parameters than the context-dependent HMM. This is a consequence of the high number of states required for HMMs to model contextual dependencies.

Note also that the networks in the hybrid systems were initially trained with hand segmented training data. Although the experiments could have been carried out with a flat segmentation, this would have probably led to inferior results.

Chapter 7

Connectionist Temporal Classification

This chapter introduces *connectionist temporal classification* (CTC; Graves et al., 2006; Fernández et al., 2007a; Liwicki et al., 2007; Graves et al., 2008a,b), a novel output layer for temporal classification with RNNs. Unlike the approach described in the previous chapter, CTC models all aspects of the sequence with a single RNN, and does not require the network to be combined with hidden Markov models. It also does not require presegmented training data, or external post-processing to extract the label sequence from the network outputs. We carry out several experiments on speech and handwriting recognition, and find that CTC networks with the BLSTM architecture outperform HMMs and HMM-RNN hybrids, as well as more recent sequence learning algorithms such as large margin HMMs (Sha and Saul, 2006). For a widely studied phoneme recognition benchmark on the TIMIT speech corpus, CTC's performance is on a par with the best previous systems, even though these were specifically tuned for the task.

Section 7.1 introduces CTC and motivates its use for temporal classification tasks. Section 7.2 defines the mapping from CTC outputs onto label sequences, Section 7.3 provides an algorithm for efficiently calculating the probability of a given label sequence, and Section 7.4 combines these to derive the CTC objective function. Section 7.5 describes methods for decoding with CTC. Experimental results are presented in Section 7.6, and a discussion of the differences between CTC networks and HMMs is given in Section 7.7.

7.1 Motivation

In 1994, Bourlard and Morgan (Bourlard and Morgan, 1994, chap. 5) identified the following reason for the failure of purely connectionist approaches to continuous speech recognition:

> There is at least one fundamental difficulty with supervised training of a connectionist network for continuous speech recognition: a target function must be defined, even though the training is done for connected speech units where the segmentation is generally unknown.

In other words, neural networks, at least with the standard objective functions, require separate training targets for every segment or timestep in the input sequence. This has two important consequences. Firstly, it means that the training data must be presegmented to provide the targets. Secondly, since the network only outputs local classifications, the global aspects of the sequence (e.g. the likelihood of two labels appearing together) must be modelled externally. Indeed, without some form of post-processing the final label sequence cannot reliably be inferred at all.

In Chapter 6 we showed how RNNs could be used for temporal classification by combining them with HMMs in hybrid systems. However, as well as inheriting the disadvantages of HMMs (non-discriminative training, unrealistic dependency assumptions etc. — see Section 7.7 for an in-depth discussion), hybrid systems do not exploit the potential of RNNs for global sequence modelling. It therefore seems preferable to train RNNs directly for temporal classification tasks.

Connectionist temporal classification (CTC) achieves this by allowing the network to make label predictions at any point in the input sequence, so long as the overall sequence of labels is correct. This removes the need for presegmented data, since the alignment of the labels with the input is no longer important. Moreover, CTC directly estimates the probabilities of the complete label sequences, which means that no external post-processing is required to use the network as a temporal classifier.

Figure 7.1 illustrates the difference between CTC and framewise classification, when applied to a speech signal.

7.2 From Outputs to Labellings

For a sequence labelling task where the labels are drawn from an alphabet L, CTC consists of a softmax output layer (Bridle, 1990) with one more unit than there are labels in L. The activations of the first $|L|$ units are used to estimate the probabilities of observing the corresponding labels at particular times, conditioned on the training set and the current input sequence. The activation of the extra unit estimates the probability of observing a ‘blank’, or no label. Together, the outputs give the joint conditional probability of all labels at all timesteps. The conditional probability of any one label sequence can then be found by summing over the corresponding joint probabilities.

More formally, the activation y_k^t of output unit k at time t is interpreted as the probability of observing label k (or blank if $k = |L| + 1$) at time t,

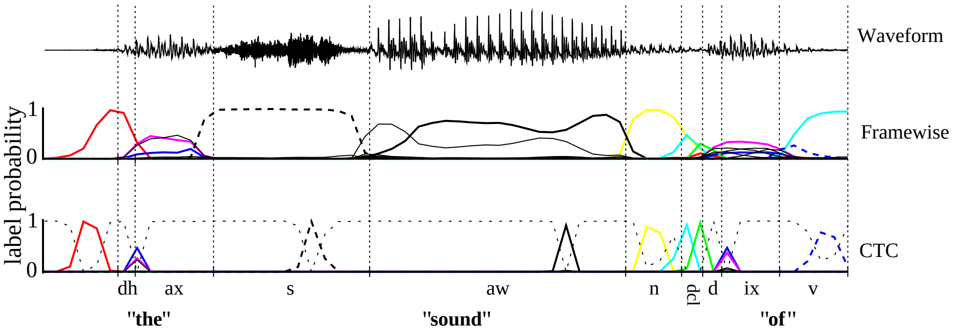


Figure 7.1: **CTC and framewise classification networks applied to a speech signal.** The shaded lines are the output activations, corresponding to the probabilities of observing phonemes at particular times. The CTC network predicts only the sequence of phonemes (typically as a series of spikes, separated by 'blanks', or null predictions), while the framewise network attempts to align them with the manual segmentation (vertical lines).

given the length T input sequence $\mathbf{x}$ and the training set S. Together, these probabilities estimate the distribution over the elements $\pi \in L'^T$ (where L'^T is the set of length T sequences over the alphabet $L' = L \cup \{blank\}$)

$$p(\pi|\mathbf{x}, S) = \prod_{t=1}^{T} y^t_{\pi_t} \tag{7.1}$$

Implicit in the above formulation is the assumption that the probabilities of the labels at each timestep are conditionally independent, given $\mathbf{x}$ and S. Provided S is large enough this is in principle true, since the likelihood of all label sequences and subsequences can be learned directly from the data. However it can be advantageous to condition on inter-label probabilities as well, especially for tasks involving labels with known or partially known relationships (such as words in human language). We discuss this matter further in Section 7.5.3, where we introduce a CTC decoding algorithm that includes explicit word-word transition probabilities.

CTC can be used with any RNN architecture. However, because the label probabilities are conditioned on the entire input sequence, and not just on the part occurring before the label is emitted, bidirectional RNNs are preferred.

From now on, we refer to the elements $\pi \in L'^T$ as *paths*, and we drop the explicit dependency of the output probabilities on S to simplify notation.

The next step is to define a many-to-one map $\mathcal{B} : L'^T \mapsto L^{\leq T}$, from the set of paths onto the set $L^{\leq T}$ of possible labellings (i.e. the set of sequences of length less than or equal to T over the original label alphabet L). We do this by removing first the repeated labels and then the blanks from the paths. For example $\mathcal{B}(a - ab-) = \mathcal{B}(-aa - -abb) = aab$. Intuitively, this corresponds to outputting a new label when the network either switches from predicting no label to predicting a label, or from predicting one label to another. Since the paths are mutually exclusive, the conditional probability of some labelling $\mathbf{l} \in L^{\leq T}$ can be calculated by summing the probabilities of all the paths mapped onto it by $\mathcal{B}$:

$$p(\mathbf{l}|\mathbf{x}) = \sum_{\pi \in \mathcal{B}^{-1}(\mathbf{l})} p(\pi|\mathbf{x}). \tag{7.2}$$

This 'collapsing together' of different paths onto the same labelling is what allows CTC to use unsegmented data, because it removes the requirement of knowing *where* in the input sequence the labels occur. In theory, it also makes CTC unsuitable for tasks where the location of the labels must be determined. However in practice CTC networks tend to output labels close to where they occur in the input sequence. In Section 7.6.3 an experiment is presented in which both the labels and their approximate positions are successfully predicted by CTC.

7.2.1 Role of the Blank Labels

In our original formulation of CTC, there were no blank labels, and $\mathcal{B}(\pi)$ was simply π with repeated labels removed. This led to two problems. Firstly, the same label could not appear twice in a row, since transitions only occurred when π passed between different labels. And secondly, the network was required to continue predicting one label until the next began, which is a burden in tasks where the input segments corresponding to consecutive labels are widely separated by unlabelled data (e.g. speech recognition, where there are often pauses or non-speech noises between the words in an utterance).

7.3 CTC Forward-Backward Algorithm

So far we have defined the conditional probabilities $p(\mathbf{l}|\mathbf{x})$ of the possible label sequences. Now we need an efficient way of calculating them. At first sight Eqn. (7.2) suggests this will be problematic: the sum is over all paths corresponding to a given labelling, and for a labelling of length s and an input sequence of length T, there are $2^{T-s^2+s(T-3)}3^{(s-1)(T-s)-2}$ of these.

Fortunately the problem can be solved with a dynamic-programming algorithm similar to the forward-backward algorithm for HMMs (Rabiner, 1989). The key idea is that the sum over paths corresponding to a labelling $\mathbf{l}$ can be broken down into an iterative sum over paths corresponding to prefixes of that labelling.

To allow for blanks in the output paths, we consider a modified label sequence $\mathbf{l}'$, with blanks added to the beginning and the end of $\mathbf{l}$, and inserted between every pair of consecutive labels. The length of $\mathbf{l}'$ is therefore $2|\mathbf{l}| + 1$. In calculating the probabilities of prefixes of $\mathbf{l}'$ we allow all transitions between blank and non-blank labels, and also those between any pair of distinct non-blank labels.

For a labelling $\mathbf{l}$, we define the *forward variable* $\alpha_t(s)$ as the summed probability of all paths whose length t prefixes are mapped by $\mathcal{B}$ onto the length $s/2$ prefix of $\mathbf{l}$, i.e.

$$\alpha_t(s) = P(\pi_{1:t} : \mathcal{B}(\pi_{1:t}) = \mathbf{l}_{1:s/2}, \pi_t = l'_s|\mathbf{x}) = \sum_{\substack{\pi: \\ \mathcal{B}(\pi_{1:t})=\mathbf{l}_{1:s/2}}} \prod_{t'=1}^{t} y^{t'}_{\pi_{t'}} \tag{7.3}$$

where, for some sequence $\mathbf{s}$, $\mathbf{s}_{a:b}$ is the subsequence $(\mathbf{s}_a, \mathbf{s}_{a+1}, ..., \mathbf{s}_{b-1}, \mathbf{s}_b)$, and $s/2$ is rounded down to an integer value. As we will see, $\alpha_t(s)$ can be calculated recursively from $\alpha_{t-1}(s)$, $\alpha_{t-1}(s-1)$ and $\alpha_{t-1}(s-2)$.

Given the above formulation, the probability of $\mathbf{l}$ can be expressed as the sum of the forward variables with and without the final blank at time T

$$p(\mathbf{l}|\mathbf{x}) = \alpha_T(|\mathbf{l}'|) + \alpha_T(|\mathbf{l}'| - 1) \tag{7.4}$$

Allowing all paths to start with either a blank (b) or the first symbol in $\mathbf{l}$ (l_1), we get the following rules for initialisation

$$\alpha_1(1) = y^1_b \tag{7.5}$$

$$\alpha_1(2) = y^1_{l_1} \tag{7.6}$$

$$\alpha_1(s) = 0, \ \forall s > 2 \tag{7.7}$$

and recursion

$$\alpha_t(s) = y^t_{l'_s} \begin{cases} \sum_{i=s-1}^{s} \alpha_{t-1}(i) & \text{if } l'_s = b \text{ or } l'_{s-2} = l'_s \\ \sum_{i=s-2}^{s} \alpha_{t-1}(i) & \text{otherwise,} \end{cases} \tag{7.8}$$

where

$$\alpha_t(s) = 0 \ \forall s < |\mathbf{l}'| - 2(T-t) - 1, \tag{7.9}$$

because these variables correspond to states for which there are not enough timesteps left to complete the sequence (the unconnected circles in the top right of Figure 7.2), and

$$\alpha_t(0) = 0 \ \forall t \tag{7.10}$$

The *backward variables* $\beta_t(s)$ are defined as the summed probability of all paths whose suffixes starting at t map onto the suffix of $\mathbf{l}$ starting at label $s/2$

$$\beta_t(s) = P(\pi_{t+1:T} : \mathcal{B}(\pi_{t:T}) = \mathbf{l}_{s/2:|\mathbf{l}|}, \pi_t = l'_s|\mathbf{x}) = \sum_{\substack{\pi: \\ \mathcal{B}(\pi_{t:T})=\mathbf{l}_{s/2:|\mathbf{l}|}}} \prod_{t'=t+1}^{T} y^{t'}_{\pi_{t'}} \tag{7.11}$$

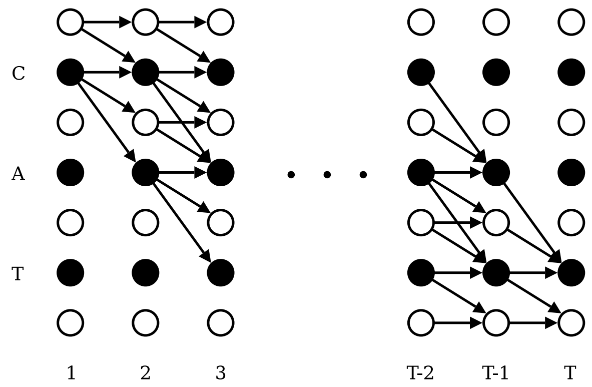


Figure 7.2: **CTC forward-backward algorithm.** Black circles represent labels, and white circles represent blanks. Arrows signify allowed transitions. Forward variables are updated in the direction of the arrows, and backward variables are updated against them.

The rules for initialisation and recursion of the backward variables are as follows

$$\beta_T(|\mathbf{l}'|) = 1 \tag{7.12}$$

$$\beta_T(|\mathbf{l}'| - 1) = 1 \tag{7.13}$$

$$\beta_T(s) = 0,\ \forall s < |\mathbf{l}'| - 1 \tag{7.14}$$

$$\beta_t(s) = \begin{cases} \sum_{i=s}^{s+1} \beta_{t+1}(i) y^t_{l'_i} & \text{if } l'_s = b \text{ or } l'_{s+2} = l'_s \\ \sum_{i=s}^{s+2} \beta_{t+1}(i) y^t_{l'_i} & \text{otherwise} \end{cases} \tag{7.15}$$

where

$$\beta_t(s) = 0\ \forall s > 2t, \tag{7.16}$$

as shown by the unconnected circles in the bottom left of Figure 7.2, and

$$\beta_t(|\mathbf{l}'| + 1) = 0\ \forall t \tag{7.17}$$

7.3.1 Log Scale

In practice, the above recursions will soon lead to underflows on any digital computer. A good way to avoid this is to work in the log scale, and only exponentiate to find the true probabilities at the end of the calculation. A useful equation in this context is

$$\ln(a + b) = \ln a + ln\left(1 + e^{\ln b - \ln a}\right), \tag{7.18}$$

which allows the forward and backward variables to be summed while remaining in the log scale. Note that rescaling the variables at every timestep (Rabiner, 1989) is less robust, and can fail for very long sequences.

7.4 CTC Objective Function

So far we have described how an RNN with a CTC output layer can be used for temporal classification. In this section we derive an objective function that allows CTC networks to be trained with gradient descent.

Like the standard neural network objective functions (see Section 3.1.3), the CTC objective function is derived from the principle of maximum likelihood. Because the objective function is differentiable, its derivatives with respect to the network weights can be calculated with backpropagation through time (Section 3.2.2), and the network can then be trained with any gradient-based nonlinear optimisation algorithm (Section 3.3.1).

As usual, The objective function O is defined as the negative log probability of correctly labelling the entire training set:

$$O = -\ln\left(\prod_{(\mathbf{x},\mathbf{z})\in S} p(\mathbf{z}|\mathbf{x})\right) = -\sum_{(\mathbf{x},\mathbf{z})\in S} \ln p(\mathbf{z}|\mathbf{x}) \tag{7.19}$$

To find the gradient of (7.19), we first differentiate with respect to the network outputs y_k^t during some training example $(\mathbf{x}, \mathbf{z})$

$$\frac{\partial O}{\partial y_k^t} = -\frac{\partial \ln p(\mathbf{z}|\mathbf{x})}{\partial y_k^t} = -\frac{1}{p(\mathbf{z}|\mathbf{x})}\frac{\partial p(\mathbf{z}|\mathbf{x})}{\partial y_k^t} \tag{7.20}$$

We now show how the algorithm of Section 7.3 can be used to calculate (7.20).

The key point is that the product of the forward and backward variables at a given s and t is the summed probability of all paths mapped onto $\mathbf{z}$ by $\mathcal{B}$ that go through symbol s in $\mathbf{z}'$ at time t. Setting $\mathbf{l} = \mathbf{z}$, (7.3) and (7.11) give us

$$\alpha_t(s)\beta_t(s) = \sum_{\substack{\pi\in\mathcal{B}^{-1}(\mathbf{z}):\\ \pi_t=\mathbf{z}'_s}} \prod_{t=1}^{T} y_{\pi_t}^t \tag{7.21}$$

Substituting from (7.1) we get

$$\alpha_t(s)\beta_t(s) = \sum_{\substack{\pi\in\mathcal{B}^{-1}(\mathbf{z}):\\ \pi_t=\mathbf{z}'_s}} p(\pi|\mathbf{x}) \tag{7.22}$$

From (7.2) we can see that this is the portion of the total probability $p(\mathbf{z}|\mathbf{x})$ due to those paths going through $\mathbf{z}'_s$ at time t, as claimed. For any t, we can therefore sum over all s to get

$$p(\mathbf{z}|\mathbf{x}) = \sum_{s=1}^{|\mathbf{z}'|} \alpha_t(s)\beta_t(s) \tag{7.23}$$

To differentiate this with respect to y^t_k, we need only consider those paths going through label k at time t, since the network outputs do not influence each other. Noting that the same label (or blank) may occur several times in a single labelling, we define the set of positions where label k occurs in $\mathbf{z}'$ as $lab(\mathbf{z}, k) = \{s : \mathbf{z}'_s = k\}$, which may be empty. Observing from (7.21) that

$$\frac{\partial \alpha_t(s)\beta_t(s)}{\partial y^t_k} = \begin{cases} \frac{\alpha_t(s)\beta_t(s)}{y^t_k} & \text{if } k \text{ occurs in } \mathbf{z}' \\ 0 & \text{otherwise,} \end{cases} \tag{7.24}$$

we can differentiate (7.23) to get

$$\frac{\partial p(\mathbf{z}|\mathbf{x})}{\partial y^t_k} = \frac{1}{y^t_k} \sum_{s \in lab(\mathbf{z},k)} \alpha_t(s)\beta_t(s). \tag{7.25}$$

and substitute this into (7.20) to get

$$\frac{\partial O}{\partial y^t_k} = -\frac{1}{p(\mathbf{z}|\mathbf{x}) y^t_k} \sum_{s \in lab(\mathbf{z},k)} \alpha_t(s)\beta_t(s). \tag{7.26}$$

Finally, to backpropagate the gradient through the output layer, we need the objective function derivatives with respect to the outputs a^t_k *before* the activation function is applied

$$\frac{\partial O}{\partial a^t_k} = -\sum_{k'} \frac{\partial O}{\partial y^t_{k'}} \frac{\partial y^t_{k'}}{\partial a^t_k} \tag{7.27}$$

where k' ranges over all the output units. Recalling that for softmax outputs

$$\begin{aligned} y^t_k &= \frac{e^{a^t_k}}{\sum_{k'} e^{a^t_{k'}}} \\ \Rightarrow \frac{\partial y^t_{k'}}{\partial a^t_k} &= y^t_{k'}\delta_{kk'} - y^t_{k'} y^t_k, \end{aligned} \tag{7.28}$$

we can substitute (7.28) and (7.26) into (7.27) to obtain

$$\frac{\partial O}{\partial u^t_k} = y^t_k - \frac{1}{p(\mathbf{z}|\mathbf{x})} \sum_{s \in lab(\mathbf{z},k)} \alpha_t(s)\beta_t(s), \tag{7.29}$$

which is the 'error signal' received by the network during training, as illustrated in Figure 7.3. Note that, for numerical stability, it is advised to recalculate the $p(\mathbf{z}|\mathbf{x})$ term for every t using (7.23).

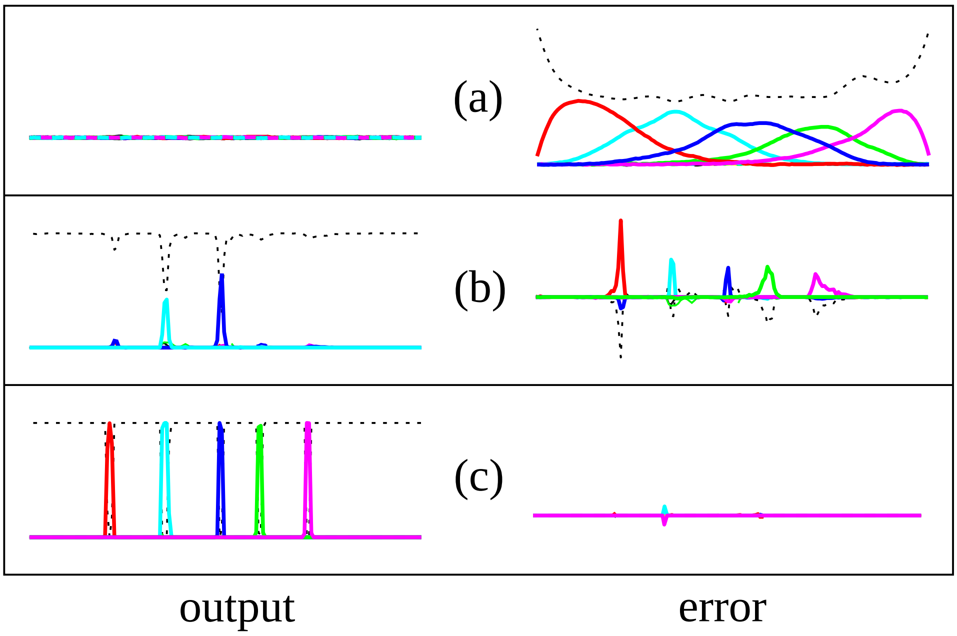


Figure 7.3: **Evolution of the CTC error signal during training.** The left column shows the output activations for the same sequence at various stages of training (the dashed line is the 'blank' unit); the right column shows the corresponding error signals. Errors above the horizontal axis act to increase the corresponding output activation and those below act to decrease it. (a) Initially the network has small random weights, and the error is determined by the target sequence only. (b) The network begins to make predictions and the error localises around them. (c) The network strongly predicts the correct labelling and the error virtually disappears.

7.5 Decoding

Once the network is trained, we would ideally label some unknown input sequence $\mathbf{x}$ by choosing the most probable labelling $\mathbf{l}^*$:

$$\mathbf{l}^* = \arg\max_{\mathbf{l}} p(\mathbf{l}|\mathbf{x}) \tag{7.30}$$

Using the terminology of HMMs, we refer to the task of finding this labelling as *decoding*. Unfortunately, we do not know of a general, tractable decoding algorithm for CTC. However there are two approximate methods that give good results in practice.

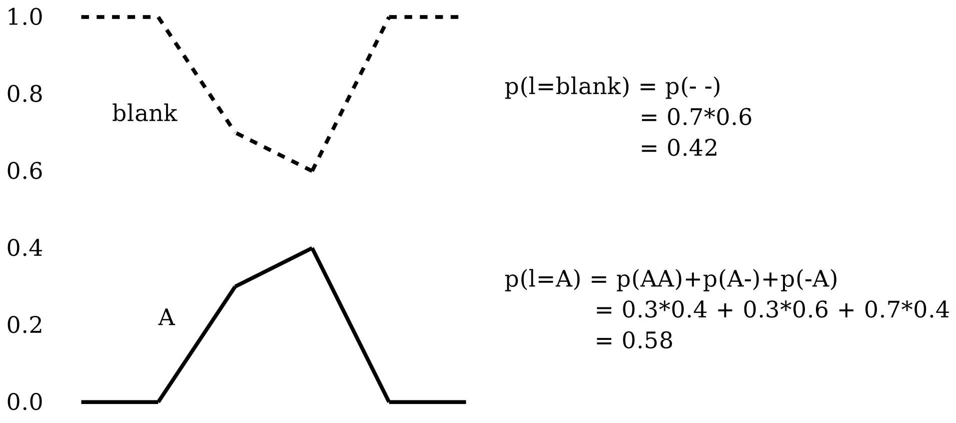


Figure 7.4: **Problem with best path decoding.** The single most probable path contains no labels, and best path decoding therefore outputs the labelling 'blank'. However the combined probabilities of the paths corresponding to the labelling 'A' is greater.

7.5.1 Best Path Decoding

The first method, which refer to as *best path decoding*, is based on the assumption that the most probable path corresponds to the most probable labelling

$$\mathbf{l}^* \approx \mathcal{B}(\pi^*) \tag{7.31}$$
$$\text{where } \pi^* = \arg\max_{\pi} p(\pi|\mathbf{x}).$$

Best path decoding is trivial to compute, since π^* is just the concatenation of the most active outputs at every timestep. However it can lead to errors, particularly if a label is weakly predicted for several consecutive timesteps (see Figure 7.4).

7.5.2 Prefix Search Decoding

The second method (prefix search decoding) relies on the fact that, by modifying the forward variables of Section 7.3, we can efficiently calculate the probabilities of successive extensions of labelling prefixes.

Prefix search decoding is a best-first search (see e.g. Russell and Norvig, 2003, chap. 4) through the tree of labellings, where the children of a given labelling are those that share it as a prefix. At each step the search extends the labelling whose children have the largest cumulative probability (see Figure 7.5).

Let $\gamma_t(\mathbf{p}_n)$ be the probability of the network outputting prefix $\mathbf{p}$ by time t such that a non-blank label is output at t. Similarly, let $\gamma_t(\mathbf{p}_b)$ be the

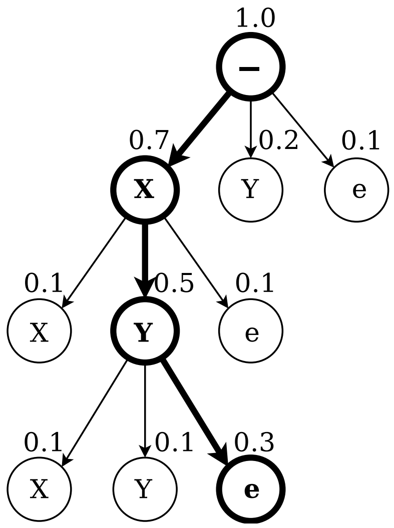


Figure 7.5: **Prefix search decoding on the alphabet {X,Y}.** Each node either ends ('e') or extends the prefix at its parent node. The number above an extending node is the total probability of all labellings beginning with that prefix. The number above an end node is the probability of the single labelling ending at its parent. At every iteration the extensions of the most probable remaining prefix are explored. Search ends when a single labelling (here 'XY') is more probable than any remaining prefix.

probability of the network outputting prefix $\mathbf{p}$ by time t such that the blank label is output at t. i.e.

$$\gamma_t(\mathbf{p}_n) = P(\pi_{1:t} : \mathcal{B}(\pi_{1:t}) = \mathbf{p}, \pi_t = \mathbf{p}_{|\mathbf{p}|}|\mathbf{x}) \tag{7.32}$$

$$\gamma_t(\mathbf{p}_b) = P(\pi_{1:t} : \mathcal{B}(\pi_{1:t}) = \mathbf{p}, \pi_t = blank|\mathbf{x}) \tag{7.33}$$

Then for a length T input sequence $\mathbf{x}$, $p(\mathbf{p}|\mathbf{x}) = \gamma_T(\mathbf{p}_n) + \gamma_T(\mathbf{p}_b)$. Also let $p(\mathbf{p}\ldots|\mathbf{x})$ be the cumulative probability of all labellings not equal to $\mathbf{p}$ of which $\mathbf{p}$ is a prefix

$$p(\mathbf{p}\ldots|\mathbf{x}) = \sum_{\mathbf{l}\neq\emptyset} P(\mathbf{p}+\mathbf{l}|\mathbf{x}), \tag{7.34}$$

where $\emptyset$ is the empty sequence. With these definitions is mind, the pseudocode for prefix search decoding is given in Algorithm 7.1.

Given enough time, prefix search decoding always finds the most probable labelling. However, the maximum number of prefixes it must expand grows exponentially with the input sequence length. If the output distribution is sufficiently peaked around the mode, it will nonetheless finish in reasonable time. For many tasks however, a further heuristic is required to make its application feasible.

```
Initialisation:
$1 \le t \le T \begin{cases} \gamma_t(\emptyset_n) = 0 \\ \gamma_t(\emptyset_b) = \prod_{t'=1}^{t} y_b^{t'} \end{cases}$
$p(\emptyset|\mathbf{x}) = \gamma_T(\emptyset_b)$
$p(\emptyset \ldots |\mathbf{x}) = 1 - p(\emptyset|\mathbf{x})$
$\mathbf{l}^* = \mathbf{p}^* = \emptyset$
$P = \{\emptyset\}$

Algorithm:
while $p(\mathbf{p}^* \ldots |\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ do
   $probRemaining = p(\mathbf{p}^* \ldots |\mathbf{x})$
   for all labels $k \in L$ do
      $\mathbf{p} = \mathbf{p}^* + k$
      $\gamma_1(\mathbf{p}_n) = \begin{cases} y_k^1 \text{ if } \mathbf{p}^* = \emptyset \\ 0 \text{ otherwise} \end{cases}$
      $\gamma_1(\mathbf{p}_b) = 0$
      $prefixProb = \gamma_1(\mathbf{p}_n)$
      for $t = 2$ to $T$ do
         $newLabelProb = \gamma_{t-1}(\mathbf{p}_b^*) + \begin{cases} 0 \text{ if } \mathbf{p}^* \text{ ends in } k \\ \gamma_{t-1}(\mathbf{p}_n^*) \text{ otherwise} \end{cases}$
         $\gamma_t(\mathbf{p}_n) = y_k^t \, (newLabelProb + \gamma_{t-1}(\mathbf{p}_n))$
         $\gamma_t(\mathbf{p}_b) = y_b^t \, (\gamma_{t-1}(\mathbf{p}_b) + \gamma_{t-1}(\mathbf{p}_n))$
         $prefixProb \mathrel{+}= y_k^t \; newLabelProb$
      $p(\mathbf{p}|\mathbf{x}) = \gamma_T(\mathbf{p}_n) + \gamma_T(\mathbf{p}_b)$
      $p(\mathbf{p} \ldots |\mathbf{x}) = prefixProb - p(\mathbf{p}|\mathbf{x})$
      $probRemaining \mathrel{-}= p(\mathbf{p} \ldots |\mathbf{x})$
      if $p(\mathbf{p}|\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ then
         $\mathbf{l}^* = \mathbf{p}$
      if $p(\mathbf{p} \ldots |\mathbf{x}) > p(\mathbf{l}^*|\mathbf{x})$ then
         add $\mathbf{p}$ to $P$
      if $probRemaining \le p(\mathbf{l}^*|\mathbf{x})$ then
         break
   remove $\mathbf{p}^*$ from $P$
   set $\mathbf{p}^* = \mathbf{p} \in P$ that maximises $p(\mathbf{p} \ldots |\mathbf{x})$

Termination:
output $\mathbf{l}^*$
```

Algorithm 7.1: Prefix Search Decoding Algorithm.

Observing that the outputs of a trained CTC network tend to form a series of spikes separated by strongly predicted blanks (see Figure 7.1), we can divide the output sequence into sections that are very likely to begin and end with a blank. We do this by choosing boundary points where the probability of observing a blank label is above a certain threshold. We then apply Algorithm 7.1 to each section individually and concatenate these to get the final transcription.

In practice, prefix search works well with this heuristic, and generally outperforms best path decoding. However it still makes mistakes in some cases, e.g. if the same label is predicted weakly on both sides of a section boundary.

7.5.3 Constrained Decoding

For certain tasks we want to constrain the output labellings according to some predefined grammar. For example, in speech and handwriting recognition, the final transcriptions are usually required to form sequences of dictionary words. In addition it is common practice to use a language model to weight the probabilities of particular sequences of words.

We can express these constraints by altering the label sequence probabilities in (7.30) to be conditioned on some probabilistic grammar G, as well as the input sequence $\mathbf{x}$

$$\mathbf{l}^* = \arg\max_{\mathbf{l}} p(\mathbf{l}|\mathbf{x}, G). \tag{7.35}$$

Absolute requirements, for example that $\mathbf{l}$ contains only dictionary words, can be incorporated by setting the probability of all sequences that fail to meet them to 0.

At first sight, conditioning on G would seem to contradict a basic assumption of CTC: that the labels are conditionally independent given the input sequences (see Section 7.2). Since the network attempts to model the probability of the whole labelling at once, there is nothing to stop it from learning inter-label transitions direct from the data, which would then be skewed by the external grammar. Indeed, when we tried using a biphone model to decode a CTC network trained for phoneme recognition, the error rate increased. However, CTC networks are typically only able to learn local relationships such as commonly occurring pairs or triples of labels. Therefore as long as G focuses on long range label dependencies (such as the probability of one word following another when the outputs are letters) it doesn't interfere with the dependencies modelled internally by CTC. This argument is supported by the results in Sections 7.6.4 and 7.6.5.

Applying the basic rules of probability we obtain

$$p(\mathbf{l}|\mathbf{x}, G) = \frac{p(\mathbf{l}|\mathbf{x})p(\mathbf{l}|G)p(\mathbf{x})}{p(\mathbf{x}|G)p(\mathbf{l})}, \tag{7.36}$$

where we have used the fact that $\mathbf{x}$ is conditionally independent of G given $\mathbf{l}$. If we assume that $\mathbf{x}$ is independent of G, (7.36) reduces to

$$p(\mathbf{l}|\mathbf{x}, G) = \frac{p(\mathbf{l}|\mathbf{x})p(\mathbf{l}|G)}{p(\mathbf{l})}. \tag{7.37}$$

This assumption is in general false, since both the input sequences and the grammar depend on the underlying generator of the data, for example the language being spoken. However it is a reasonable first approximation, and is particularly justifiable in cases where the grammar is created using data other than that from which $\mathbf{x}$ was drawn (as is common practice in speech and handwriting recognition, where separate textual corpora are used to generate language models).

If we further assume that, prior to any knowledge about the input or the grammar, all label sequences are equally probable, (7.35) reduces to

$$\mathbf{l}^* = \arg\max_{\mathbf{l}} p(\mathbf{l}|\mathbf{x})p(\mathbf{l}|G). \tag{7.38}$$

Note that, since the number of possible label sequences is finite (because both L and $|\mathbf{l}|$ are finite), assigning equal prior probabilities does not lead to an improper prior.

CTC Token Passing Algorithm

We now describe an algorithm, based on the *token passing algorithm* for HMMs (Young et al., 1989), that allows us to find an approximate solution to (7.38) for a simple grammar.

Let G consist of a dictionary D containing W words, and a set of W^2 bigrams $p(w|\hat{w})$ that define the probability of making a transition from word $\hat{w}$ to word w. The probability of any label sequence that does not form a sequence of dictionary words is 0.

For each word w, define the modified word w' as w with blanks added at the beginning and end and between each pair of labels. Therefore $|w'| = 2|w| + 1$. Define a token $tok = (score, history)$ to be a pair consisting of a real valued *score* and a *history* of previously visited words. In fact, each token corresponds to a particular path through the network outputs, and the token score is the log probability of that path. The basic idea of the token passing algorithm is to pass along the highest scoring tokens at every word state, then maximise over these to find the highest scoring tokens at the next state. The transition probabilities are used when a token is passed from the last state in one word to the first state in another. The output word sequence is then given by the history of the highest scoring end-of-word token at the final timestep.

At every timestep t of the length T output sequence, each segment s of each modified word w' holds a single token $tok(w, s, t)$. This is the highest scoring token reaching that segment at that time. In addition we define the *input token* $tok(w, 0, t)$ to be the highest scoring token arriving at word w at time t, and the *output token* $tok(w, -1, t)$ to be the highest scoring token leaving word w at time t.

Pseudocode is provided in Algorithm 7.2.

```
 1: **Initialisation:**
 2: for all words $w \in D$ do
 3:   $tok(w, 1, 1) = (\ln y^1_b, (w))$
 4:   $tok(w, 2, 1) = (\ln y^1_{w_1}, (w))$
 5:   if $|w| = 1$ then
 6:     $tok(w, -1, 1) = tok(w, 2, 1)$
 7:   else
 8:     $tok(w, -1, 1) = (-\infty, ())$
 9:   $tok(w, s, 1) = (-\infty, ())$ for all other $s$
10:
11: **Algorithm:**
12: for $t = 2$ to $T$ do
13:   sort output tokens $tok(w, -1, t-1)$ by ascending score
14:   for all words $w \in D$ do
15:     $w^* = \arg\max_{\hat{w} \in D} [tok(\hat{w}, -1, t-1).score + \ln p(w|\hat{w})]$
16:     $tok(w, 0, t).score = tok(w^*, -1, t-1).score + \ln p(w|w^*)$
17:     $tok(w, 0, t).history = tok(w^*, -1, t-1).history + w$
18:     for segment $s = 1$ to $|w'|$ do
19:       $P = \{tok(w, s, t-1), tok(w, s-1, t-1)\}$
20:       if $w'_s \neq blank$ and $s > 2$ and $w'_{s-2} \neq w'_s$ then
21:         add $tok(w, s-2, t-1)$ to $P$
22:       $tok(w, s, t) =$ token in $P$ with highest score
23:       $tok(w, s, t).score \mathrel{+}= \ln y^t_{w'_s}$
24:     $tok(w, -1, t) =$ highest scoring of $\{tok(w, |w'|, t), tok(w, |w'|-1, t)\}$
25:
26: **Termination:**
27: $w^* = \arg\max_w tok(w, -1, T).score$
28: output $tok(w^*, -1, T).history$
```

Algorithm 7.2: CTC Token Passing Algorithm

Computational Complexity

The CTC token passing algorithm has a worst case complexity of $O(TW^2)$, since line 15 requires a potential search through all W words. However, because the output tokens $tok(w, -1, T)$ are sorted in order of score, the search can be terminated when a token is reached whose score is less than the current best score with the transition included. The typical complexity is therefore considerably lower, with a lower bound of $O(TWlogW)$ to account for the sort. If no bigrams are used, lines 15-17 can be replaced by a simple search for the highest scoring output token, and the complexity reduces to $O(TW)$.

7.6 Experiments

In this section we apply RNNs with CTC output layers to five temporal classification tasks. The first three tasks are on speech recognition, and the remaining two are on handwriting recognition. In all cases, we use bidirectional LSTM for the network architecture.

For the handwriting tasks, a dictionary and language model were present, and we recorded results both with and without the constrained decoding algorithm of Section 7.5.3. For the speech experiments there was no dictionary or language model, and the output labels (whether phonemes or whole words) were used directly for transcription. For the experiment in Section 7.6.1, we compare prefix search and best path decoding (see Section 7.5).

As discussed in Chapter 3, the choice of input representation is crucial to any machine learning algorithm. For most of the experiments here, we used standard input representations that have been tried and tested with other sequence learning algorithms, such as HMMs. The experiment in Section 7.6.4 was different in that we explicitly compared the performance of CTC using two different input representations. As usual, all inputs were standardised to have mean 0 and standard deviation 1 over the training set.

For all the experiments, the BLSTM hidden layers were fully connected to themselves, and to the input and output layers. Each memory block contained a single LSTM cell, with hyperbolic tangent used for the activation functions g and h and a logistic sigmoid used for the activation function of the gates. The sizes of the input and output layers were determined by the numbers of inputs and labels in each task. The weights were randomly initialised from a Gaussian distribution with mean 0 and standard deviation 0.1. Online steepest descent with momentum was used for training, with a learning rate 10^{-4} and a momentum of 0.9. All experiments used separate training, validation and testing sets. Training was stopped when 50 epochs had passed with no reduction of error on the validation set.

The only network parameters manually adjusted for the different tasks were (1) the number of blocks in the LSTM layers and (2) the variance of the input noise added during training. We specify these for each experiment.

For all experiments, the basic error measure used to evaluate performance was the *label error rate* defined in Section 2.3.4, applied to the test set. Note however that we rename the label error rate according to the type of labels used: for example *phoneme error rate* was used for phoneme

Table 7.1: **Phoneme error rate (PER) on TIMIT with 61 phonemes.** CTC and hybrid results are means over 5 runs, ± standard error. All differences were significant ($p < 0.01$), except that between HMM-BLSTM with weighted errors and CTC with best path decoding.

System	PER
HMM	35.21%
HMM-BLSTM (weighted error)	31.57 ± 0.06%
CTC (best path decoding)	31.47 ± 0.21%
CTC (prefix search decoding)	30.51 ± 0.19%

recognition, and *word error rate* for keyword spotting. For the handwriting recognition tasks, we evaluate both the *character error rate* for the labellings provided by the CTC output layer, and the *word error rate* for the word sequences obtained from the token passing algorithm. All the CTC experiments were repeated several times and the results are quoted as a mean ± the standard error. We also recorded the mean and standard error in the number of training epochs before the best results were achieved.

7.6.1 Phoneme Recognition

In this section we compare a CTC network with the best HMM and HMM-BLSTM hybrid results given in Chapter 6 for phoneme recognition on the TIMIT speech corpus (Garofolo et al., 1993). The task, data and preprocessing were identical to those described in Section 5.1.

Experimental Setup

The CTC network had 26 input units and 100 LSTM blocks in both the forward and backward hidden layers. It had 62 output units, one for each phoneme plus the 'blank', giving $114,662$ weights in total. Gaussian noise with mean 0 and standard deviation 0.6 was added to the inputs during training. When prefix search decoding was used, the probability threshold for the boundary points was 99.99%.

Results

Table 7.1 shows that, with prefix search decoding, CTC outperformed both an HMM and an HMM-RNN hybrid with the same RNN architecture. It also shows that prefix search gave a small improvement over best path decoding.

Note that the best hybrid results were achieved with a weighted error signal. Such heuristics are unnecessary for CTC, as its objective function

aa	aa, ao
ah	ah, ax, ax-h
er	er, axr
hh	hh, hv
ih	ih, ix
l	l, el
m	m, em
n	n, en, nx
ng	ng, eng
sh	sh, zh
sil	pcl, tcl, kcl, bcl, dcl, gcl, h#, pau, epi
uw	uw, ux
—	q

Table 7.2: **Folding the 61 phonemes in TIMIT onto 39 categories** (Lee and Hon, 1989). The phonemes in the right column are folded onto the corresponding category in the left column ('q' is discarded). All other TIMIT phonemes are left intact.

depends only on the *sequence* of labels, and not on their duration or segmentation.

Input noise had a greater impact on generalisation for CTC than the hybrid system, and a slightly higher level of noise was found to be optimal for CTC ($\sigma = 0.6$ instead of 0.5). The mean training time for the CTC network was 60.0 ± 7 epochs.

7.6.2 Phoneme Recognition with Reduced Label Set

In this Section we consider a variant of the previous task, where the number of distinct phoneme labels is reduced from 61 to 39. In addition, only the so-called *core test set* of TIMIT is used for evaluation. The purpose of these changes was to allow a direct comparison with other results in the literature.

Data and Preprocessing

In most previous studies, a set of 48 phonemes were selected for modelling during training, and confusions between several of these were ignored during testing, leaving an effective set of 39 distinct labels (Lee and Hon, 1989). Since CTC is a discriminative algorithm, using extra phonemes during training is unnecessary (and probably counterproductive), and the networks were therefore trained directly with 39 labels. The folding of the original 61 phoneme labels onto 39 categories is shown in table 7.2.

The TIMIT corpus was divided into a training set, a validation set and a test set according to (Halberstadt, 1998). As in our previous experiments, the training set contained 3696 sentences from 462 speakers. However in this case the test set was much smaller, containing only 192 sentences from 24 speakers, and the validation set, which contained 400 sentences from 50 speakers, was drawn from the unused test sentences rather than the training set. This left us with slightly more sentences for training than before. However this advantage was offset by the fact that the core test set is somewhat harder than the full test set.

As before, the speech data was transformed into Mel frequency cepstral coefficients (MFCC) using the HTK software package (Young et al., 2006). Spectral analysis was carried out with a 40 channel Mel filter bank from 64 Hz to 8 kHz. A pre-emphasis coefficient of 0.97 was used to correct spectral tilt. Twelve MFCC plus the 0th order coefficient were computed on Hamming windows 25 ms long, every 10 ms. In this case the second as well as first derivatives of the coefficients were used, giving a vector of 39 inputs in total.

Note that the best previous results on this task were achieved with more complex preprocessing techniques than MFCCs. For example, in (Yu et al., 2006) frequency-warped LPC cepstra were used as inputs, while Halberstadt and Glass tried a number of variations and combinations of MFCC, perceptual linear prediction (PLP) cepstral coefficients, energy and duration (Halberstadt, 1998; Glass, 2003).

Experimental Setup

The CTC network had an input layer of size 39, the forward and backward hidden layers had 128 blocks each, and the output layer was size 40 (39 phonemes plus blank). The total number of weights was 183,080. Gaussian noise with a standard deviation of 0.6 was added to the inputs during training to improve generalisation. When prefix search was used, the probability threshold was 99.99%.

Results

Results are shown in table 7.3, along with the best results found in the literature. The performance of CTC with prefix search decoding was not significantly different from either of the best two results so far recorded (by Yu *et al* and Glass). However, unlike CTC, both of these systems were heavily tuned to speech recognition, and to TIMIT in particular, and would not be suitable for other sequence labelling tasks. Furthermore, some of the adaptations used by Yu *et al* and Glass, such as improved preprocessing and decoding, could also be applied to CTC networks, giving potentially better results.

Yu *et al.*'s hidden trajectory models (HTM) are a type of probabilistic generative model aimed at modelling speech dynamics and adding long range

Table 7.3: **Phoneme error rate (PER) on TIMIT with 39 phonemes.** Results for CTC are the average ± standard error over 10 runs. On average, the networks were trained for 112.5 epochs (± 6.4). CTC with prefix search is not significantly different from either Yu *et al* or Glass's results, but is significantly better than all other results (including CTC with best path).

System	PER
Large margin HMM (Sha and Saul, 2006)	28.7%
Baseline HMM (Yu et al., 2006)	28.57%
Triphone continuous density HMM (Lamel and Gauvain, 1993)	27.1%
RNN-HMM hybrid (Robinson, 1994)	26.1%
Bayesian triphone HMM (Ming and Smith, 1998)	25.6%
Near-miss, probabilistic segmentation (chang, 1998)	25.5%
CTC (best path decoding)	25.17 ± 0.2%
HTM (Yu et al., 2006)	24.9%
CTC (prefix search decoding)	24.58 ± 0.2%
Committee-based classifier (Glass, 2003)	24.4%

context that is missing in standard HMMs.

Glass's system is a segment-based speech recogniser (as opposed to a frame-based recogniser) which attempts to detect *landmarks* in the speech signal. Acoustic features are computed over hypothesised segments and at their boundaries. The standard decoding framework is modified and extended to deal with this.

Yu *et al.*'s best result was achieved with a lattice-constrained A* search with weighted HTM, HMM, and language model scores. Glass's best results were achieved with many heterogeneous information sources and classifier combinations. It is likely that both Yu *et al.*'s HTM and CTC would achieve improved performance when combined with other classifiers and/or more sources of input information.

7.6.3 Keyword Spotting

The task in this section is keyword spotting, using the Verbmobil speech corpus (Verbmobil, 2004). The aim of keyword spotting is to identify a particular set of spoken words within (typically unconstrained) speech signals. In most cases, the keywords occupy only a small fraction of the total data. Discriminative approaches are interesting for keyword spotting, because they are able to concentrate on identifying and distinguishing the keywords, while ignoring the rest of the signal. However, the predominant method is to use hidden Markov models, which are generative, and must therefore model the unwanted speech, and even the non-speech noises, in addition to the keywords.

In many cases one seeks not only the identity of the keywords, but also their approximate position. For example, this would be desirable if the goal were to further examine those segments of a long telephone conversation in which a keyword occurred. In principle, locating the keywords presents a problem for CTC, since the network is only trained to find the sequence of labels, and not their position. However we have observed that in most cases CTC predicts labels close to the relevant segments of the input sequence. In the following experiments (Fernández et al., 2007a), we confirm this observation by recording the word error rate both with and without the requirement that the network find the approximate location of the keywords.

Data and Preprocessing

Verbmobil consists of dialogues of noisy, spontaneous German speech, where the purpose of each dialogue is to schedule a date for an appointment or meeting. It comes divided into training, validation and testing sets, all of which have a complete phonetic transcription. The training set includes 748 speakers and 23,975 dialogue turns, giving a total of 45.6 hours of speech. The validation set includes 48 speakers, 1,222 dialogue turns and a total of 2.9 hours of speech. The test set includes 46 speakers, 1,223 dialogue turns and a total of 2.5 hours of speech. Each speaker appears in only one of the sets.

The twelve keywords we chose were: *April, August, Donnerstag, Februar, Frankfurt, Freitag, Hannover, Januar, Juli, Juni, Mittwoch, Montag.* Since the dialogues are concerned with dates and places, all of these occurred fairly frequently in the data sets. One complication is that there are pronunciation variants of some of these keywords (e.g. "Montag" can end either with a /g/ or with a /k/). Another is that several keywords appear as sub-words, e.g. in plural form such as "Montags" or as part of another word such as "Ostermontag" (Easter Monday). The start and end times of the keywords were given by the automatic segmentation provided with the phonetic transcription.

In total there were 10,469 keywords on the training set with an average of 1.7% keywords per non-empty utterance (73.6% of the utterances did not have any keyword); 663 keywords on the validation set with an average of 1.7% keywords per non-empty utterance (68.7% of the utterances did not have any keyword); and 620 keywords on the test set with an average of 1.8 keywords per non-empty utterance (71.1% of the utterances did not have any keyword).

The audio preprocessing was identical to that described in Section 5.1, except that the second order derivatives of the MFCC coefficients were also computed, giving a total of 39 inputs per frame.

Table 7.4: **Keyword error rate (KER) on Verbmobil.** Results are a mean over 4 runs, ± standard error.

System	KER
CTC (approx. location)	15.5 ± 1.2%
CTC (any location)	13.9 ± 0.7%

Experimental Setup

The CTC network contained 128 LSTM blocks in the forward and backward hidden layers. The output layer contained 13 units and the input layer contained 39 units, giving 176,141 weights in total. Gaussian noise with a mean of 0 and a standard deviation of 0.5 was added during training.

We defined the CTC network as having found the approximate location of a keyword if the corresponding label was output within 0.5 seconds of the boundary of the keyword segment. The experiment was not repeated for the approximate location results: the output of the network was simply re-scored with location required.

Results

Table 7.4 shows that the CTC network gave a mean error rate of 15.5%. The CTC system performed slightly better without the constraint that it find the approximate location of the keywords. This shows in most cases it aligned the keywords with the relevant portion of the input signal.

Although we don't have a direct comparison for this result, a benchmark HMM system performing full speech recognition on the same dataset achieved a word error rate of 35%. We attempted to train an HMM system specifically for keyword spotting, with a single junk model for everything apart from the keywords, but found that it did not converge. This is symptomatic of the difficulty of using a generative model for a task where so much of the signal is irrelevant.

The mean training time for the CTC network was 91.3 ± 22.5 epochs.

Analysis

Figure 7.6 shows the CTC outputs during a dialogue turn containing several keywords. For a zoomed in section of the same dialogue turn, Figure 7.7 shows the sequential Jacobian for the output unit associated with the keyword "Donnerstag" at the time step indicated by an arrow at the top of the figure. The extent (0.9 s) and location of the keyword in the speech signal is shown at the top of the figure. As can be seen, the output is most sensitive to the first part of the keyword. This is unsurprising, since the

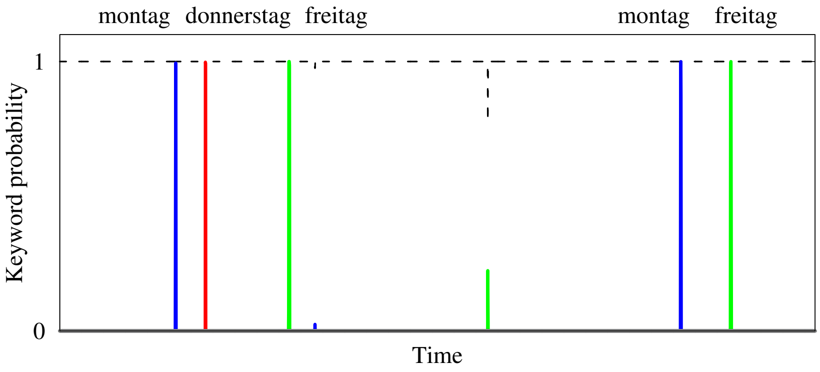


Figure 7.6: **CTC outputs for keyword spotting on Verbmobil**

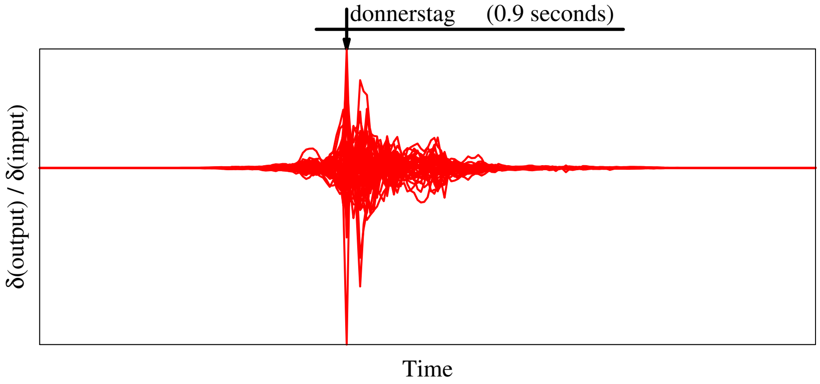


Figure 7.7: **Sequential Jacobian for keyword spotting on Verbmobil**

ending, “tag”, is shared by many of the keywords and is therefore the least discriminative part.

7.6.4 Online Handwriting Recognition

The task in this section is online handwriting recognition, using the IAM-OnDB handwriting database (Liwicki and Bunke, 2005b)[1]. In online handwriting recognition, the state and position of the pen is recorded during writing, and used as input to the learning algorithm.

For the CTC experiments (Liwicki et al., 2007; Graves et al., 2008a,b), we record the the character error rate using standard best-path decoding, and the word error rate using constrained decoding. For the HMM system, only the word error rate is given.

We compare results using two different input representations, one hand

[1] Available for public download at http://www.iam.unibe.ch/ fki/iamondb/

crafted for HMMs, the other consisting of raw data direct from the pen sensor.

Data and Preprocessing

IAM-OnDB consists of pen trajectories collected from 221 different writers using a ‘smart whiteboard’ (Liwicki and Bunke, 2005a). The writers were asked to write forms from the LOB text corpus (Johansson et al., 1986), and the position of their pen was tracked using an infra-red device in the corner of the board. The original input data consists of the x and y pen coordinates, the points in the sequence when individual strokes (i.e. periods when the pen is pressed against the board) end, and the times when successive position measurements were made. Recording errors in the x, y data was corrected by interpolating to fill in for missing readings, and removing steps whose length exceeded a certain threshold.

The character level transcriptions contain 80 distinct target labels (capital letters, lower case letters, numbers, and punctuation). A dictionary consisting of the $20,000$ most frequently occurring words in the LOB corpus was used for decoding, along with a bigram language model. 5.6% of the words in the test set were not contained in the dictionary. The language model was optimised on the training and validation sets only.

IAM-OnDB is divided into a training set, two validation sets, and a test set, containing respectively 5364, 1,438, 1,518 and 3,859 written lines taken from 775, 192, 216 and 544 forms. For our experiments, each line was assumed to be an independent sequence (meaning that the dependencies between successive lines, e.g. for a continued sentence, were ignored).

Two input representations were used for this task. The first consisted simply of the offset of the x, y coordinates from the top left of the line, along with the time from the beginning of the line, and an extra input to mark the points when the pen was lifted off the whiteboard (see Figure 7.8). We refer to this as the *raw* representation.

The second representation required a large amount of sophisticated preprocessing and feature extraction (Liwicki et al., 2007). We refer to this as the *preprocessed* representation. Briefly, in order to account for the variance in writing styles, the pen trajectories were first normalised with respect to such properties as the slant, skew and width of the letters, and the slope of the line as a whole. Two sets of input features were then extracted, one consisting of ‘online’ features, such as pen position, pen speed, line curvature etc., and the other consisting of ‘offline’ features derived from a two dimensional window of the image reconstructed from the pen trajectory. Delayed strokes (such as the crossing of a ‘t’ or the dot of an ‘i’) are removed by the preprocessing because they introduce difficult long time dependencies

Table 7.5: **CTC Character error rate (CER) on IAM-OnDB.** Results are a mean over 4 runs, ± standard error.

Input	CER
Raw	13.9 ± 0.1%
Preprocessed	11.5 ± 0.05%

Table 7.6: **Word error rate (WER) on IAM-OnDB.** LM = language model. CTC results are a mean over 4 runs, ± standard error. All differences were significant ($p < 0.01$)

System	Input	LM	WER
HMM	Preprocessed	✓	35.5%
CTC	Raw	✗	30.1 ± 0.5%
CTC	Preprocessed	✗	26.0 ± 0.3%
CTC	Raw	✓	22.8 ± 0.2%
CTC	Preprocessed	✓	20.4 ± 0.3%

Experimental Setup

The CTC network contained 100 LSTM blocks in the forward and backward hidden layers. The output layer contained 81 units. For the raw representation, there were 4 input units, giving 100,881 weights in total. For the preprocessed representation, there were 25 input units, giving 117,681 weights in total. No noise was added during training.

The HMM setup (Liwicki et al., 2007) contained a separate, linear HMM with 8 states for each character ($8 * 81 = 648$ states in total). Diagonal mixtures of 32 Gaussians were used to estimate the observation probabilities. All parameters, including the word insertion penalty and the grammar scale factor, were optimised on the validation set.

Results

From Table 7.6 we can see that, with a language model and the preprocessed input representation, CTC gives a mean word error rate of 20.4%, compared to 35.5% with a benchmark HMM. This is an error reduction of 42.5%. Moreover, even without the language model or the handcrafted preprocessing, CTC well outperforms HMMs.

The mean training time for the CTC network was 41.3 ± 2.4 epochs for the preprocessed data, and 233.8 ± 16.8 epochs for the raw data. This disparity reflects the fact that the preprocessed data contains simpler correlations and shorter time-dependencies, and is therefore easier for the network to learn. It is interesting to note how large the variance in training epochs was for the raw data, given how small the variance in final performance was.

Analysis

The CTC results with the raw inputs, where the information required to identify each character is distributed over many timesteps, demonstrate the ability of BLSTM to make use of long range contextual information. An indication of the amount of context required is given by the fact that when we attempted to train a CTC network with a standard BRNN architecture on the same task, it did not converge.

Figures 7.9 and 7.10 show sequential Jacobians for BLSTM CTC networks using respectively the raw and preprocessed inputs for a phrase from IAM-OnDB. As expected, the size of the region of high sensitivity is considerably larger for the raw representation, because the preprocessing creates localised input features that do not require as much use of long range context.

7.6.5 Offline Handwriting Recognition

This section describes an *offline* handwriting recognition experiment, using the IAM-DB offline handwriting corpus (Marti and Bunke, 2002)[2]. Offline handwriting differs from online in that only the final image created by the pen is available to the algorithm. This makes the extraction of relevant input features more difficult, and usually leads to lower recognition rates.

Data and Preprocessing

The IAM-DB training set contains 6,161 text lines written by 283 writers, the validation set contains 920 text lines by 56 writers, and the test set contains 2,781 text lines by 161 writers. No writer in the test set appears in either the training or validation sets.

Substantial preprocessing was used for this task (Marti and Bunke, 2001). Briefly, to reduce the impact of different writing styles, a handwritten text line image is normalised with respect to skew, slant, and baseline position in the preprocessing phase. After these normalisation steps, a handwritten text line is converted into a sequence of feature vectors. For this purpose a sliding window is used which is moved from left to right, one pixel at each step. Nine geometrical features are extracted at each position of the sliding window.

A dictionary and statistical language model, derived from the same three textual corpora as used in Section 7.6.4, were used for decoding (Bertolami

[2] Available for public download at http://www.iam.unibe.ch/ fki/iamDB

Table 7.7: **Word error rate (WER) on IAM-DB.** LM = language model. CTC results are a mean over 4 runs, ± standard error.

System	LM	WER
HMM	✓	35.5%
CTC	✗	34.6 ± 1.1%
CTC	✓	25.9 ± 0.8%

and Bunke, 2007). The integration of the language model was optimised on a validation set. As before, the dictionary consisted of the 20,000 most frequently occurring words in the corpora.

Experimental Setup

The HMM-based recogniser was identical to the one used in (Bertolami and Bunke, 2007), with each character modelled by a linear HMM. The number of states was chosen individually for each character (Zimmermann et al., 2006b), and twelve Gaussian mixture components were used to model the output distribution in each state.

The CTC network contained 100 LSTM blocks in the forward and backward hidden layers. There were 9 inputs and 82 outputs, giving 105,082 weights in total. No noise was added during training.

Results

From Table 7.7 it can be seen that CTC with a language model gave a mean word error rate of 25.9%, compared to 35.5% with a benchmark HMM. This is an error reduction of 27.0%. Without the language model however, CTC did not significantly outperform the HMM. The character error rate for the CTC system was 18.2 ± 0.6%.

While these results are impressive, the advantage of CTC over HMMs was smaller for this task than for the online recognition in Section 7.6.4. A possible reason for this is that the two-dimensional nature of offline handwriting is less well suited to RNNs than the one-dimensional time series found in online handwriting.

The mean training time for the CTC network was 71.3 ± 7.5 epochs.

7.7 Discussion

For most of the experiments in this chapter, the performance gap between CTC networks and HMMs is substantial. In what follows, we discuss the key differences between the two systems, and suggest reasons for the RNN's superiority.

Firstly, HMMs are generative, while an RNN trained with CTC is discriminative. As discussed in Section 2.2.3, advantages of the generative approach include the possibility of adding extra models to an already trained system, and being able to generate synthetic data. However, discriminative methods tend to give better results for classification tasks, because they focus entirely on finding the correct labels. Additionally, for tasks where the prior data distribution is hard to determine, generative approaches can only provide unnormalised likelihoods for the label sequences. Discriminative approaches, on the other hand, yield normalised label probabilities, which can be used to assess prediction confidence, or to combine the outputs of several classifiers.

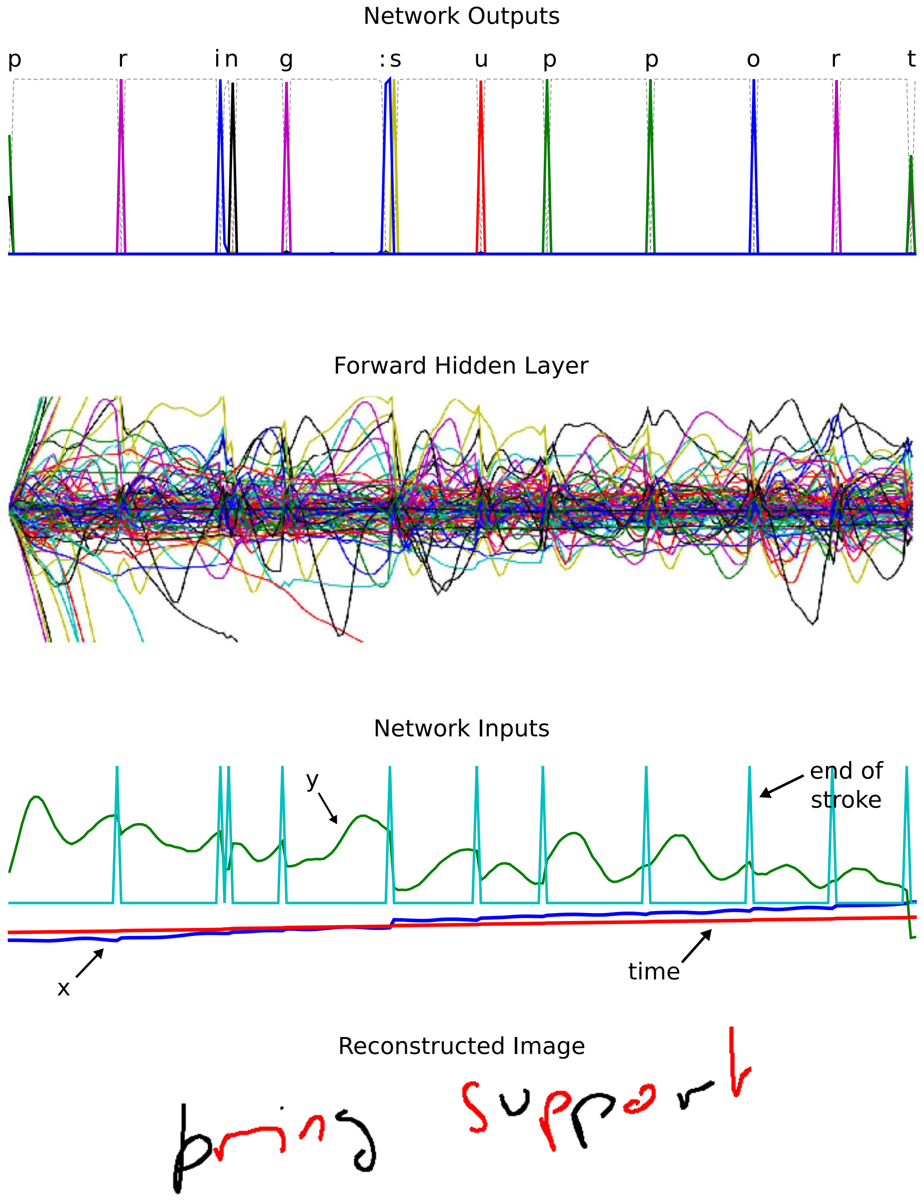
A second difference is that RNNs, and particularly LSTM, provide more flexible models of the input features than the mixtures of diagonal Gaussians used in standard HMMs. In general, mixtures of Gaussians can model complex, multi-modal distributions; however, when the Gaussians have diagonal covariance matrices (as is usually the case) they are limited to modelling distributions over independent variables. This assumes that the input features are decorrelated, which can be difficult to ensure for real-world tasks. RNNs, on the other hand, do not assume that the features come from a particular distribution, or that they are independent, and can model nonlinear relationships among features. However, RNNs generally perform better using input features with simpler inter-dependencies.

A third difference is that the internal states of a standard HMM are discrete and single valued, while those of an RNN are defined by the vector of activations of the hidden units, and are therefore continuous and multivariate. This means that for an HMM with N states, only $O(logN)$ bits of information about the past observation sequence are carried by the internal state. For an RNN, on the other hand, the amount of internal information grows linearly with the number of hidden units.

A fourth difference is that HMMs are constrained to segment the entire input, in order to determine the sequence of hidden states. This is often problematic for continuous input sequences, since the precise boundary between units, e.g. characters, can be ambiguous. It is also an unnecessary burden in tasks, such as keyword spotting, where most of the inputs should be ignored. A further problem with segmentation is that, at least with standard Markovian transition probabilities, the probability of remaining in a particular state decreases exponentially with time. Exponential decay is in general a poor model of state duration, and various measures have been suggested to alleviate this (Johnson, 2005). However, an RNN trained with CTC does not need to segment the input sequence, and therefore avoids both of these problems.

A final, and perhaps most crucial, difference is that unlike RNNs, HMMs assume that the probability of each observation depends only on the current state. A consequence of this is that data consisting of continuous trajectories (such as the sequence of pen coordinates for online handwriting, and the sequence of window positions in offline handwriting) are difficult to model with standard HMMs, since each observation is heavily dependent on those around it. Similarly, data with long range contextual dependencies is troublesome, because individual sequence elements (e.g., letters or phonemes) are influenced by the elements surrounding them. The latter problem can be mitigated by adding extra models to account for each sequence element in all different contexts (e.g., using triphones instead of phonemes for speech recognition). However, increasing the number of models exponentially increases the number of parameters that must be inferred which, in turn, increases the amount of data required to reliably train the system. For RNNs on the other hand, modelling continuous trajectories is natural, since their own hidden state is itself a continuous trajectory. Furthermore, the range of contextual information accessible to an RNN is limited only by the choice of architecture, and in the case of BLSTM can in principle extend to the entire input sequence.

To summarise, the observed advantages of CTC networks over HMMs can be explained by the fact that, as researchers approach problems of increasing complexity, the assumptions HMMs are based on lose validity. For example, unconstrained handwriting or spontaneous speech are more dynamic and show more pronounced contextual effects than hand printed scripts or read speech.


Network Outputs

Forward Hidden Layer

Network Inputs

Reconstructed Image

Figure 7.8: **CTC network labelling an excerpt from IAM-OnDB, using raw inputs.** The ':' label in the outputs is an end-of-word marker. The 'Reconstructed Image' was recreated from the pen positions stored by the sensor. Successive strokes have been alternately coloured red and black to highlight their boundaries. Note that strokes do not necessarily correspond to individual letters: this is no problem for CTC because it does not require segmented data. This example demonstrates the robustness of CTC to line slope, and illustrates the need for context when classifying letters (the 'ri' in 'bring' is ambiguous on its own, and the final 't' could equally be an 'l').

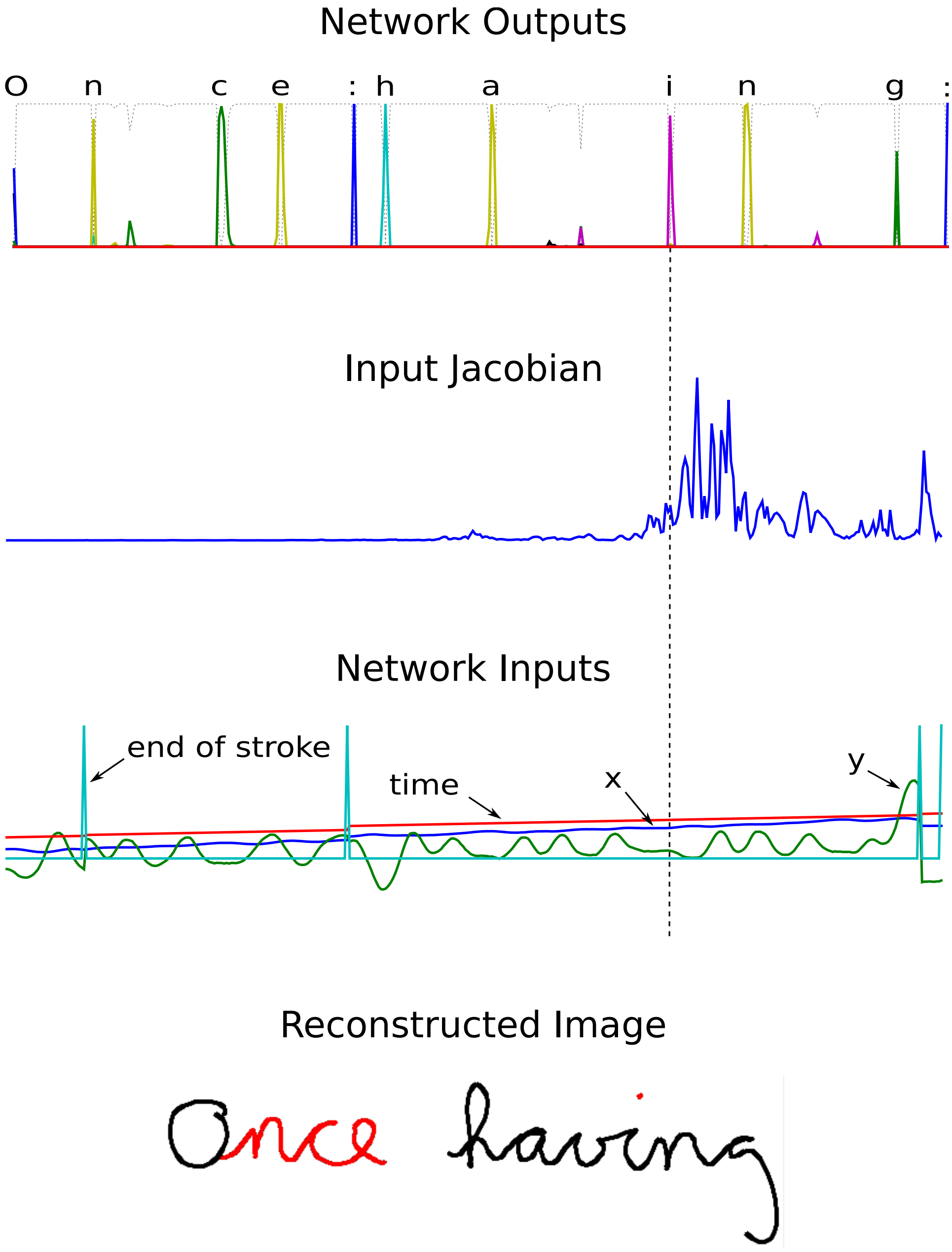


Figure 7.9: **CTC Sequential Jacobian from IAM-OnDB, with raw inputs.** For ease of visualisation, only the derivative with highest absolute value is plotted at each timestep. The Jacobian is plotted for the output corresponding to the label 'i' at the point when 'i' is emitted (indicated by the vertical dashed lines). Note that the network is mostly sensitive to the end of the word: this is possibly because 'ing' is a common suffix, and finding the 'n' and 'g' therefore increases the probability of identifying the 'i'. Note also the spike in sensitivity at the very end of the sequence: this corresponds to the delayed dot of the 'i'.

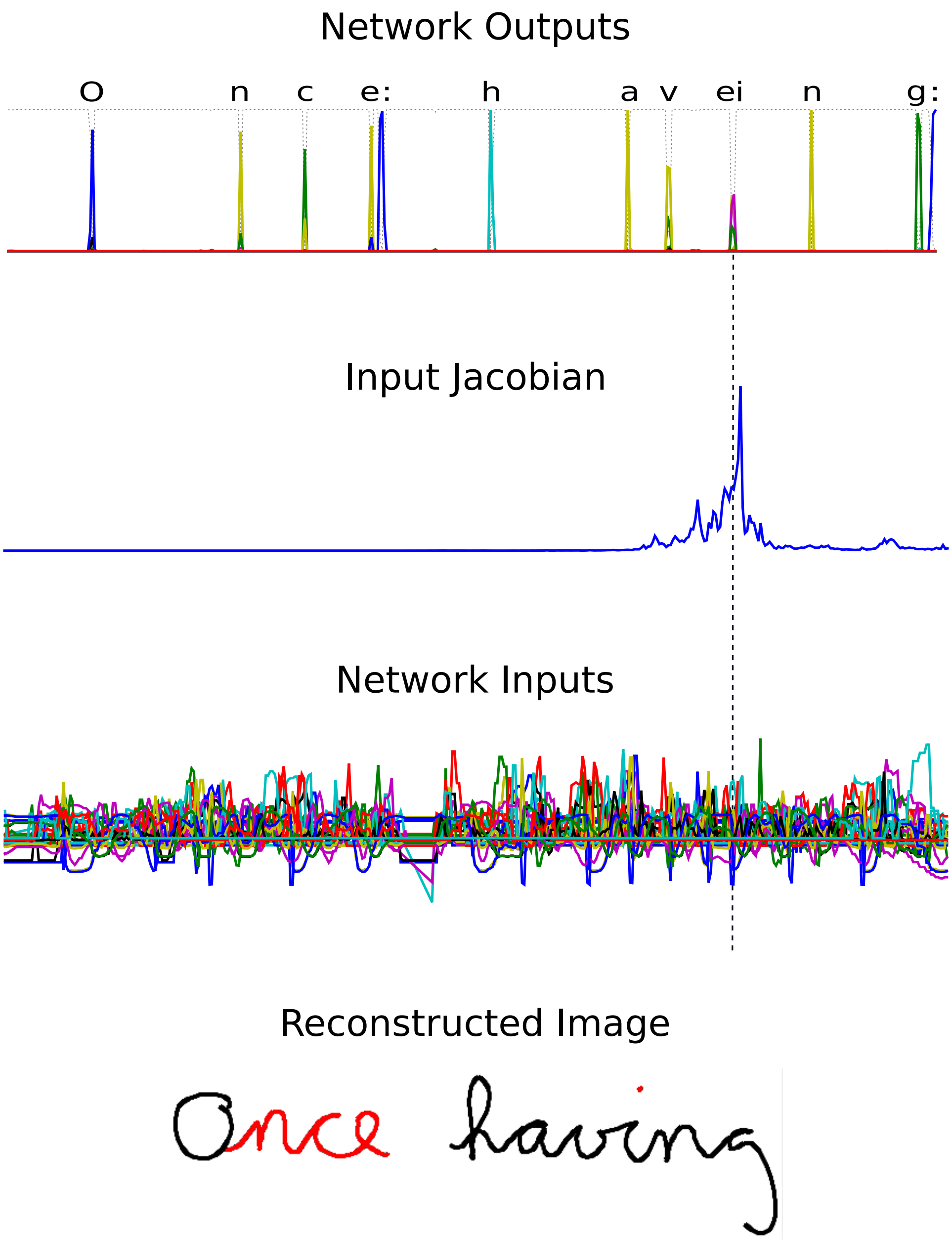


Figure 7.10: **CTC Sequential Jacobian from IAM-OnDB, with preprocessed inputs.** As before, only the highest absolute derivatives are shown, and the Jacobian is plotted at the point when ‘i’ is emitted. The range of sensitivity is smaller and more symmetrical than for the raw inputs.

Chapter 8

Multidimensional Recurrent Neural Networks

As we have seen in previous chapters, recurrent networks are an effective architecture for sequence learning tasks, where the data is strongly correlated along a single axis. This axis typically corresponds to time, or in some cases (such as protein secondary structure prediction) one-dimensional space. Some of the properties that make RNNs suitable for sequence learning, for example robustness to input warping and the ability to incorporate context, are also desirable in domains with more than one spatio-temporal dimension. However, standard RNNs are inherently one dimensional, and therefore poorly suited to multidimensional data. This chapter describes multidimensional recurrent neural networks (MDRNNs; Graves et al., 2007), a special case of directed acyclic graph RNNs (DAG-RNNs; Baldi and Pollastri, 2003). MDRNNs extend the potential applicability of RNNs to vision, video processing, medical imaging and many other areas, while avoiding the scaling problems that have plagued other multidimensional models. We also introduce multidimensional Long Short-Term Memory, thereby bringing the benefits of long range contextual processing to multidimensional tasks.

Although we will focus on the application of MDRNNs to supervised labelling and classification, it should be noted that the same architecture could be used for any task requiring the processing of multidimensional data.

Section 8.1 provides the background material and literature review for multidimensional algorithms. Section 8.2 describes the MDRNN architecture in detail. Section 8.3 presents experimental results on two image classification tasks.

8.1 Background

Recurrent neural networks were originally developed as a way of extending neural networks to sequential data. As discussed in previous chapters, the addition of recurrent connections allows RNNs to make use of previous context, as well as making them more more robust to warping along the time axis than non-recursive models. Access to contextual information and robustness to warping are also important when dealing with multidimensional data. For example, a face recognition algorithm should use the entire face as context, and should be robust to changes in perspective, distance etc. It therefore seems desirable to apply RNNs to such tasks.

However, the standard RNN architectures are inherently one dimensional, meaning that in order to use them for multidimensional tasks, the data must be preprocessed to one dimension, for example by presenting one vertical line of an image at a time to the network. Perhaps the best known use of neural networks for multidimensional data has been the application of convolution networks (LeCun et al., 1998a) to image processing tasks such as digit recognition (Simard et al., 2003). One disadvantage of convolution networks is that, because they are not recurrent, they rely on hand specified kernel sizes to introduce context. Another disadvantage is that they do not scale well to large images. For example, sequences of handwritten digits must be presegmented into individual characters before they can be recognised by convolution networks (LeCun et al., 1998a).

Other neural network based approaches to two-dimensional data (Pang and Werbos, 1996; Lindblad and Kinser, 2005) have been structured like cellular automata. A network update is performed at every timestep for every data-point, and contextual information propagates one step at a time in all directions. This provides an intuitive solution to the problem of simultaneously assimilating context from all directions. However, one problem is that the total computation time grows linearly with the range of context required, up to the length of the longest diagonal in the sequence.

A more efficient way of building multidimensional context into recurrent networks is provided by *directed acyclic graph RNNs* (DAG-RNNs; Baldi and Pollastri, 2003; Pollastri et al., 2006). DAG-RNNs generalise the forwards-backwards structure of bidirectional RNNs (see Section 3.2.3) to networks whose pattern of recurrency is determined by an arbitrary directed acyclic graph. The network processes the entire sequence in one pass, and the diffusion of context is as efficient as for one dimensional RNNs. An important special case of the DAG-RNN architecture occurs when the recurrency graph corresponds to an n-dimensional grid, with 2^n distinct networks used to process the data along all possible directions. Baldi refers to this as the "canonical" generalisation of BRNNs. In two dimensions, canonical DAG-RNNs have been successfully used to evaluate positions in the board game Go (Wu and Baldi, 2006). Furthermore the multidirectional version of the MDRNNs discussed in this chapter are equivalent to n-dimensional canonical DAG-RNNs, although the formulation is somewhat different.[1]

Various statistical models have also been proposed for multidimensional data, notably multidimensional hidden Markov models. However, multidimensional HMMs suffer from two serious drawbacks: (1) the time required to run the Viterbi algorithm, and thereby calculate the optimal state sequences, grows exponentially with the size of the data exemplars, and (2) the number of transition probabilities, and hence the required memory, grows exponentially with the data dimensionality. Numerous approximate methods have been proposed to alleviate one or both of these problems, including pseudo 2D and 3D HMMs (Hülsken et al., 2001), isolating elements (Li et al., 2000), approximate Viterbi algorithms (Joshi et al., 2005), and dependency tree HMMs (Jiten et al., 2006). However, none of these methods exploit the full multidimensional structure of the data.

As we will see, MDRNNs bring the benefits of RNNs to multidimensional data, without suffering from the scaling problems described above.

8.2 The MDRNN architecture

The basic idea of MDRNNs is to replace the single recurrent connection found in standard RNNs with as many recurrent connections as there are dimensions in the data. During the forward pass, at each point in the data sequence, the hidden layer of the network receives both an external input and its own activations from one step back along all dimensions. Figure 8.1 illustrates the two dimensional case.

Note that, although the word *sequence* usually denotes one dimensional data, we will use it to refer to independent data exemplars of any dimensionality. For example, an image is a two dimensional sequence, a video is a three dimensional sequence, and a series of fMRI brain scans is a four dimensional sequence.

Clearly, the data must be processed in such a way that when the network reaches a point in an n-dimensional sequence, it has already passed through all the points from which it will receive its previous activations. This can be ensured by following a suitable ordering on the set of points $\{(p_1, p_2, ..., p_n)\}$. One example of such an ordering is $(p_1, \ldots, p_n) < (p'_1, \ldots, p'_n)$ if $\exists\ m \in (1, \ldots, n)$ such that $p_m < p'_m$ and $p_d = p'_d\ \forall\ d \in (1, \ldots, m-1)$. Note that this is not the only possible ordering, and that its realisation for a particular sequence depends on an arbitrary choice of axes. We will return to this point in Section 8.2.1. Figure 8.3 illustrates the above ordering for a 2 dimensional sequence.

[1] We only discovered the equivalence between MDRNNs and DAG-RNNs during the review process of the thesis, after our first publication on the subject Graves et al., 2007 had already appeared.

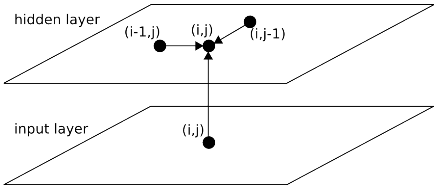


Figure 8.1: **2D RNN forward pass**

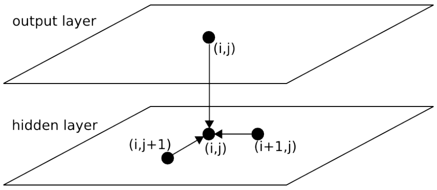


Figure 8.2: **2D RNN backward pass**

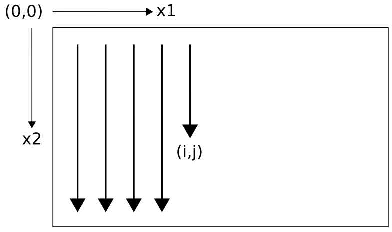


Figure 8.3: **Sequence ordering of 2D data.** The MDRNN forward pass starts at the origin and follows the direction of the arrows. The point (i,j) is never reached before both (i-1,j) and (i,j-1).

The forward pass of an MDRNN can then be carried out by feeding forward the input and the n previous hidden layer activations at each point in the ordered input sequence, and storing the resulting hidden layer activations. Care must be taken at the sequence boundaries not to feed forward activations from points outside the sequence.

Note that the 'points' in the input sequence will in general be multivalued vectors. For example, in a two dimensional colour image, the inputs could be single pixels represented by RGB triples, or blocks of pixels, or the outputs of a preprocessing method such as a discrete cosine transform.

The error gradient of an MDRNN (that is, the derivative of some objective function with respect to the network weights) can be calculated with an n-dimensional extension of backpropagation through time (BPTT; see Section 3.1.4 for more details). As with one dimensional BPTT, the sequence is processed in the reverse order of the forward pass. At each timestep, the hidden layer receives both the output error derivatives and its own n 'future' derivatives. Figure 8.2 illustrates the BPTT backward pass for two dimensions. Again, care must be taken at the sequence boundaries.

Define $a_j^{\mathbf{p}}$ and $b_j^{\mathbf{p}}$ respectively as the network input to unit j and the activation of unit j at point $\mathbf{p} = (p_1, \ldots, p_n)$ in an n-dimensional sequence $\mathbf{x}$. Let w_{ij}^d be the weight of the recurrent connection from unit i to unit j along dimension d. Consider an n-dimensional MDRNN with I input units,

K output units, and H hidden summation units. Let θ_h be the activation function of hidden unit h. Then the forward pass up to the hidden layer for a sequence with dimensions $(D_1, D_2, \ldots, D_n)$ is given in Algorithm 8.1.

```
for $p_1 = 0$ to $D_1 - 1$ do
  for $p_2 = 0$ to $D_2 - 1$ do
    ...
    for $p_n = 0$ to $D_n - 1$ do
      for $h = 1$ to $H$ do
        $a_h^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{ih}$
        for $d = 1$ to $n$ do
          if $p_d > 0$ then
            $a_h^{\mathbf{p}} \mathrel{+}= \sum_{h'=1}^{H} b_{h'}^{(p_1,\ldots,p_d-1,\ldots,p_n)} w_{h'h}^{d}$
        $b_h^{\mathbf{p}} = \theta_h(a_h^{\mathbf{p}})$
```

Algorithm 8.1: MDRNN Forward Pass

Note that units are indexed starting at 1, while coordinates are indexed starting at 0. For some unit j and some differentiable objective function O, define

$$\delta_j^{\mathbf{p}} \overset{\text{def}}{=} \frac{\partial O}{\partial a_j^{\mathbf{p}}} \tag{8.1}$$

Then the backward pass for the hidden layer is given in Algorithm 8.2.

```
for $p_1 = D_1 - 1$ to $0$ do
  for $p_2 = D_2 - 1$ to $0$ do
    ...
    for $p_n = D_n - 1$ to $0$ do
      for $h = 1$ to $H$ do
        $e_h^{\mathbf{p}} = \sum_{k=1}^{K} \delta_k^{\mathbf{p}} w_{hk}$
        for $d = 1$ to $n$ do
          if $p_d < D_d - 1$ then
            $e_h^{\mathbf{p}} \mathrel{+}= \sum_{h'=1}^{H} \delta_{h'}^{(p_1,\ldots,p_d+1,\ldots,p_n)} w_{hh'}^{d}$
        $\delta_h^{\mathbf{p}} = \theta_h'(e_h^{\mathbf{p}})$
```

Algorithm 8.2: MDRNN Backward Pass

The ‘loop over loops’ in these algorithms can be naturally implemented with recursive functions.

Defining $\mathbf{p}_d^- \overset{\text{def}}{=} (p_1, \ldots, p_d - 1, \ldots, p_n)$ and $\mathbf{p}_d^+ \overset{\text{def}}{=} (p_1, \ldots, p_d + 1, \ldots, p_n)$, the above procedures can be compactly expressed as follows:

Forward Pass

$$a_h^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{ih} + \sum_{\substack{d=1:\\ p_d>0}}^{n} \sum_{h'=1}^{H} b_{h'}^{\mathbf{p}_d^-} w_{h'h}^d \tag{8.2}$$

$$b_h^{\mathbf{p}} = \theta_h(a_h^{\mathbf{p}}), \tag{8.3}$$

Backward Pass

$$\delta_h^{\mathbf{p}} = \theta_h'(a_h^{\mathbf{p}}) \left(\sum_{k=1}^{K} \delta_k^{\mathbf{p}} w_{hk} + \sum_{\substack{d=1:\\ p_d<D_d-1}}^{n} \sum_{h'=1}^{H} \delta_{h'}^{\mathbf{p}_d^+} w_{hh'}^d \right) \tag{8.4}$$

Since the forward and backward pass require one pass each through the data sequence, the overall complexity of MDRNN training is linear in the number of data points and the number of network weights.

In the special case where $n = 1$, the above equations reduce to those of a standard RNN (see Section 3.2).

8.2.1 Multidirectional MDRNNs

At some point $(p_1, \ldots, p_n)$ in the input sequence, the network described above has access to all points $(p_1', \ldots, p_n')$ such that $p_d' \leq p_d \; \forall \; d \in (1, \ldots, n)$. This defines an n-dimensional 'context region' of the full sequence, as illustrated in Figure 8.4. For some tasks, such as object recognition, this would in principle be sufficient. The network could process the image according to the ordering, and output the object label at a point when the object to be recognised is entirely contained in the context region.

Intuitively however, we would prefer the network to have access to the surrounding context in all directions. This is particularly true for tasks where precise localisation is required, such as image segmentation. As discussed in Chapter 3, for one dimensional RNNs, the problem of multidirectional context was solved by the introduction of bidirectional recurrent neural networks (BRNNs). BRNNs contain two separate hidden layers that process the input sequence in the forward and reverse directions. The two hidden layers are connected to a single output layer, thereby providing the network with access to both past and future context.

BRNNs can be extended to n-dimensional data by using 2^n separate hidden layers, each of which processes the sequence using the ordering defined above, but with a different choice of axes. The axes are chosen so that each

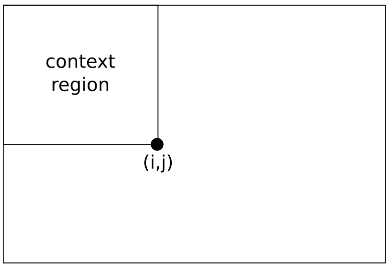


Figure 8.4: **Context available at (i,j) to a 2D RNN with a single hidden layer**

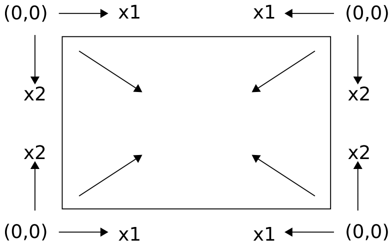


Figure 8.5: **Axes used by the 4 hidden layers in a multidirectional 2D RNN.** The arrows inside the rectangle indicate the direction of propagation during the forward pass.

one has its origin on a distinct vertex of the sequence. The 2 dimensional case is illustrated in Figure 8.5. As before, the hidden layers are connected to a single output layer, which now has access to all surrounding context (see Figure 8.6).

Clearly, if the size of the hidden layers is held constant, the complexity of the multidirectional MDRNN architecture scales as $O(2^n)$ for n-dimensional data. In practice however, the computing power of the network tends to be governed by the overall number of weights, rather than the size of the hidden layers, because the data processing is shared between the layers. Since the complexity of the algorithm is linear in the number of parameters, the $O(2^n)$ scaling factor can be offset by simply using smaller hidden layers for higher dimensions. Furthermore, the complexity of a task, and therefore the number of weights likely to be needed for it, does not necessarily increase with the dimensionality of the data. For example, both the networks described in this chapter have less than half the weights than the one dimensional networks we applied to speech recognition in Chapters 5–7. For a given task, we have also found that using a multidirectional MDRNN gives better results than a unidirectional MDRNN with the same overall number of weights, as demonstrated for one dimension in Chapter 5.

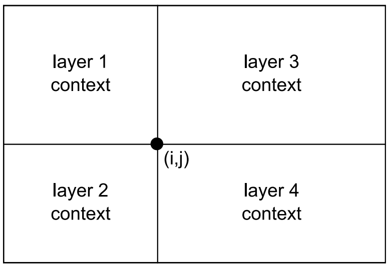


Figure 8.6: **Context available at (i,j) to a multidirectional 2D RNN**

In fact, the main scaling concern for MDRNNs is that there tend to be many more data points in higher dimensions (e.g. a video sequence contains far more pixels than an image). However this can be alleviated by gathering together the inputs into multidimensional 'blocks' — for example 8 by 8 by 8 pixels for a video.

For a multidirectional MDRNN, the forward and backward passes through an n-dimensional sequence can be summarised as follows:

```
For each of the $2^n$ hidden layers choose a distinct vertex of the sequence, then define a set of axes such that the vertex is the origin and all sequence coordinates are $\geq 0$
Repeat Algorithm 8.1 for each hidden layer
At each point in the sequence, feed forward all hidden layers to the output layer
```

Algorithm 8.3: Multidirectional MDRNN Forward Pass

```
At each point in the sequence, calculate the derivative of the objective function with respect to the activations of output layer
With the same axes as above, repeat Algorithm 8.2 for each hidden layer
```

Algorithm 8.4: Multidirectional MDRNN Backward Pass

8.2.2 Multidimensional Long Short-Term Memory

The standard formulation of LSTM is explicitly one-dimensional, since the cell contains a single self connection, whose activation is controlled by a single forget gate. However we can extend this to n dimensions by using instead n self connections (one for each of the cell's previous states along every dimension) with n forget gates. The suffix ι, d denotes the forget gate corresponding to connection d. As before, peephole connections lead from the cells to the gates. Note however that the input gates ι is connected to previous cell c along all dimensions with the same weight ($w_{c\iota}$) whereas each forget gate d is only connected to cell c along dimension d, with a separate weight $w_{c(\iota,d)}$ for each d. The peephole to the output gate receives input from the *current* state, and therefore requires only a single weight.

Combining the above notation with that of Sections 4.5 and 8.2, the equations for training multidimensional LSTM can be written as follows:

Forward Pass

Input Gates

$$a_\iota^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{i\iota} + \sum_{\substack{d=1:\\ p_d>0}}^{n} \left(\sum_{h=1}^{H} b_h^{\mathbf{p}_d^-} w_{h\iota}^d + \sum_{c=1}^{C} w_{c\iota} s_c^{\mathbf{p}_d^-} \right) \tag{8.5}$$

$$b_\iota^{\mathbf{p}} = f(a_\iota^{\mathbf{p}}) \tag{8.6}$$

Forget Gates

$$a_{\phi,d}^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{i(\phi,d)} + \sum_{\substack{d'=1:\\ p_{d'}>0}}^{n} \sum_{h=1}^{H} b_h^{\mathbf{p}_{d'}^-} w_{h(\phi,d)}^{d'} + \begin{cases} \sum_{c=1}^{C} w_{c(\phi,d)} s_c^{\mathbf{p}_d^-} & \text{if } p_d > 0 \\ 0 \text{ otherwise} \end{cases} \tag{8.7}$$

$$b_{\phi,d}^{\mathbf{p}} = f(a_{\phi,d}^{\mathbf{p}}) \tag{8.8}$$

Cells

$$a_c^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{ic} + \sum_{\substack{d=1:\\ p_d>0}}^{n} \sum_{h=1}^{H} b_h^{\mathbf{p}_d^-} w_{hc}^d \tag{8.9}$$

$$s_c^{\mathbf{p}} = b_\iota^{\mathbf{p}} g(a_c^{\mathbf{p}}) + \sum_{\substack{d=1:\\ p_d>0}}^{n} s_c^{\mathbf{p}_d^-} b_{\phi,d}^{\mathbf{p}} \tag{8.10}$$

Output Gates

$$a_\omega^{\mathbf{p}} = \sum_{i=1}^{I} x_i^{\mathbf{p}} w_{i\omega} + \sum_{\substack{d=1:\\ p_d>0}}^{n} \sum_{h=1}^{H} b_h^{\mathbf{p}_d^-} w_{h\omega}^d + \sum_{c=1}^{C} w_{c\omega} s_c^{\mathbf{p}} \tag{8.11}$$

$$b_\omega^{\mathbf{p}} = f(a_\omega^{\mathbf{p}}) \tag{8.12}$$

Cell Outputs

$$b_c^{\mathbf{p}} = b_\omega^{\mathbf{p}} h(s_c^{\mathbf{p}}) \tag{8.13}$$

Backward Pass

$$\epsilon_c^{\mathbf{p}} \overset{\text{def}}{=} \frac{\partial O}{\partial b_c^{\mathbf{p}}} \qquad \epsilon_s^{\mathbf{p}} \overset{\text{def}}{=} \frac{\partial O}{\partial s_c^{\mathbf{p}}} \tag{8.14}$$

Cell Outputs

$$\epsilon_c^{\mathbf{p}} = \sum_{k=1}^{K} \delta_k^{\mathbf{p}} w_{ck} + \sum_{\substack{d=1: \\ p_d < D_d - 1}}^{n} \sum_{h=1}^{H} \delta_h^{\mathbf{p}_d^+} w_{ch}^d \tag{8.15}$$

Output Gates

$$\delta_\omega^{\mathbf{p}} = f'(a_\omega^{\mathbf{p}}) \sum_{c=1}^{C} \epsilon_c^{\mathbf{p}} h(s_c^{\mathbf{p}}) \tag{8.16}$$

States

$$\epsilon_s^{\mathbf{p}} = b_\omega^{\mathbf{p}} h'(s_c^{\mathbf{p}}) \epsilon_c^{\mathbf{p}} + \delta_\omega^{\mathbf{p}} w_{c\omega} + \sum_{\substack{d=1: \\ p_d < D_d - 1}}^{n} \left(\epsilon_s^{\mathbf{p}_d^+} b_{\phi,d}^{\mathbf{p}_d^+} + \delta_\iota^{\mathbf{p}_d^+} w_{c\iota} + \delta_{\phi,d}^{\mathbf{p}_d^+} w_{c(\phi,d)} \right) \tag{8.17}$$

Cells

$$\delta_c^{\mathbf{p}} = b_\iota^{\mathbf{p}} g'(a_c^{\mathbf{p}}) \epsilon_s^{\mathbf{p}} \tag{8.18}$$

Forget Gates

$$\delta_{\phi,d}^{\mathbf{p}} = \begin{cases} f'(a_{\phi,d}^{\mathbf{p}}) \sum_{c=1}^{C} s_c^{\mathbf{p}_d^-} \epsilon_s^{\mathbf{p}} \text{ if } p_d > 0 \\ 0 \text{ otherwise} \end{cases} \tag{8.19}$$

Input Gates

$$\delta_\iota^{\mathbf{p}} = f'(a_\iota^{\mathbf{p}}) \sum_{c=1}^{C} g(a_c^{\mathbf{p}}) \epsilon_s^{\mathbf{p}} \tag{8.20}$$

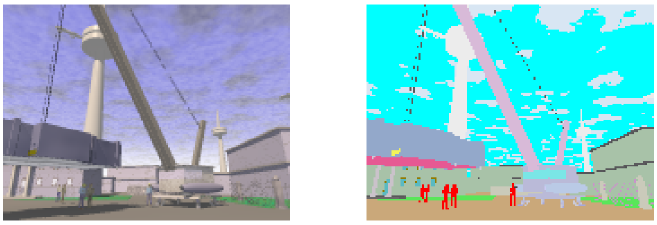


Figure 8.7: **Frame from the Air Freight database.** The original image is on the left and the colour-coded texture segmentation is on the right.

8.3 Experiments

8.3.1 Air Freight Data

The Air Freight database (McCarter and Storkey, 2007) is a ray-traced colour image sequence that comes with a ground truth segmentation into the different textures mapped onto the 3-d models (Figure 8.7). The sequence is 455 frames long and contains 155 distinct textures. Each frame is 120 pixels high and 160 pixels wide.

The advantage of ray-traced data is the true segmentation can be defined directly from the 3D models. Although the images are not real, they are realistic in the sense that they have significant lighting, specular effects etc.

We used the sequence to define a 2D image segmentation task, where the aim was to assign each pixel in the input data to the correct texture class. We divided the data at random into a 250 frame train set, a 150 frame test set and a 55 frame validation set. We could have instead defined a 3D task where the network processed segments of the video as independent sequences. However, this would have left us with fewer exemplars for training and testing.

For this task we used a multidirectional 2D RNN with LSTM hidden layers. Each of the 4 layers consisted of 25 memory blocks, each containing 1 cell, 2 forget gates, 1 input gate, 1 output gate and 5 peephole weights. This gave a total 600 hidden units. *tanh* was used for the input and output activation functions of the cells, and the activation function for the gates was the logistic sigmoid. The input layer was size 3 (one each for the red, green and blue components of the pixels) and the output layer was size 155 (one unit for each texture). The network contained 43,257 trainable weights in total. The softmax activation function was used at the output layer, with the cross-entropy objective function (Section 3.1.3). The network was trained using online gradient descent (weight updates after every training sequence) with a learning rate of 10^{-6} and a momentum of 0.9.

The final pixel classification error rate, after 330 training epochs, was 7.1% on the test set.

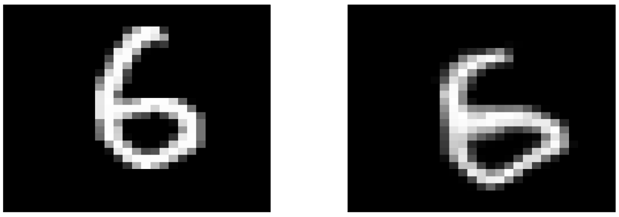


Figure 8.8: **MNIST image before and after deformation**

8.3.2 MNIST Data

The MNIST database (LeCun et al., 1998a) of isolated handwritten digits is a subset of a larger database available from NIST. It consists of size-normalised, centred images, each of which is 28 pixels high and 28 pixels wide and contains a single handwritten digit. The data comes divided into a training set with 60,000 images and a test set with 10,000 images. We used 10,000 of the training images for validation, leaving 50,000 for training.

The task on MNIST is to label the images with the corresponding digits. This is a well-known benchmark for which many pattern classification algorithms have been evaluated.

We trained the network to perform a slightly modified task where each pixel was classified according to the digit it belonged to, with an additional class for background pixels. We then recovered the original task by choosing for each sequence the digit whose corresponding output unit had the highest cumulative activation over the entire sequence.

To test the network's robustness to input warping, we also evaluated it on an altered version of the MNIST test set, where elastic deformations had been applied to every image (see Figure 8.8). The deformations were the same as those used by Simard (2003) to augment the MNIST training set, with parameters $\sigma = 4.0$ and $\alpha = 34.0$, and using a different initial random field for every sample image.

We compared our results with the convolution neural network that has achieved the best results so far on MNIST (Simard et al., 2003). Note that we re-implemented the convolution network ourselves, and we did not augment the training set with elastic distortions, which gives a substantial improvement in performance.

The MDRNN for this task was identical to that for the Air Freight task with the following exceptions: the sizes of the input and output layers were now 1 (for grayscale pixels) and 11 (one for each digit, plus background) respectively, giving 27,511 weights in total, and the learning rate was 10^{-5}.

Table 8.1 shows that the MDRNN matched the convolution network on the clean test set, and was considerably better on the warped test set. This suggests that MDRNNs are more robust to input warping than convolution

Table 8.1: **Image error rate on MNIST**

Algorithm	Clean Test Set	Warped Test Set
Convolution	0.9%	11.3%
MDRNN	0.9%	7.1%

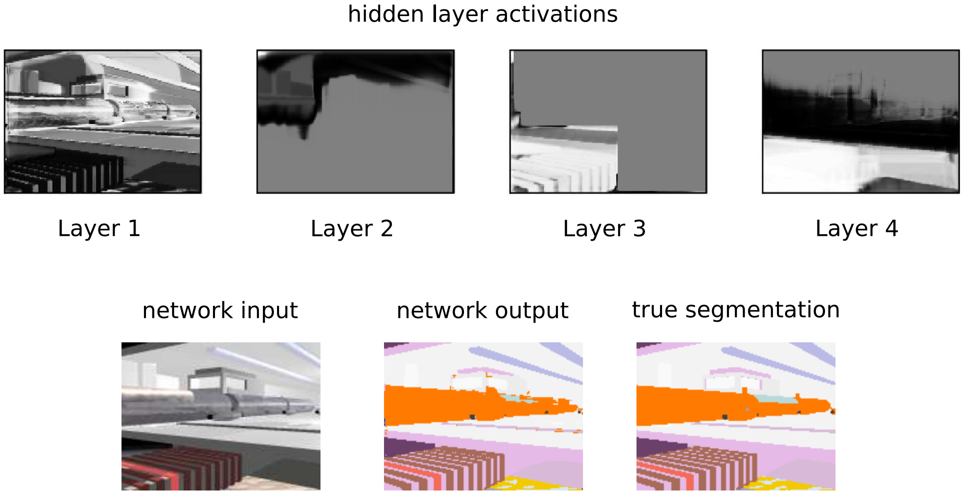


Figure 8.9: **2D RNN applied to an image from the Air Freight database.** The hidden layer activations display one unit from each of the layers. A common behaviour is to 'mask off' parts of the image, exhibited here by layers 2 and 3.

networks. The pixel classification error rates for the MDRNN were 0.4% for the clean test set and 3.8% for the warped test set.

However one area in which the convolution net greatly outperformed the MDRNN was training time. The MDRNN required 95 training epochs to converge, whereas the convolution network required 20. Furthermore, each training epoch took approximately 3.7 hours for the MDRNN compared to under ten minutes for the convolution network, using a similar processor. The total training time for the MDRNN was over two weeks.

8.3.3 Analysis

One benefit of two dimensional tasks is that the operation of the network can be easily visualised. Figure 8.9 shows the network activations during a frames from the Air Freight database. As can be seen, the network segments this image almost perfectly, in spite of difficult, reflective surfaces such as the glass and metal tube running from left to right. Clearly, classifying

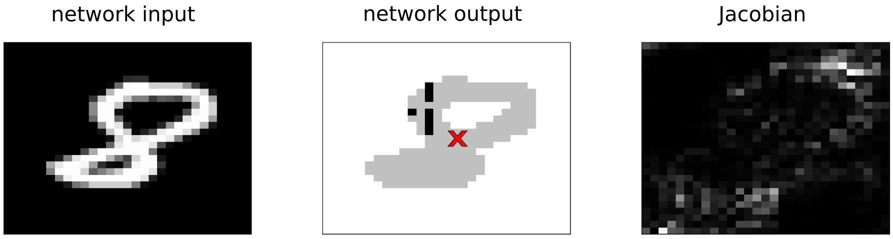


Figure 8.10: **Sequential Jacobian of a 2D RNN for an image from MNIST.** The white outputs correspond to the class 'background' and the light grey ones to '8'. The black outputs represent misclassifications. The output pixel for which the Jacobian is calculated is marked with a cross. Absolute values are plotted for the Jacobian, and lighter colours are used for higher values.

individual pixels in such a surface requires the use of contextual information.

Figure 8.10 shows the absolute value of the sequential Jacobian of an output during classification of an image from the MNIST database. It can be seen that the network responds to context from across the entire image, and seems particularly attuned to the outline of the digit.

Chapter 9

Conclusions and Future Work

The aim of this thesis was to advance the state-of-the-art in supervised sequence labelling with RNNs. In particular, it focused on applying and extending the LSTM RNN architecture. As well as presenting an exact calculation of LSTM error gradient, we introduced bidirectional LSTM and demonstrated its advantage over other neural network architectures in a real-world segment classification task. We investigated the use of HMM-LSTM hybrids on a real-world temporal classification task. We introduced the connectionist temporal classification (CTC) output layer, which allows RNNs to be trained directly for sequence labelling tasks with unknown input-output alignments. We presented an efficient decoding algorithm for CTC in the case where a dictionary and a bigram language model is used to constrain the outputs. We found that a CTC network outperformed both HMMs and HMM-RNN hybrids on several experiments in speech and handwriting recognition.

Lastly, we discussed the how RNNs can be extended to data with several spatio-temporal dimensions using multidimensional RNNs (MDRNNs). We showed how multidirectional MDRNNs are able to access to context along all input directions, and we introduced multidimensional LSTM, thereby bringing the benefits of long range conext to MDRNNs. We successfully applied 2-dimensional LSTM networks to two tasks in image segmentation, and achieved better generalisation to distorted images than a state-of-the-art image recognition algorithm.

In the future, we would to like to investigate the use of MDRNNs for data with more than two spatio-temporal dimensions, such as videos and sequences of fMRI brain scans. We would also like to combine MDRNNs with advanced visual preprocessing techniques to see how they compare with the state-of-the-art in computer vision.

Another direction we would like to pursue is the use of RNNs for hierarchical sequence learning. In our experiments so far, higher level constraints such as language models and dictionaries have been applied externally to the CTC outputs, during decoding. Ideally, however, the network would be able to learn both high and low level constraints directly from the data. One way to do this would be to create a hierarchy of RNNs in which the detection of progressively more complex patterns is carried out by progressively higher level networks. Our preliminary research in this direction has been encouraging (Fernández et al., 2007b).

Lastly, we would like to investigate the use of RNNs for unsupervised learning. One straightforward approach would be to use RNNs to predict future inputs for sequential data. This may overlap with hierarchical learning, since a possible technique for ensuring that complex patterns are efficiently decomposed into multiple levels would be for each level to attempt to predict the behaviour of the levels above and below it.

Bibliography

G. An. The effects of adding noise during backpropagation training on a generalization performance. *Neural Comput.*, 8(3):643–674, 1996. ISSN 0899-7667.

B. Bakker. Reinforcement learning with Long Short-Term Memory. In *Advances in Neural Information Processing Systems, 14*, 2002.

P. Baldi and G. Pollastri. The principled design of large-scale recursive neural network architectures–dag-rnns and the protein structure prediction problem. *J. Mach. Learn. Res.*, 4:575–602, 2003. ISSN 1533-7928.

P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri. Exploiting the past and the future in protein secondary structure prediction. *BIOINF: Bioinformatics*, 15, 1999. URL `citeseer.ist.psu.edu/article/baldi99exploiting.html`.

P. Baldi, S. Brunak, P. Frasconi, G. Pollastri, and G. Soda. Bidirectional dynamics for protein secondary structure prediction. *Lecture Notes in Computer Science*, 1828:80–104, 2001. URL `citeseer.ist.psu.edu/baldi99bidirectional.html`.

Y. Bengio. A connectionist approach to speech recognition. *International Journal on Pattern Recognition and Artificial Intelligence*, 7(4):647–668, 1993. URL `http://www.iro.umontreal.ca/~lisa/pointeurs/ijprai93.ps`.

Y. Bengio. Markovian models for sequential data. *Neural Computing Surveys*, 2:129–162, 1999. URL `http://www.iro.umontreal.ca/~lisa/pointeurs/hmms.ps`.

Y. Bengio and Y. LeCun. Scaling learning algorithms towards ai. In L. Bottou, O. Chapelle, D. DeCoste, and J. Weston, editors, *Large-Scale Kernel Machines*. MIT Press, 2007.

Y. Bengio, R. De Mori, G. Flammia, and R. Kompe. Global optimization of a neural network–Hidden Markov Model hybrid. *IEEE Transactions on*

Neural Networks, 3(2):252–259, March 1992. URL citeseer.ist.psu.edu/bengio91global.html.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, March 1994. URL citeseer.ist.psu.edu/bengio94learning.html.

Y. Bengio, Y. LeCun, C. Nohl, and C. Burges. LeRec: A NN/HMM hybrid for on-line handwriting recognition. *Neural Computation*, 7(6):1289–1303, 1995. URL citeseer.ist.psu.edu/bengio95lerec.html.

N. Beringer. Human language acquisition in a machine learning task. *Proc. ICSLP*, 2004.

R. Bertolami and H. Bunke. Multiple classifier methods for offline handwritten text line recognition. In *7th International Workshop on Multiple Classifier Systems, Prague, Czech Republic*, 2007.

C. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, Inc., 1995. ISBN 0198538642.

C. M. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.

L. Bottou and Y. LeCun. Graph transformer networks for image recognition. In *Proceedings of ISI*, 2005. (invited paper).

H. Bourlard and N. Morgan. *Connnectionist Speech Recognition: A Hybrid Approach.* Kluwer Academic Publishers, 1994.

H. Bourlard, Y. Konig, N. Morgan, and C. Ris. A new training algorithm for hybrid hmm/ann speech recognition systems. In *8th European Signal Processing Conference*, volume 1, pages 101–104, 1996.

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

D. Broomhead and D. Lowe. Multivariate functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

R. H. Byrd, P. Lu, J. Nocedal, and C. Y. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(6):1190–1208, 1995. URL citeseer.ist.psu.edu/byrd94limited.html.

J. chang. *Near-Miss Modeling: A Segment-Based Approach to Speech Recognition.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

J. Chen and N. Chaudhari. Protein secondary structure prediction with bidirectional lstm networks. In *Post-Conference Workshop on Computational Intelligence Approaches for the Analysis of Bio-data (CI-BIO)*, Montreal, Canada, August 2005.

J. Chen and N. S. Chaudhari. Capturing long-term dependencies for protein secondary structure prediction. In F. Yin, J. Wang, and C. Guo, editors, *Advances in Neural Networks - ISNN 2004, International Symposiumon Neural Networks, Part II*, volume 3174 of *Lecture Notes in Computer Science*, pages 494–500, Dalian, China, 2004. Springer. ISBN 3-540-22843-8.

R. Chen and L. Jamieson. Experiments on the implementation of recurrent neural networks for speech phone recognition. In *Proceedings of the Thirtieth Annual Asilomar Conference on Signals, Systems and Computers*, pages 779–782, 1996.

D. Decoste and B. Schölkopf. Training invariant support vector machines. *Machine Learning*, 46(1-3):161–190, 2002. URL `citeseer.ist.psu.edu/decoste02training.html`.

R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* Wiley-Interscience Publication, 2000. ISBN 0471056693.

D. Eck and J. Schmidhuber. Finding temporal structure in music: Blues improvisation with LSTM recurrent networks. In H. Bourlard, editor, *Neural Networks for Signal Processing XII, Proceedings of the 2002 IEEE Workshop*, pages 747–756, New York, 2002. IEEE.

J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

S. Fahlman. Faster learning variations on back-propagation: An empirical study. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 connectionist models summer school*, pages 38–51, San Mateo, 1989. Morgan Kaufmann.

S. Fernández, A. Graves, and J. Schmidhuber. An application of recurrent neural networks to discriminative keyword spotting. In *Proceedings of the 2007 International Conference on Artificial Neural Networks*, Porto, Portugal, September 2007a.

S. Fernández, A. Graves, and J. Schmidhuber. Sequence labelling in structured domains with hierarchical recurrent neural networks. In *Proceedings* *of the 20th International Joint Conference on Artificial Intelligence, IJ-CAI 2007*, Hyderabad, India, 2007b.

T. Fukada, M. Schuster, and Y. Sagisaka. Phoneme boundary estimation using bidirectional recurrent neural networks and its applications. *Systems and Computers in Japan*, 30(4):20–30, 1999.

J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, D. S. Pallett, , and N. L. Dahlgren. Darpa timit acoustic phonetic continuous speech corpus cdrom, 1993.

F. Gers. *Long Short-Term Memory in Recurrent Neural Networks.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, 2001. URL `citeseer.nj.nec.com/article/gers01long.html`.

F. Gers, N. Schraudolph, and J. Schmidhuber. Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research*, 3: 115–143, 2002.

F. A. Gers and J. Schmidhuber. LSTM recurrent networks learn simple context free and context sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340, 2001.

F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Computation*, 12(10):2451–2471, 2000.

C. Giraud-Carrier, R. Vilalta, and P. Brazdil. Introduction to the special issue on meta-learning. *Mach. Learn.*, 54(3):187–193, 2004. ISSN 0885-6125.

J. R. Glass. A probabilistic framework for segment-based speech recognition. *Computer Speech and Language*, 17:137–152, 2003.

A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM networks. In *Proceedings of the 2005 International Joint Conference on Neural Networks*, Montreal, Canada, 2005a.

A. Graves and J. Schmidhuber. Framewise phoneme classification with bidirectional LSTM and other neural network architectures. *Neural Networks*, 18(5-6):602–610, June/July 2005b.

A. Graves, N. Beringer, and J. Schmidhuber. Rapid retraining on speech data with lstm recurrent networks. Technical Report IDSIA-09-05, IDSIA, www.idsia.ch/techrep.html, 2005a.

A. Graves, S. Fernández, and J. Schmidhuber. Bidirectional LSTM networks for improved phoneme classification and recognition. In *Proceedings of the 2005 International Conference on Artificial Neural Networks*, Warsaw, Poland, 2005b.

A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber. Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the International Conference on Machine Learning, ICML 2006*, Pittsburgh, USA, 2006.

A. Graves, S. Fernández, and J. Schmidhuber. Multi-dimensional recurrent neural networks. In *Proceedings of the 2007 International Conference on Artificial Neural Networks*, Porto, Portugal, September 2007.

A. Graves, S. Fernández, M. Liwicki, H. Bunke, and J. Schmidhuber. Unconstrained online handwriting recognition with recurrent neural networks. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems 20*. MIT Press, Cambridge, MA, 2008a.

A. Graves, M. Liwicki, S. Fernández, R. Bertolami, H. Bunke, and J. Schmidhuber. A novel connectionist system for unconstrained handwriting recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 2008b. To appear.

A. K. Halberstadt. *Heterogeneous acoustic measurements and multiple classifiers for speech recognition.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

B. Hammer. On the approximation capability of recurrent neural networks. *Neurocomputing*, 31(1-4):107–123, 2000.

J. Hennebert, C. Ris, H. Bourlard, S. Renals, and N. Morgan. Estimation of global posteriors and forward-backward training of hybrid HMM/ANN systems. In *Proc. of the European Conference on Speech Communication and Technology (Eurospeech 97)*, pages 1951–1954, Rhodes, 1997. URL `http://www.dcs.shef.ac.uk/~{}sjr/pubs/1997/eurosp97-remap.html`.

M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of Research of the National Bureau of Standards*, 49(6):409–436, 1952.

G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, 2006. ISSN 0899-7667.

S. Hochreiter. *Untersuchungen zu dynamischen neuronalen Netzen.* PhD thesis, Institut für Informatik, Technische Universität München, 1991. See http://ni.cs.tu-berlin.de/∼hochreit/papers/hochreiter.dipl.ps.gz.

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

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In S. C. Kremer and J. F. Kolen, editors, *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE Press, 2001.

S. Hochreiter, M. Heusel, and K. Obermayer. Fast Model-based Protein Homology Detection without Alignment. *Bioinformatics*, 2007.

J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. *PNAS*, 79(8):2554–2558, April 1982.

K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Netw.*, 2(5):359–366, 1989. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/0893-6080(89)90020-8.

F. Hülsken, F. Wallhoff, and G. Rigoll. Facial expression recognition with pseudo-3d hidden markov models. In *Proceedings of the 23rd DAGM-Symposium on Pattern Recognition*, pages 291–297, London, UK, 2001. Springer-Verlag. ISBN 3-540-42596-9.

H. Jaeger. The "echo state" approach to analysing and training recurrent neural networks. Technical Report GMD Report 148, German National Research Center for Information Technology, 2001. URL `http://www.faculty.iu-bremen.de/hjaeger/pubs/EchoStatesTechRep.pdf`.

J. Jiten, B. Mérialdo, and B. Huet. Multi-dimensional dependency-tree hidden Markov models. In *ICASSP 2006, 31st IEEE International Conference on Acoustics, Speech, and Signal Processing, May 14-19, 2006, Toulouse, France*, May 2006. doi: http://ieeexplore.ieee.org/servlet/opac?punumber=11024.

S. Johansson, R. Atwell, R. Garside, and G. Leech. The tagged LOB corpus user's manual; Norwegian Computing Centre for the Humanities, 1986.

M. T. Johnson. Capacity and complexity of HMM duration modeling techniques. *IEEE Signal Processing Letters*, 12(5):407–410, May 2005.

M. I. Jordan. *Attractor dynamics and parallelism in a connectionist sequential machine*, pages 112–127. IEEE Press, Piscataway, NJ, USA, 1990. ISBN 0-8186-2015-3.

D. Joshi, J. Li, and J. Wang. Parameter estimation of multi-dimensional hidden markov models: A scalable approach. In *Proc. of the IEEE International Conference on Image Processing (ICIP05)*, pages III: 149–152, 2005.

M. W. Kadous. *Temporal Classification: Extending the Classification Paradigm to Multivariate Time Series.* PhD thesis, School of Computer Science & Engineering, University of New South Wales, 2002. URL `http://www.cse.unsw.edu.au/~waleed/phd/html/phd.html`.

D. Kershaw, A. Robinson, and M. Hochberg. Context-dependent classes in a hybrid recurrent network-HMM speech recognition system. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 750–756. The MIT Press, 1996. URL `citeseer.ist.psu.edu/kershaw95contextdependent.html`.

T. Kohonen. *Self-organization and associative memory: 3rd edition.* Springer-Verlag New York, Inc., New York, NY, USA, 1989. ISBN 0-387-51387-6.

P. Koistinen and L. Holmström. Kernel regression and backpropagation training with noise. In J. E. Moody, S. J. Hanson, and R. Lippmann, editors, *Advances in Neural Information Processing Systems, 4*, pages 1033–1039. Morgan Kaufmann, 1991. URL `http://dblp.uni-trier.de/db/conf/nips/nips1991.html#KoistinenH91`.

L. Lamel and J. Gauvain. High performance speaker-independent phone recognition using cdhmm. In *Proc. Eurospeech*, Berlin, Germany, September 1993.

K. J. Lang, A. H. Waibel, and G. E. Hinton. A time-delay neural network architecture for isolated word recognition. *Neural Netw.*, 3(1):23–43, 1990. ISSN 0893-6080. doi: http://dx.doi.org/10.1016/0893-6080(90)90044-L.

Y. LeCun, L. Bottou, and Y. Bengio. Reading checks with graph transformer networks. In *International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 151–154, Munich, 1997. IEEE.

Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998a.

Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and M. K., editors, *Neural Networks: Tricks of the trade.* Springer, 1998b.

K.-F. Lee and H.-W. Hon. Speaker-independent phone recognition using hidden Markov models. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(11):1641–1648, November 1989.

J. Li, A. Najmi, and R. M. Gray. Image classification by a two-dimensional hidden markov model. *IEEE Transactions on Signal Processing*, 48(2): 517–533, 2000. URL `citeseer.ist.psu.edu/article/li98image.html`.

T. Lin, B. G. Horne, P. Tiño, and C. L. Giles. Learning long-term dependencies in NARX recurrent neural networks. *IEEE Transactions on Neural Networks*, 7(6):1329–1338, November 1996. URL `citeseer.ist.psu.edu/lin96learning.html`.

T. Lindblad and J. M. Kinser. *Image Processing Using Pulse-Coupled Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. ISBN 354024218X.

M. Liwicki and H. Bunke. Handwriting recognition of whiteboard notes. In *Proc. 12th Conf. of the International Graphonomics Society*, pages 118–122, 2005a.

M. Liwicki and H. Bunke. IAM-OnDB - an on-line English sentence database acquired from handwritten text on a whiteboard. In *Proc. 8th Int. Conf. on Document Analysis and Recognition*, volume 2, pages 956–961, 2005b.

M. Liwicki, A. Graves, S. Fernández, H. Bunke, and J. Schmidhuber. A novel approach to on-line handwriting recognition based on bidirectional long short-term memory networks. In *Proceedings of the 9th International Conference on Document Analysis and Recognition, ICDAR 2007*, Curitiba, Brazil, September 2007.

D. MacKay. Probable networks and plausible predictions - a review of practical bayesian methods for supervised neural networks. *Network: Computation in Neural Systems*, 6:469–505, 1995. URL `citeseer.ist.psu.edu/64617.html`.

U.-V. Marti and H. Bunke. Using a statistical language model to improve the performance of an HMM-based cursive handwriting recognition system. *Int. Journal of Pattern Recognition and Artificial Intelligence*, 15:65–90, 2001.

U.-V. Marti and H. Bunke. The IAM-database: an English sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5:39 – 46, 2002.

G. McCarter and A. Storkey. Air Freight Image Segmentation Database. `http://homepages.inf.ed.ac.uk/amos/afreightdata.html`, 2007.

W. S. McCulloch and W. Pitts. *A logical calculus of the ideas immanent in nervous activity*, pages 15–27. MIT Press, Cambridge, MA, USA, 1988. ISBN 0-262-01097-6.

J. Ming and F. J. Smith. Improved Phone Recognition Using Bayesian Triphone Models. In *ICASSP*, volume 1, pages 409–412, 1998.

M. C. Mozer. Induction of multiscale temporal structure. In J. E. Moody, S. J. Hanson, and R. P. Lippmann, editors, *Advances in Neural Information Processing Systems*, volume 4, pages 275–282. Morgan Kaufmann Publishers, Inc., 1992. URL `citeseer.ist.psu.edu/mozer92induction.html`.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

R. M. Neal. *Bayesian Learning for Neural Networks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0387947248.

J. Neto, L. Almeida, M. Hochberg, C. Martins, L. Nunes, S. Renals, and A. Robinson. Speaker adaptation for hybrid hmm-ann continuous speech recognition system. In *Proceedings of Eurospeech 1995*, volume 1, pages 2171–2174, 1995.

X. Pang and P. J. Werbos. Neural network design for J function approximation in dynamic programming. *Mathematical Modeling and Scientific Computing*, 5(2/3), 1996. URL `http://citeseer.ist.psu.edu/395025.html`. Also available at `http://arxiv.org/abs/adap-org/9806001`.

T. A. Plate. Holographic recurrent networks. In C. L. Giles, S. J. Hanson, and J. D. Cowan, editors, *Advances in Neural Information Processing Systems 5: NIPS * 92, Denver, CO, November 1992*, pages 34–41, San Mateo, CA, 1993. Morgan Kaufmann. URL `citeseer.ist.psu.edu/plate93holographic.html`.

D. C. Plaut, S. J. Nowlan, and G. E. Hinton. Experiments on learning back propagation. Technical Report CMU–CS–86–126, Carnegie–Mellon University, Pittsburgh, PA, 1986.

G. Pollastri, A. Vullo, P. Frasconi, and P. Baldi. Modular dag-rnn architectures for assembling coarse protein structures. *J Comput Biol*, 13(3): 631–650, April 2006. ISSN 1066-5277. doi: 10.1089/cmb.2006.13.631. URL `http://dx.doi.org/10.1089/cmb.2006.13.631`.

L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proc. IEEE*, 77(2):257–286, 1989.

S. Renals, N. Morgan, H. Bourlard, M. Cohen, and H. Franco. Connectionist probability estimators in HMM speech recognition. *IEEE Transactions Speech and Audio Processing*, 1993. URL `citeseer.ist.psu.edu/renals94connectionist.html`.

M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The RPROP algorithm. In *Proc. of the IEEE Intl.*

Conf. on Neural Networks, pages 586–591, San Francisco, CA, 1993. URL `citeseer.ist.psu.edu/riedmiller93direct.html`.

A. Robinson, J. Holdsworth, J. Patterson, and F. Fallside. A comparison of preprocessors for the cambridge recurrent error propagation network speech recognition system. In *Proceedings of the First International Conference on Spoken Language Processing, ICSLP-1990*, pages Kobe, Japan, Vietri (Italy), 1990.

A. J. Robinson. Several improvements to a recurrent error propagation network phone recognition system. Technical Report CUED/F-INFENG/TR82, University of Cambridge, September 1991.

A. J. Robinson. An application of recurrent nets to phone probability estimation. *IEEE Transactions on Neural Networks*, 5(2):298–305, March 1994. URL `citeseer.nj.nec.com/robinson94application.html`.

A. J. Robinson and F. Fallside. The utility driven dynamic error propagation network. Technical Report CUED/F-INFENG/TR.1, Cambridge University Engineering Department, 1987.

A. J. Robinson, L. Almeida, J.-M. Boite, H. Bourlard, F. Fallside, M. Hochberg, D. Kershaw, P. Kohn, Y. Konig, N. Morgan, J. P. Neto, S. Renals, M. Saerens, and C. Wooters. A neural network based, speaker independent, large vocabulary, continuous speech recognition system: the Wernicke project. In *Proc. of the Third European Conference on Speech Communication and Technology (Eurospeech 93)*, pages 1941–1944, Berlin, 1993.

F. Rosenblatt. *Principles of Neurodynamics.* Spartan, New York, 1963.

F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65:386–408, 1958.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

J. Schmidhuber. Learning complex extended sequences using the principle of history compression. *Neural Computing*, 4(2):234–242, 1992. URL `citeseer.ist.psu.edu/article/schmidhuber92learning.html`.

J. Schmidhuber, D. Wierstra, M. Gagliolo, and F. Gomez. Training recurrent networks by evolino. *Neural Computation*, 19(3):757–779, 2007. ISSN 0899-7667.

N. Schraudolph. Fast curvature matrix-vector products for second-order gradient descent. *Neural Computation*, 14(7):1723–1738, 2002.

M. Schuster. *On supervised learning from sequential data with applications for speech recognition.* PhD thesis, Nara Institute of Science and Technolog, Kyoto, Japan, 1999.

M. Schuster and K. K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45:2673–2681, November 1997.

A. Senior and A. J. Robinson. Forward-backward retraining of recurrent neural networks. In D. S. Touretzky, M. C. Mozer, and M. E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 743–749. The MIT Press, 1996. URL `citeseer.ist.psu.edu/senior96forwardbackward.html`.

F. Sha and L. K. Saul. Large margin hidden markov models for automatic speech recognition. In *NIPS*, pages 1249–1256, 2006.

J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.

P. Y. Simard, D. Steinkraus, and J. C. Platt. Best practices for convolutional neural networks applied to visual document analysis. In *ICDAR '03: Proceedings of the Seventh International Conference on Document Analysis and Recognition*, page 958, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1960-1.

T. Thireou and M. Reczko. Bidirectional long short-term memory networks for predicting the subcellular localization of eukaryotic proteins. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 4(3):441–446, 2007. ISSN 1545-5963.

E. Trentin and M. Gori. Robust combination of neural networks and hidden markov models for speech recognition. *Neural Networks, IEEE Transactions onn*, 14(6):1519–1531, 2003.

V. N. Vapnik. *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995. ISBN 0-387-94559-8.

Verbmobil. Database version 2.3, March 2004. http://www.phonetik.uni-muenchen.de/Forschung/Verbmobil/Verbmobil.html.

P. Werbos. Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10):1550 – 1560, 1990.

P. J. Werbos. Generalization of backpropagation with application to a recurrent gas market model. *Neural Networks*, 1, 1988.

D. Wierstra, F. J. Gomez, and J. Schmidhuber. Modeling systems with internal state using evolino. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1795–1802, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-010-8. doi: http://doi.acm.org/10.1145/1068009.1068315.

R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. In Y. Chauvin and D. E. Rumelhart, editors, *Back-propagation: Theory, Architectures and Applications*, pages 433–486. Lawrence Erlbaum Publishers, Hillsdale, N.J., 1995. URL `citeseer.nj.nec.com/williams95gradientbased.html`.

L. Wu and P. Baldi. A scalable machine learning approach to go. In B. Schlkopf, J. Platt, and T. Hoffman, editors, *NIPS*, pages 1521–1528. MIT Press, 2006. ISBN 0-262-19568-2. URL `http://dblp.uni-trier.de/db/conf/nips/nips2006.html#WuB06`.

S. Young and P. Woodland. *HTK Version 3.2: User, Reference and Programmer Manual*, 2002.

S. Young, N. Russell, and J. Thornton. Token passing: A simple conceptual model for connected speech recognition systems. Technical Report CUED/F-INFENG/TR38, Cambridge University Engineering Dept., Cambridge, UK, 1989.

S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey, V. Valtchev, and P. Woodland. *The HTK Book*. Cambridge University Engineering Department, HTK version 3.4 edition, December 2006.

D. Yu, L. Deng, and A. Acero. A lattice search technique for a long-contextual-span hidden trajectory model of speech. *Speech Communication*, 48(9):1214–1226, 2006. URL `http://dblp.uni-trier.de/db/journals/speech/speech48.html#YuDA06`.

G. Zavaliagkos, S. Austin, J. Makhoul, and R. M. Schwartz. A hybrid continuous speech recognition system using segmental neural nets with hidden markov models. *IJPRAI*, 7(4):949–963, 1993.

H. G. Zimmermann, R. Grothmann, A. M. Schaefer, and C. Tietz. Identification and forecasting of large dynamical systems by dynamical consistent neural networks. In S. Haykin, J. Principe, T. Sejnowski, and J. McWhirter, editors, *New Directions in Statistical Signal Processing: From Systems to Brain*, pages 203–242. MIT Press, 2006a.

M. Zimmermann, J.-C. Chappelier, and H. Bunke. Offline grammar-based recognition of handwritten sentences. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(5):818–821, 2006b.