Supervised Sequence Labelling with Recurrent Neural Networks

Alex Graves

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 discrete labels. Well-known examples include speech and handwriting recognition, protein secondary structure prediction and part-of-speech tagging. *Supervised* sequence labelling refers specifically to those cases where a set of hand-transcribed sequences 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. 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 mutually 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) are a class of artificial neural network architecture that—inspired by the cyclical connectivity of neurons in the brain—uses iterative function loops to store information. RNNs have several properties that make them an attractive choice for sequence labelling: they are flexible in their use of context information (because they can learn what to store and what to ignore); they accept many different types and representations of data; and they can recognise sequential patterns in the presence of sequential distortions. However they also have several drawbacks that have limited their application to real-world sequence labelling problems.

Perhaps the most serious flaw of standard RNNs is that it is very difficult to get them to store information for long periods of time (Hochreiter et al., 2001b). This limits the range of context they can access, which is of critical importance to sequence labelling. *Long Short-Term Memory* (LSTM; Hochreiter and Schmidhuber, 1997) is a redesign of the RNN architecture around special 'memory cell' units. In various synthetic tasks, LSTM has been shown capable of storing and accessing information over very long timespans (Gers et al., 2002; Gers and Schmidhuber, 2001). It has also proved advantageous in real-world domains such as speech processing (Graves and Schmidhuber, 2005b) and bioinformatics (Hochreiter et al., 2007). LSTM is therefore the architecture of choice throughout the book.

Another issue with the standard RNN architecture is that it can only access

contextual information in one direction (typically the past, if the sequence is temporal). This makes perfect sense for time-series prediction, but for sequence labelling it is usually advantageous to exploit the context on both sides of the labels. *Bidirectional RNNs* (Schuster and Paliwal, 1997) scan the data forwards and backwards with two separate recurrent layers, thereby removing the asymmetry between input directions and providing access to all surrounding context. *Bidirectional LSTM* (Graves and Schmidhuber, 2005b) combines the benefits of long-range memory and bidirectional processing.

For tasks such as speech recognition, where the alignment between the inputs and the labels is unknown, RNNs have so far been limited to an auxiliary role. The problem is that the standard training methods require a separate target for every input, which is usually not available. The traditional solution—the so-called *hybrid approach*—is to use hidden Markov models to generate targets for the RNN, then invert the RNN outputs to provide observation probabilities (Bourlard and Morgan, 1994). However the hybrid approach does not exploit the full potential of RNNs for sequence processing, and it also leads to an awkward combination of discriminative and generative training. The *connectionist temporal classification* (CTC) output layer (Graves et al., 2006) removes the need for hidden Markov models by directly training RNNs to label sequences with unknown alignments, using a single discriminative loss function. CTC can also be combined with probabilistic language models for word-level speech and handwriting recognition.

Recurrent neural networks were designed for one-dimensional sequences. However some of their properties, such as robustness to warping and flexible use of context, are also desirable in multidimensional domains like image and video processing. *Multidimensional RNNs*, a special case of *directed acyclic graph RNNs* (Baldi and Pollastri, 2003), generalise to multidimensional data by replacing the one-dimensional chain of network updates with an n-dimensional grid. *Multidimensional LSTM* (Graves et al., 2007) brings the improved memory of LSTM to multidimensional networks.

Even with the LSTM architecture, RNNs tend to struggle with very long data sequences. As well as placing increased demands on the network's memory, such sequences can be be prohibitively time-consuming to process. The problem is especially acute for multidimensional data such as images or videos, where the volume of input information can be enormous. *Hierarchical subsampling RNNs* (Graves and Schmidhuber, 2009) contain a stack of recurrent network layers with progressively lower spatiotemporal resolution. As long as the reduction in resolution is large enough, and the layers at the bottom of the hierarchy are small enough, this approach can be made computationally efficient for almost any size of sequence. Furthermore, because the effective distance between the inputs decreases as the information moves up the hierarchy, the network's memory requirements are reduced.

The combination of multidimensional LSTM, CTC output layers and hierarchical subsampling leads to a general-purpose sequence labelling system entirely constructed out of recurrent neural networks. The system is flexible, and can be applied with minimal adaptation to a wide range of data and tasks. It is also powerful, as this book will demonstrate with state-of-the-art results in speech and handwriting recognition.

1.1 Structure of the Book

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 7–9.

Chapter 2 briefly reviews supervised learning in general, and pattern classification in particular. It also provides a formal definition of sequence labelling, and discusses three classes of sequence labelling task that arise under different relationships between the input and label sequences. Chapter 3 provides background material for feedforward and recurrent neural networks, with emphasis on their application to labelling and classification tasks. It also introduces the *sequential Jacobian* as a tool for analysing the use of context by RNNs.

Chapter 4 describes the LSTM architecture and introduces bidirectional LSTM (BLSTM). Chapter 5 contains an experimental comparison of BLSTM to other neural network architectures applied to framewise phoneme classification. Chapter 6 investigates the use of LSTM in hidden Markov model-neural network hybrids. Chapter 7 introduces connectionist temporal classification, Chapter 8 covers multidimensional networks, and hierarchical subsampling networks are described 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 describes the different classes of sequence labelling task that arise under different assumptions about the label sequences.

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 scalar reward values are provided for training, and *unsupervised learning*, where no training signal exists at all, and the algorithm attempts to uncover the structure of the data by inspection alone. We will not consider either reinforcement learning or unsupervised learning in this book.

A 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 $\mathcal{Z}$, along with a disjoint *test set* S'. We will sometimes refer to the elements of S as *training examples*. Both S and S' are assumed to have been 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 during training; in particular validation sets are frequently used to determine when training should stop, in order to prevent overfitting. The goal is to use the training set to minimise some task-specific error measure E defined 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 test-set targets. For parametric algorithms (such as neural networks) the usual approach to error minimisation is to incrementally adjust the algorithm parameters to optimise a *loss function* on the training set, which is as closely related as possible to E. The transfer of learning from the training set to the test set is known as *generalisation*, and

will be discussed further in later chapters.

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.

2.2 Pattern Classification

Pattern classification, also known as pattern recognition, is one of the most extensively studied areas of machine learning (Bishop, 2006; Duda et al., 2000), and certain pattern classifiers, such as multilayer perceptrons (Rumelhart et al., 1986; Bishop, 1995) and support vector machines (Vapnik, 1995) have become familiar to the scientific community at large.

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.

The input space $\mathcal{X}$ for supervised pattern classification tasks is typically $\mathbf{R}^M$; that is, the set of all real-valued vectors of some fixed length M. The target spaces $\mathcal{Z}$ is a discrete set of K classes. A pattern classifier $h : \mathcal{X} \mapsto \mathcal{Z}$ is therefore a function mapping from vectors to labels. If all misclassifications are equally bad, the usual error measure for h is the *classification error rate* $E^{class}(h, S')$ on the test set S'

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

2.2.1 Probabilistic Classification

Classifiers that directly output class labels, of which support vector machines are a well known example, are sometimes referred to as *discriminant functions*. An alternative 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 $h(x)$:

$$h(x) = \arg\max_k p(C_k|x) \tag{2.2}$$

One advantage of the probabilistic approach is that the relative magnitude of the probabilities can be used to determine the degree of confidence the classifier has in its outputs. Another is that it allows the classifier to be combined with other probabilistic algorithms in a consistent way.

2.2.2 Training Probabilistic Classifiers

If a probabilistic classifier h_w yields a conditional distribution $p(C_k|x, w)$ over the class labels C_k given input x and parameters w, we can take a product

over the independent and identically distributed (i.i.d.) input-target pairs in the training set S to get

$$p(S|w) = \prod_{(x,z)\in S} p(z|x,w) \tag{2.3}$$

which can be inverted with Bayes' rule to obtain

$$p(w|S) = \frac{p(S|w)p(w)}{p(S)} \tag{2.4}$$

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

$$p(C_k|x,S) = \int_w p(C_k|x,w)p(w|S)dw \tag{2.5}$$

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

$$p(C_k|x,S) \approx p(C_k|x,w_{MAP}) \tag{2.6}$$

Since $p(S)$ is independent of w, Eqn. (2.4) tells us that

$$w_{MAP} = \arg\max_w p(S|w)p(w) \tag{2.7}$$

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 remove the $p(w)$ term from (2.7) to obtain the maximum likelihood (ML) parameter vector w_{ML}

$$w_{ML} = \arg\max_w p(S|w) = \arg\max_w \prod_{(x,z)\in S} p(z|x,w) \tag{2.8}$$

From now on we will drop the explicit dependence of the classifier outputs on w, with the understanding that $p(z|x)$ is the probability of x being correctly classified by h_w.

2.2.2.1 Maximum-Likelihood Loss Functions

The standard procedure for finding w_{ML} is to minimise a maximum-likelihood loss function $\mathcal{L}(S)$ defined as the negative logarithm of the probability assigned to S by the classifier

$$\mathcal{L}(S) = -\ln \prod_{(x,z)\in S} p(z|x) = -\sum_{(x,z)\in S} \ln p(z|x) \tag{2.9}$$

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

Observing that each example training example $(x, z) \in S$ contributes to a single term in the above sum, we define the *example loss* $\mathcal{L}(x, z)$ as

$$\mathcal{L}(x, z) = -\ln p(z|x) \tag{2.10}$$

and note that

$$\mathcal{L}(S) = \sum_{(x,z)\in S} \mathcal{L}(x, z) \tag{2.11}$$

$$\frac{\partial \mathcal{L}(S)}{\partial w} = \sum_{(x,z)\in S} \frac{\partial \mathcal{L}(x, z)}{\partial w} \tag{2.12}$$

It therefore suffices to derive $\mathcal{L}(x, z)$ and $\frac{\partial \mathcal{L}(x,z)}{\partial w}$ to completely define a maximum-likelihood loss function and its derivatives with respect to the network weights.

When the precise form of the loss function is not important, we will refer to it simply as $\mathcal{L}$.

2.2.3 Generative and Discriminative Methods

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

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

where

$$p(x) = \sum_k p(x|C_k)p(C_k) \tag{2.14}$$

Algorithms following this approach are referred to as *generative*, because the prior $p(x)$ 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 methods 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 book focuses on discriminative sequence labelling. However, we will frequently refer to the well-known generative method *hidden Markov models* (Rabiner, 1989; Bengio, 1999).

2.3 Sequence Labelling

The goal of sequence labelling is to assign sequences of labels, drawn from a fixed alphabet, to sequences of input data. For example, one might wish to transcribe

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

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 book 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.

If the sequences are assumed to be independent and identically distributed, we recover the basic framework of pattern classification, only with sequences in place of patterns (of course the data-points within each sequence are *not* assumed to be independent). In practice this assumption may not be entirely justified (for example, the sequences may represent turns in a spoken dialogue, or lines of text in a handwritten form); however it is usually not too damaging as long as the sequence boundaries are sensibly chosen. We further assume that each target sequence is at most as long as the corresponding input sequence. 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 $(\mathbf{x}, \mathbf{z})$ (From now on a bold typeface will be used to denote sequences). 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. $|\mathbf{z}| = U \leq |\mathbf{x}| = T$. Regardless of whether the data is a time series, the distinct points in the *input* sequence are referred to as *timesteps*. The task is 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, as accurately as possible.

In some cases we can apply additional constraints to the label sequences. These may affect both the choice of sequence labelling algorithm and the error measures used to assess performance. The following sections describe three classes of sequence labelling task, corresponding to progressively looser assump-

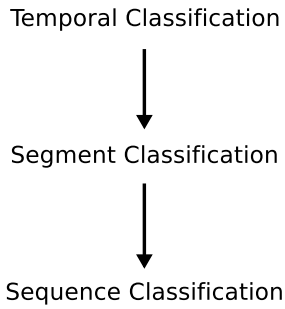

Figure 2.2: **Three classes of sequence labelling task.** Sequence classification, where each input sequence is assigned a single class, is a special case of segment classification, where each of a predefined set of input segments is given a label. Segment classification is a special case of temporal classification, where any alignment between input and label sequences is allowed. Temporal classification data can be *weakly labelled* with nothing but the target sequences, while segment classification data must be *strongly labelled* with both targets and input-target alignments.

tions about the relationship between the input and label sequences, and discuss algorithms and error measures suitable for each. The relationship between the classes is outlined in Figure 2.2.

2.3.1 Sequence Classification

The most restrictive case is where the label sequences are constrained to be length one. This is referred to as *sequence classification*, since each input sequence is assigned to a single class. Examples of sequence classification task include the identification of a single spoken work and the recognition of an individual handwritten letter. A key feature of such tasks 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 the application of multidimensional recurrent neural networks to MNIST in Chapter 8.

As with pattern classification the obvious error measure is the percentage of misclassifications, referred to as the *sequence error rate* E^{seq} in this context:

$$E^{seq}(h, S') = \frac{100}{|S'|} \sum_{(\mathbf{x},\mathbf{z}) \in S'} \begin{cases} 0 \text{ if } h(\mathbf{x}) = \mathbf{z} \\ 1 \text{ otherwise} \end{cases} \tag{2.15}$$

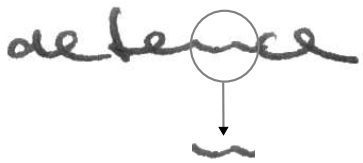

Figure 2.3: **Importance of context in segment classification.** The word 'defence' is clearly legible. However the letter 'n' in isolation is ambiguous.

where $|S'|$ is the number of elements in S'.

2.3.2 Segment Classification

Segment classification refers to those tasks where the target sequences consist of multiple labels, but the locations of the labels—that is, the positions of the input segments to which the labels apply—are known in advance. Segment classification is common in domains such as natural language processing and bioinformatics, where the inputs are discrete and can be trivially segmented. It can also occur in domains where segmentation is difficult, such as audio or image processing; however this typically requires hand-segmented training data, which is difficult to obtain.

A crucial element of segment classification, missing from sequence classification, is the use of *context* information from 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 segments into *time-windows*, and use the windows as input patterns. 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 from segment to segment. Consequently the case for sequential algorithms is stronger here than in sequence classification.

The obvious error measure for segment classification is the *segment error rate* E^{seg}, which simply counts the percentage of misclassified segments.

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

Where

$$Z = \sum_{(\mathbf{x},\mathbf{z}) \in S'} |z| \tag{2.17}$$

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 phonetic 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*.

Various neural network architectures are applied 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*. Multidimensional recurrent neural networks are applied to image segmentation in Chapter 8.

2.3.3 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 segment boundaries are unknown. Instead we measure the total number of substitutions, insertions and deletions that would be required to turn one sequence into the other, giving us the *label error rate* E^{lab}:

$$E^{lab}(h, S') = \frac{1}{Z} \sum_{(\mathbf{x},\mathbf{z}) \in S'} ED(h(\mathbf{x}), \mathbf{z}) \tag{2.18}$$

Where $ED(\mathbf{p}, \mathbf{q})$ is the *edit 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 $\mathbf{q}$). $ED(\mathbf{p}, \mathbf{q})$ can be calculated in $O(|\mathbf{p}||\mathbf{q}|)$ time (Navarro, 2001). The label error rate is typically multiplied by 100 so that it can be interpreted as a percentage (a convention we will follow in this book); however, unlike the other error measures considered in this chapter, it is not a true percentage, and may give values higher than 100.

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 book however, the label error rate is sufficient. We will usually refer to the label error rate according to the type of label in question, for example *phoneme error rate* or *word error rate*. For some temporal classification tasks a completely correct labelling is required and the degree of error is unimportant. In this case the sequence error rate (2.15) should be used to assess performance.

The use of hidden Markov model-recurrent neural network hybrids for temporal classification is investigated in Chapter 6, and a neural-network-only approach to temporal classification is introduced in Chapter 7.

Chapter 3

Neural Networks

This chapter provides an overview of artificial neural networks, with 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. 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, 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, 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).

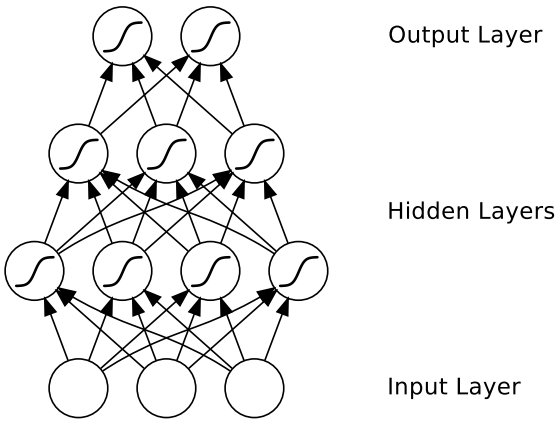

Figure 3.1: **A multilayer perceptron.** The S-shaped curves in the hidden and output layers indicate the application of 'sigmoidal' nonlinear activation functions

As illustrated in Figure 3.1, the units in a multilayer perceptron are arranged in layers, with connections feeding forward from one layer to the next. Input patterns are presented to the *input layer*, then propagated through the *hidden layers* to the *output layer*. This process is known as the *forward pass* of the network.

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 with a particular set of weight values defines a function from input to output vectors. By altering the 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 approximators*.

3.1.1 Forward Pass

Consider an MLP with I input units, activated by input vector x (hence $|x| = I$). 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

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

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

Several neural network activation functions are plotted in Figure 3.2. The most common choices are the hyperbolic tangent

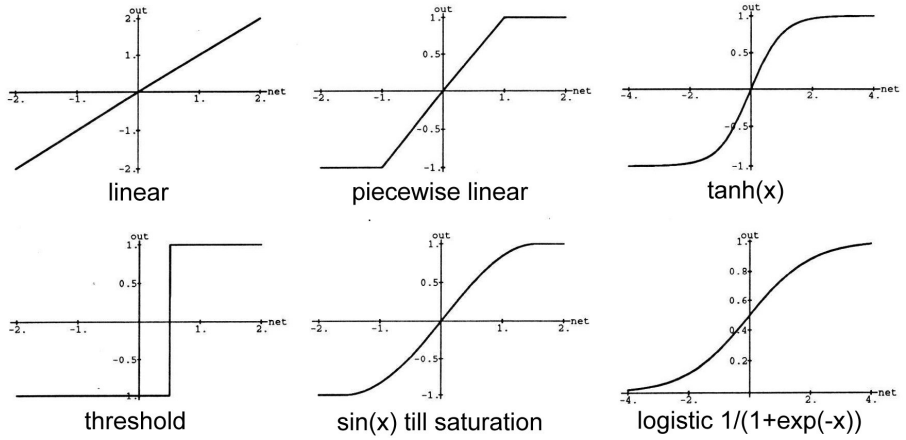

Figure 3.2: **Neural network activation functions.** Note the characteristic 'sigmoid' or S-shape.

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

and the logistic sigmoid

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

The two functions are related by the following linear transform:

$$tanh(x) = 2\sigma(2x) - 1 \tag{3.5}$$

This means that any function computed by a neural network with a hidden layer of tanh units can be computed by another network with logistic sigmoid units and vice-versa. They are therefore largely equivalent as activation functions. However one reason to distinguish between them is that their output ranges are different; in particular if an output between 0 and 1 is required (for example, if the output represents a probability) then the logistic sigmoid should be used.

An important feature of both tanh and the logistic sigmoid is their nonlinearity. Nonlinear neural networks are 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

$$\frac{\partial tanh(x)}{\partial x} = 1 - tanh(x)^2 \tag{3.6}$$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)) \tag{3.7}$$

Because of the way they reduce an infinite input domain to 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 the l^{th} hidden layer H_l

$$a_h = \sum_{h' \in H_{l-1}} w_{h'h} b_{h'} \tag{3.8}$$

$$b_h = \theta_h(a_h) \tag{3.9}$$

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. Therefore

$$a_k = \sum_{h \in H_L} w_{hk} b_h \tag{3.10}$$

for a network with L hidden layers.

Both the number of units in the output layer and the choice of output activation function depend on the task the network is applied to. 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 the probability that the input vector belongs to the first class (and conversely, one minus the activation gives the probability that it belongs to the second class)

$$\begin{aligned} p(C_1|x) &= y = \sigma(a) \\ p(C_2|x) &= 1 - y \end{aligned} \tag{3.11}$$

The use of the logistic sigmoid as a binary probability estimator is sometimes referred 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

$$p(z|x) = y^z (1 - y)^{1-z} \tag{3.12}$$

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 the class probabilities:

$$p(C_k|x) = y_k = \frac{e^{a_k}}{\sum_{k'=1}^{K} e^{a_{k'}}} \tag{3.13}$$

which is also known as a *multinomial logit* model. 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:

$$p(z|x) = \prod_{k=1}^{K} y_k^{z_k} \tag{3.14}$$

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 Loss Functions

The derivation of loss 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. (2.5) for neural networks (MacKay, 1995; Neal, 1996), we will here focus on loss functions derived using maximum likelihood. For binary classification, substituting (3.12) into the maximum-likelihood example loss $\mathcal{L}(x, z) = -\ln p(z|x)$ described in Section 2.2.2.1, we have

$$\mathcal{L}(x, z) = (z - 1)\ln(1 - y) - z \ln y \tag{3.15}$$

Similarly, for problems with multiple classes, substituting (3.14) into (2.10) gives

$$\mathcal{L}(x, z) = -\sum_{k=1}^{K} z_k \ln y_k \tag{3.16}$$

See (Bishop, 1995, chap. 6) for more information on these and other MLP loss functions.

3.1.4 Backward Pass

Since MLPs are, by construction, differentiable operators, they can be trained to minimise any differentiable loss function using *gradient descent*. The basic idea of gradient descent is to find the derivative of the loss 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.

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 loss function with respect to the output units. For a binary classification network, differentiating the loss function defined in (3.15) with respect to the network outputs gives

$$\frac{\partial \mathcal{L}(x, z)}{\partial y} = \frac{y - z}{y(1 - y)} \tag{3.17}$$

The chain rule informs us that

$$\frac{\partial \mathcal{L}(x, z)}{\partial a} = \frac{\partial \mathcal{L}(x, z)}{\partial y} \frac{\partial y}{\partial a} \tag{3.18}$$

and we can then substitute (3.7), (3.11) and (3.17) into (3.18) to get

$$\frac{\partial \mathcal{L}(x,z)}{\partial a} = y - z \tag{3.19}$$

For a multiclass network, differentiating (3.16) gives

$$\frac{\partial \mathcal{L}(x,z)}{\partial y_k} = -\frac{z_k}{y_k} \tag{3.20}$$

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

$$\frac{\partial \mathcal{L}(x,z)}{\partial a_k} = \sum_{k'=1}^{K} \frac{\partial \mathcal{L}(x,z)}{\partial y_{k'}} \frac{\partial y_{k'}}{\partial a_k} \tag{3.21}$$

Differentiating (3.13) we obtain

$$\frac{\partial y_{k'}}{\partial a_k} = y_k \delta_{kk'} - y_k y_{k'} \tag{3.22}$$

and we can then substitute (3.22) and (3.20) into (3.21) to get

$$\frac{\partial \mathcal{L}(x,z)}{\partial a_k} = y_k - z_k \tag{3.23}$$

where we have the used the fact that $\sum_{k=1}^{K} z_k = 1$. Note the similarity to (3.19). The loss function is sometimes said to *match* the output layer activation function when the output derivative has this form (Schraudolph, 2002).

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:

$$\delta_j \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}(x,z)}{\partial a_j} \tag{3.24}$$

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

$$\delta_h = \frac{\partial \mathcal{L}(x,z)}{\partial b_h} \frac{\partial b_h}{\partial a_h} = \frac{\partial b_h}{\partial a_h} \sum_{k=1}^{K} \frac{\partial \mathcal{L}(x,z)}{\partial a_k} \frac{\partial a_k}{\partial b_h} \tag{3.25}$$

where we have used the fact that $\mathcal{L}(x,z)$ depends only on each hidden unit h through its influence on the output units. Differentiating (3.10) and (3.2) and substituting into (3.25) gives

$$\delta_h = \theta'(a_j) \sum_{k=1}^{K} \delta_k w_{hk} \tag{3.26}$$

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

$$\delta_h = \theta'(a_h) \sum_{h' \in H_{l+1}} \delta_{h'} w_{hh'} \tag{3.27}$$

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:

$$\frac{\partial \mathcal{L}(x,z)}{\partial w_{ij}} = \frac{\partial \mathcal{L}(x,z)}{\partial a_j} \frac{\partial a_j}{\partial w_{ij}} = \delta_j b_i \tag{3.28}$$

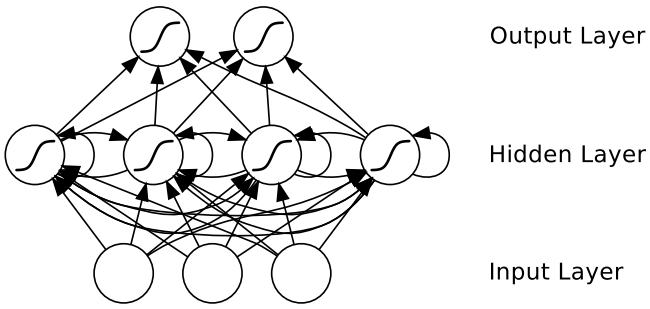

Figure 3.3: **A recurrent neural network.**

3.1.4.1 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 loss function:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\mathcal{L}(w_{ij} + \epsilon) - \mathcal{L}(w_{ij} - \epsilon)}{2\epsilon} + \mathcal{O}(\epsilon^2) \tag{3.29}$$

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 floating point accuracy of a given implementation. For the systems we used, $\epsilon = 10^{-5}$ generally gave best results.

Note that for a network with W weights, calculating the full gradient using (3.29) requires $O(W^2)$ time, whereas backpropagation only requires $O(W)$ time. Numerical differentiation is therefore impractical for network training. Furthermore, it is recommended to always the choose the smallest possible exemplar of the network architecture whose gradient you wish to check (for example, an RNN with a single hidden unit).

3.2 Recurrent Neural Networks

In the previous section we considered feedforward neural networks 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 feedforward networks, 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 chapter, we focus on a simple RNN containing a single, self connected hidden layer, as shown in Figure 3.3.

While the difference between a multilayer perceptron and an RNN 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, and thereby influence the network output.

3.2.1 Forward Pass

The forward pass of an RNN is the same as that of a multilayer perceptron with a single hidden layer, except that activations arrive at the hidden layer from both the current external input and the hidden layer activations from the previous timestep. 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

$$a_h^t = \sum_{i=1}^{I} w_{ih} x_i^t + \sum_{h'=1}^{H} w_{h'h} b_{h'}^{t-1} \tag{3.30}$$

Nonlinear, differentiable activation functions are then applied exactly as for an MLP

$$b_h^t = \theta_h(a_h^t) \tag{3.31}$$

The complete sequence of hidden activations can be calculated by starting at $t = 1$ and recursively applying (3.30) and (3.31), 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 book, the initial values are always set to zero. However, other researchers have found that RNN stability and performance 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:

$$a_k^t = \sum_{h=1}^{H} w_{hk} b_h^t \tag{3.32}$$

For sequence classification and segment classification tasks (Section 2.3) the MLP output activation functions described in Section 3.1.2 (that is, logistic sigmoid for two classes and softmax for multiple classes) can be reused for RNNs, with the classification targets typically presented at the ends of the sequences or segments. It follows that the loss functions in Section 3.1.3 can be reused too. Temporal classification is more challenging, since the locations of the target classes are unknown. Chapter 7 introduces an output layer specifically designed for temporal classification with RNNs.

3.2.2 Backward Pass

Given the partial derivatives of some differentiable loss function $\mathcal{L}$ with respect to the network outputs, the next step is to determine 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).

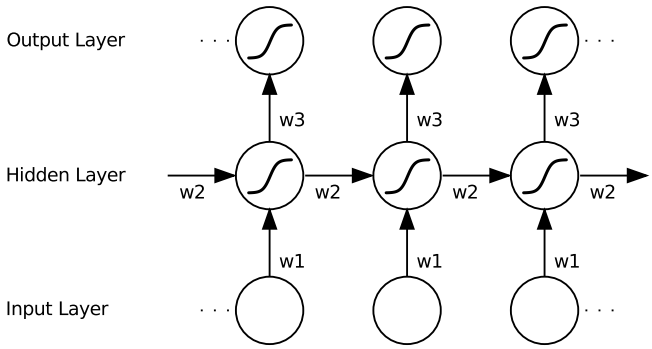

Figure 3.4: **An unfolded recurrent network.** Each node represents a layer of network units at a single timestep. The weighted connections from the input layer to hidden layer are labelled 'w1', those from the hidden layer to itself (i.e. the recurrent weights) are labelled 'w2' and the hidden to output weights are labelled 'w3'. Note that the same weights are reused at every timestep. Bias weights are omitted for clarity.

Like standard backpropagation, BPTT consists of a repeated application of the chain rule. The subtlety is that, for recurrent networks, the loss 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. Therefore

$$\delta_h^t = \theta'(a_h^t)\left(\sum_{k=1}^{K}\delta_k^t w_{hk} + \sum_{h'=1}^{H}\delta_{h'}^{t+1} w_{hh'}\right) \tag{3.33}$$

where

$$\delta_j^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_j^t} \tag{3.34}$$

The complete sequence of δ terms can be calculated by starting at $t = T$ and recursively applying (3.33), 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 same weights are reused at every timestep, we sum over the whole sequence to get the derivatives with respect to the network weights: d

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{t=1}^{T}\frac{\partial \mathcal{L}}{\partial a_j^t}\frac{\partial a_j^t}{\partial w_{ij}} = \sum_{t=1}^{T}\delta_j^t b_i^t \tag{3.35}$$

3.2.3 Unfolding

A useful way to visualise RNNs is to consider the update graph formed by 'unfolding' the network along the input sequence. Figure 3.4 shows part of an unfolded RNN. Note that the unfolded graph (unlike Figure 3.3) contains no cycles; otherwise the forward and backward pass would not be well defined.

Viewing RNNs as unfolded graphs makes it easier to generalise to networks with more complex update dependencies. We will encounter such a network

in the next section, and again when we consider multidimensional networks in Chapter 8 and hierarchical networks in Chapter 9.

3.2.4 Bidirectional Networks

For many sequence labelling tasks it is beneficial 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 ignore future context. An obvious solution is to add a time-window of future context to the network input. However, as well as increasing the number of input weights, this approach suffers from the same problems as the time-window methods discussed in Sections 2.3.1 and 2.3.2: 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. This structure provides the output layer with complete 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), and we find that they consistently outperform unidirectional RNNs at sequence labelling.

An unfolded bidirectional network is shown in Figure 3.5.

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**
  Forward pass for the forward hidden layer, storing activations at each timestep
for $t = T$ to 1 **do**
  Forward pass for the backward hidden layer, storing activations at each timestep
for all t, in any order **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

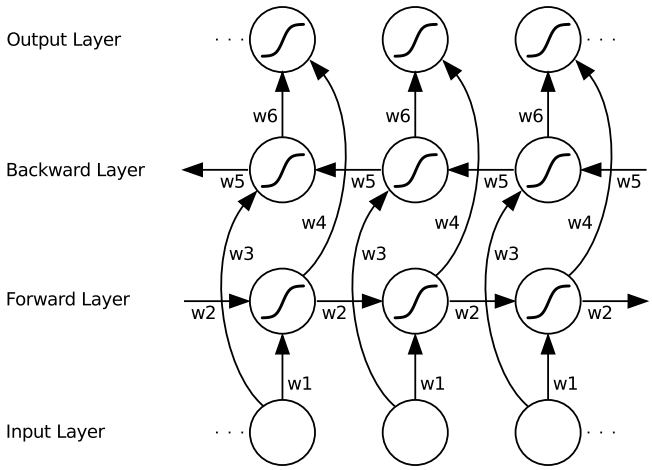

Figure 3.5: **An unfolded bidirectional network.** Six distinct sets of weights are reused at every timestep, corresponding to the input-to-hidden, hidden-to-hidden and hidden-to-output connections of the two hidden layers. Note that no information flows between the forward and backward hidden layers; this ensures that the unfolded graph is acyclic.

BPTT, except that all the output layer δ terms are calculated first, then fed back to the two hidden layers in opposite directions:

for all t, in any order **do**
  Backward pass for the output layer, storing δ terms at each timestep
for $t = T$ to 1 **do**
  BPTT backward pass for the forward hidden layer, using the stored δ terms from the output layer
for $t = 1$ to T **do**
  BPTT backward pass for the backward hidden layer, using the stored δ terms from the output layer

Algorithm 3.2: BRNN Backward Pass

3.2.4.1 Causal Tasks

One objection to bidirectional networks 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, if the input sequences are spatial and not temporal there is no reason to distinguish between past and future inputs. This is perhaps why protein structure prediction is the domain where BRNNs have been most widely adopted (Baldi et al., 2001; Thireou and Reczko, 2007). 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, such as a pause in speech, before processing a section of the data.

3.2.5 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 step towards this by measuring the sensitivity of the network outputs to the network inputs.

For feedforward neural networks, the *Jacobian* J is the matrix of partial derivatives of the network output vector y with respect to the input vector x:

$$J_{ki} = \frac{\partial y_k}{\partial x_i} \tag{3.36}$$

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

$$J_{ki}^{tt'} = \frac{\partial y_k^t}{\partial x_i^{t'}} \tag{3.37}$$

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

Figure 3.6 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 timestep (for example, the point when the network outputs a label) to the inputs at all timesteps 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 relative degree to which the output is influenced by each input.

Slices like that shown in Figure 3.6 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 intermediate delta terms (such as those in the hidden layer) are also potentially interesting, since they reveal the responsiveness of the output to different parts of the network over time.

The sequential Jacobian will be used throughout the book as a means of analysing the use of context by RNNs. However it should be stressed 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, or the first timestep in a set of audio sequences that always begin in silence, since the network does not 'expect' to see any change there. However, this input will

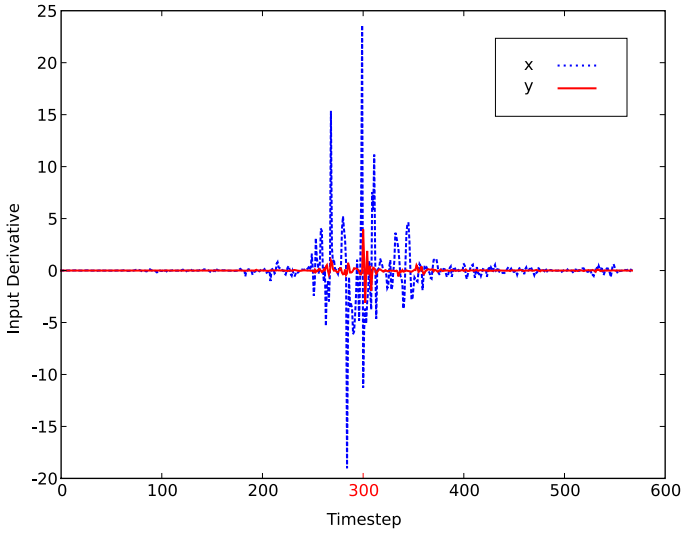

Figure 3.6: **Sequential Jacobian for a bidirectional network during an online handwriting recognition task.** The derivatives of a single output unit at time $t = 300$ are evaluated with respect to the two inputs (corresponding to the x and y coordinates of the pen) at all times throughout the sequence. 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 coordinate (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); this does *not* imply that the network makes more use of the x coordinates than the y coordinates.

not provide any useful context information. Also, as shown in Figure 3.6, 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 neural networks can be differentiated with respect to loss functions, and thereby trained with gradient descent. However, to ensure that network training is both effective and tolerably fast, and that it generalises well to unseen data, several issues must be addressed.

3.3.1 Gradient Descent Algorithms

Most obviously, we need to decide how to follow the error gradient. The simplest method, 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 of the loss function:

$$\Delta w^n = -\alpha \frac{\partial \mathcal{L}}{\partial w^n} \tag{3.38}$$

where Δw^n is the n^{th} weight update, $\alpha \in [0, 1]$ is the *learning rate* and w^n is the weight vector before Δw^n is applied. This process is repeated until some *stopping criteria* (such as failure to reduce the loss for a given number of steps) is met.

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), which effectively adds inertia to the motion of the algorithm through weight space, thereby speeding up convergence and helping to escape from local minima:

$$\Delta w^n = m\Delta w^{n-1} - \alpha \frac{\partial \mathcal{L}}{\partial w^n} \tag{3.39}$$

where $m \in [0, 1]$ is the momentum parameter.

When the above gradients are calculated with respect to a loss function defined over the entire training set, the weight update procedure is referred to as *batch learning*. This is in contrast to *online* or *sequential* learning, where weight updates are performed using the gradient with respect to individual training examples. Pseudocode for online learning with gradient descent is provided in Algorithm 3.3.

```
while stopping criteria not met do
  Randomise training set order
  for each example in the training set do
    Run forward and backward pass to calculate the gradient
    Update weights with gradient descent algorithm
```

Algorithm 3.3: Online Learning with Gradient Descent

A large number of sophisticated gradient descent algorithms have been developed, such as RPROP (Riedmiller and Braun, 1993), quickprop (Fahlman,

1989), conjugate gradients (Hestenes and Stiefel, 1952; Shewchuk, 1994) and L-BFGS (Byrd et al., 1995), that generally outperform steepest descent at batch learning. However steepest descent is much better suited than they are to online learning, because it takes very small steps at each weight update and can therefore tolerate constantly changing gradients.

Online learning tends to be more efficient than batch learning when large datasets containing significant redundancy or regularity are used (LeCun et al., 1998c). In addition, the stochasticity of online learning can help to escape from local minima (LeCun et al., 1998c), since the loss function is 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*). Training set randomisation is used for all the experiments in this book.

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 book were carried out using online steepest descent with momentum.

3.3.2 Generalisation

Although the loss functions for network training are, of necessity, defined on the training set, the real goal is to optimise performance on a test set of previously unseen 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). In general the larger the training set the better the generalisation. Many methods for improved generalisation with a fixed size training set (often referred to as *regularisers*) have been proposed over the years. In this book, however, only three simple regularisers are used: early stopping, input noise and weight noise.

3.3.2.1 Early Stopping

For early stopping, part of the training set is removed for use as a *validation set*. All stopping criteria are then tested on the validation set instead of the training set. The 'best' weight values are also chosen using the validation set, typically by picking the weights that minimise on the validation set the error function used to assess performance on the test set. In practice the two are usually done in tandem, with the error evaluated at regular intervals on the validation set, and training stopped after the error fails to decrease for a certain number of evaluations.

The test set should not be used to decide when to stop training or to choose the best weight values; these are indirect forms of training on the test set. In principle, the network should not be evaluated on the test set at all until training is complete.

During training, the error typically decreases at first on all sets, but after a certain point it begins to rise on the test and validation sets, while continuing to decrease on the training set. This behaviour, known as *overfitting*, is illustrated in Figure 3.7.

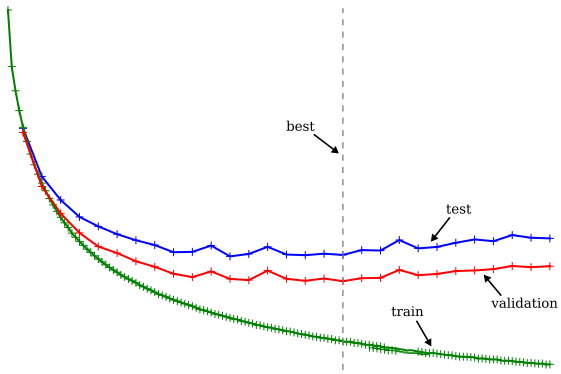

Figure 3.7: **Overfitting on training data.** Initially, network error decreases rapidly on all datasets. Soon however it begins to level off and gradually rise on the validation and test sets. The dashed line indicates the point of best performance on the validation set, which is close, but not identical to the optimal point for the test set.

Early stopping is perhaps the simplest and most universally applicable method for improved generalisation. However one drawback 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 book, 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 performance; it is only important that overfitting begins at approximately the same time on both of them.

3.3.2.2 Input Noise

Adding zero-mean, fixed-variance Gaussian noise to the network inputs during training (sometimes referred to as *training with jitter*) is a well-established method for improved generalisation (An, 1996; Koistinen and Holmström, 1991; Bishop, 1995). The desired effect is to artificially enhance the size of the training set, and thereby improve generalisation, by generating new inputs with the same targets as the original ones.

One problem with input noise is that it is difficult to determine in advance how large the noise variance should be. Although various rules of thumb exist, the most reliable method is to set the variance empirically on the validation set.

A more fundamental difficulty is that input perturbations are only effective if they reflect the variations found in the real data. For example, adding Gaussian noise to individual pixel values in an image will not generate a substantially different image (only a 'speckled' version of the original) and is therefore unlikely to aid generalisation to new images. Independently perturbing the points in a smooth trajectory is ineffectual for the same reason. Input perturbations tailored towards a particular dataset have been shown to be highly effective at improving generalisation (Simard et al., 2003); however this requires a prior model of the data variations, which is not usually available.

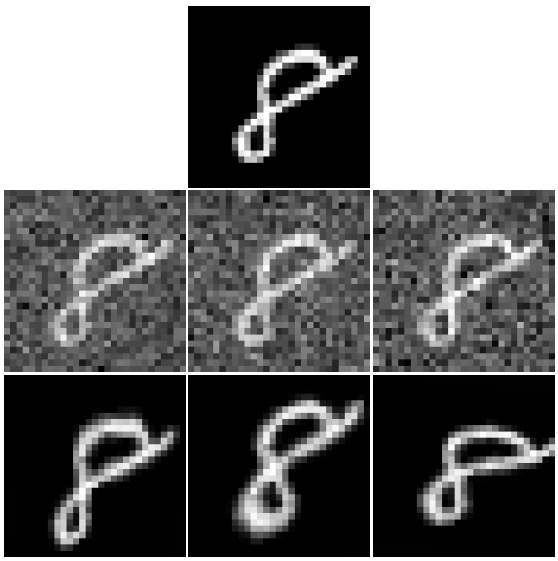

Figure 3.8: **Different Kinds of Input Perturbation.** A handwritten digit from the MNIST database (top) is shown perturbed with Gaussian noise (centre) and elastic deformations (bottom). Since Gaussian noise does not alter the outline of the digit and the noisy images all look qualitatively the same, this approach is unlikely to improve generalisation on MNIST. The elastic distortions, on the other hand, appear to create different handwriting samples out of the same image, and can therefore be used to artificially extend the training set.

Figure 3.8 illustrates the distinction between Gaussian input noise and data-specific input perturbations.

Input noise should be regenerated for every example presented to the network during training; in particular, the same noise should not be re-used for a given example as the network cycles through the data. Input noise should not be added during testing, as doing so will hamper performance.

3.3.2.3 Weight Noise

An alternative regularisation strategy is to add zero-mean, fixed variance Gaussian noise to the network *weights* (Murray and Edwards, 1994; Jim et al., 1996). Because *weight noise* or *synaptic noise* acts on the network's internal representation of the inputs, rather than the inputs themselves, it can be used for any data type. However weight noise is typically less effective than carefully designed input perturbations, and can lead to very slow convergence.

Weight noise can be used to 'simplify' neural networks, in the sense of reducing the amount of information required to transmit the network (Hinton and van Camp, 1993). Intuitively this is because noise reduces the precision with which the weights must be described. Simpler networks are preferable because they tend to generalise better—a manifestation of Occam's razor.

Algorithm 3.4 shows how weight noise should be applied during online learning with gradient descent.

while stopping criteria not met **do**
  Randomise training set order
  for each example in the training set **do**
    Add zero mean Gaussian noise to weights
    Run forward and backward pass to calculate the gradient
    Restore original weights
    Update weights with gradient descent algorithm

Algorithm 3.4: Online Learning with Gradient Descent and Weight Noise

As with input noise, weight noise should not be added when the network is evaluated on test data.

3.3.3 Input Representation

Choosing a suitable representation of the input data is a vital part of any machine learning task. Indeed, in some cases it is more important to the final performance than the algorithm itself. Neural networks, however, tend to be relatively robust to the choice of input representation: for example, in previous work on phoneme recognition, RNNs were shown to perform almost equally well using a wide range of speech preprocessing methods (Robinson et al., 1990). We report similar findings in Chapters 7 and 9, with very different input representations found to give roughly equal performance for both speech and handwriting recognition.

The only requirements for neural network input representations are that they are complete (in the sense of containing all 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 leads to an excessive number of input weights and poor generalisation. Beyond that the choice of input representation is something of a black art, whose aim is to make the relationship between the inputs and the targets as simple as possible.

One procedure that should be carried out for all neural network input data 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

$$m_i = \frac{1}{|S|} \sum_{x \in S} x_i \tag{3.40}$$

and standard deviation

$$\sigma_i = \sqrt{\frac{1}{|S|} \sum_{x \in S} (x_i - m_i)^2} \tag{3.41}$$

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

$$\hat{x}_i = \frac{x_i - m_i}{\sigma_i} \tag{3.42}$$

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., 1998c). Note that the test and validation sets should be standardised with the mean and standard deviation of the training set.

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

3.3.4 Weight Initialisation

Many gradient descent algorithms for neural networks require small, random, initial values for the weights. For the experiments in this book, 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 each experiment must be repeated several times to determine significance.

Chapter 4

Long Short-Term Memory

As discussed in the previous chapter, an important benefit of recurrent neural 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 in practice accessed is quite 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. This effect is often referred to in the literature as the *vanishing gradient problem* (Hochreiter, 1991; Hochreiter et al., 2001a; Bengio et al., 1994). 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). The approach favoured by this book is the *Long Short-Term Memory* (LSTM) architecture (Hochreiter and Schmidhuber, 1997).

This chapter reviews the background material for LSTM. Section 4.1 describes the basic structure of LSTM and explains how it tackles 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 4.6 provides all the equations required to train and apply LSTM networks.

4.1 Network 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.

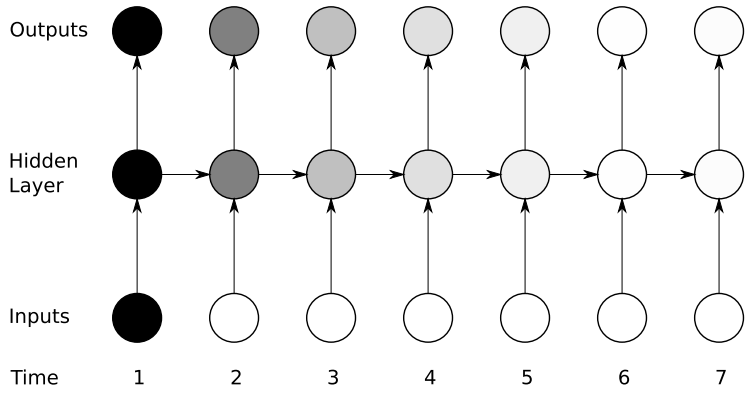

Figure 4.1: **The vanishing gradient problem for RNNs.** The shading of the nodes in the unfolded network indicates their sensitivity to the inputs at time one (the darker the shade, the greater the sensitivity). The sensitivity decays over time as new inputs overwrite the activations of the hidden layer, and the network 'forgets' the first inputs.

Figure 4.2 provides an illustration of an LSTM memory block with a single cell. An LSTM network is the same as a standard RNN, except that the summation units in the hidden layer are replaced by memory blocks, as illustrated in Fig. 4.3. LSTM blocks can also be mixed with ordinary summation units, although this is typically not necessary. The same output layers can be used for LSTM networks as for standard RNNs.

The multiplicative gates allow LSTM memory cells to store and access information over long periods of time, thereby mitigating the vanishing gradient problem. For example, as long as the input gate remains closed (i.e. has an activation near 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 4.4.

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 numbers 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), speech recognition (Graves and Schmidhuber, 2005b; Graves et al., 2006) and handwriting recognition (Liwicki et al., 2007; Graves et al., 2008). As would be expected, its advantages are most pronounced for problems requiring the use of long range contextual information.

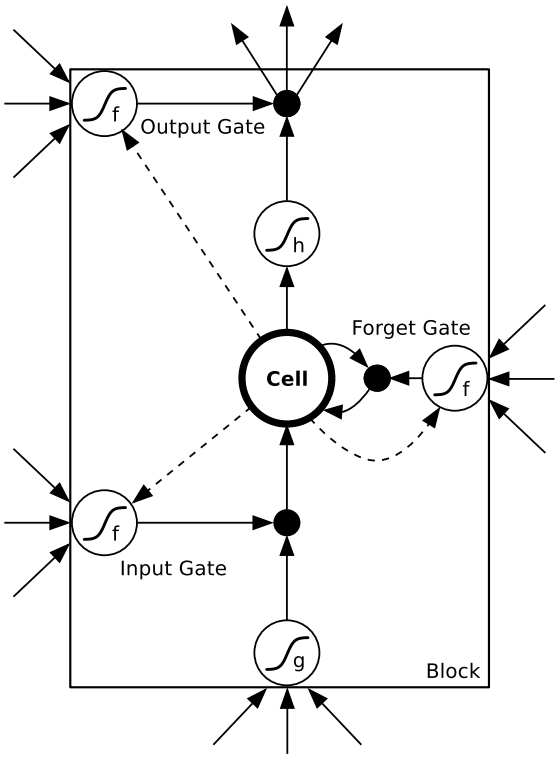

Figure 4.2: **LSTM memory block with one cell.** The three gates are nonlinear summation units that collect activations from inside and outside the block, and control the activation of the cell via multiplications (small black circles). The input and output gates multiply the input and output of the cell while the forget gate multiplies the cell's previous state. No activation function is applied within the cell. The gate activation function 'f' is usually the logistic sigmoid, so that the gate activations are between 0 (gate closed) and 1 (gate open). The cell input and output activation functions ('g' and 'h') are usually tanh or logistic sigmoid, though in some cases 'h' is the identity function. The weighted 'peephole' connections from the cell to the gates are shown with dashed lines. All other connections within the block are unweighted (or equivalently, have a fixed weight of 1.0). The only outputs from the block to the rest of the network emanate from the output gate multiplication.

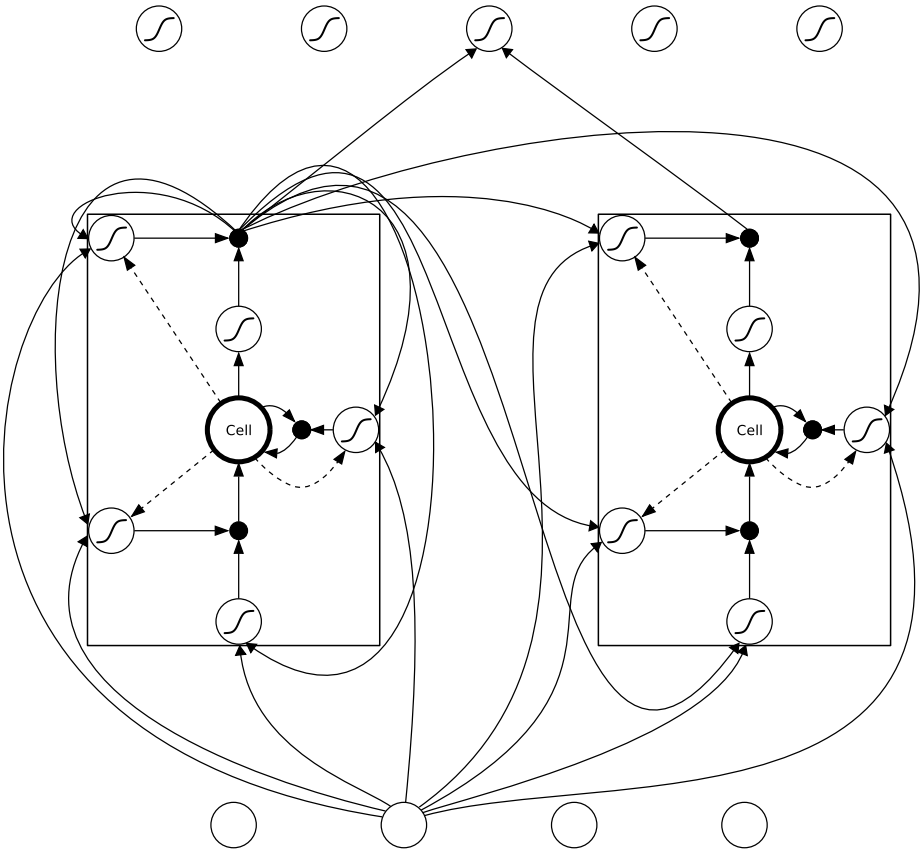

Figure 4.3: **An LSTM network.** The network consists of four input units, a hidden layer of two single-cell LSTM memory blocks and five output units. Not all connections are shown. Note that each block has four inputs but only one output.

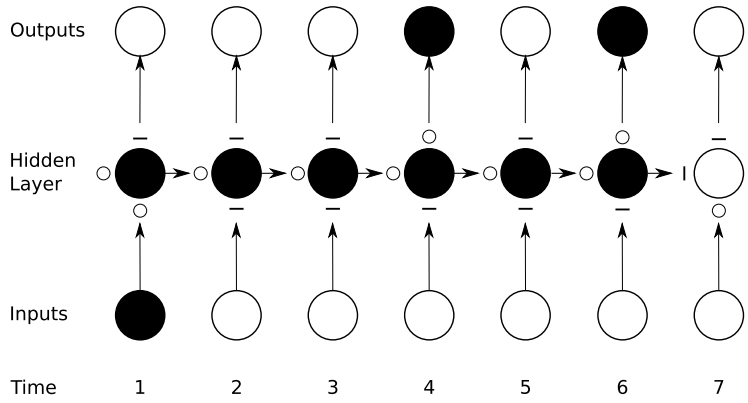

Figure 4.4: **Preservation of gradient information by LSTM.** As in Figure 4.1 the shading of the nodes indicates their sensitivity to the inputs at time one; in this case the black nodes are maximally sensitive and the white nodes are entirely insensitive. The state of the input, forget, and output gates are displayed below, to the left and above the hidden layer respectively. For simplicity, all 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. 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

The 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 has a sampling rate of over 40 kHz. Clearly, a great many timesteps would have to be spanned by a sequence learning algorithm attempting to label or model an utterance presented in this form. However when the signal is first transformed into a 100 Hz series of mel-frequency cepstral coefficients, it becomes feasible to model the data using an algorithm whose contextual range is relatively short, such as a hidden Markov model.

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 handling long time dependencies are essential.

4.3 Gradient Calculation

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

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.

However, it is also possible to calculate the exact LSTM gradient with untruncated 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 3.1.4.1. Only the exact gradient is used in this book, and the equations for it are provided in Section 4.6.

4.4 Architectural Variants

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 to give *extended LSTM* (Gers, 2001). 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.

Since LSTM is entirely composed of simple multiplication and summation units, and connections between them, it is straightforward to create further variants of the block architecture. Indeed it has been shown that alternative structures with equally good performance on toy problems such as learning context-free and context-sensitive languages can be evolved automatically (Bayer et al., 2009). However the standard extended form appears to be a good general purpose structure for sequence labelling, and is used exclusively in this book.

4.5 Bidirectional Long Short-Term Memory

Using LSTM as the network architecture in a bidirectional recurrent neural network (Section 3.2.4) yields bidirectional LSTM (Graves and Schmidhuber, 2005a,b; Chen and Chaudhari, 2005; Thireou and Reczko, 2007). Bidirectional LSTM provides access to long range context in both input directions, and will be used extensively in later chapters.

4.6 Network Equations

This section provides the equations for the activation (forward pass) and BPTT gradient calculation (backward pass) of an LSTM hidden layer within a recurrent neural network.

As before, w_{ij} is the weight of the connection from unit i to unit j, the network input to unit j at time t is denoted a_j^t and activation of unit j at time t 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 respectively to the input gate, forget gate and output gate of the block. The subscripts c refers to one of the C memory cells. The *peephole weights* from cell c to the input, forget and output gates are denoted $w_{c\iota}$, $w_{c\phi}$ and $w_{c\omega}$ respectively. 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. Unlike standard RNNs, an LSTM layer contains more inputs than outputs (because both the gates and the cells receive input from the rest of the network, but only the cells produce output visible to the rest of the network). We therefore define G as the total number of inputs to the hidden layer, including cells and gates, and use the index g to refer to these inputs when we don't wish to distinguish between the input types. For a standard LSTM layer with one cell per block G is equal to $4H$.

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 to one (see Section 3.2 for details). The final weight derivatives are found by summing over the derivatives at each timestep, as expressed in Eqn. (3.35). Recall that

$$\delta_j^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial a_j^t} \tag{4.1}$$

where $\mathcal{L}$ is the loss 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 initialised to zero at $t = 0$, and all δ terms are zero at $t = T + 1$.

4.6.1 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

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

4.6.2 Backward Pass

$$\epsilon_c^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial b_c^t} \qquad \epsilon_s^t \stackrel{\text{def}}{=} \frac{\partial \mathcal{L}}{\partial s_c^t}$$

Cell Outputs

$$\epsilon_c^t = \sum_{k=1}^{K} w_{ck} \delta_k^t + \sum_{g=1}^{G} w_{cg} \delta_g^{t+1} \tag{4.11}$$

Output Gates

$$\delta_\omega^t = f'(a_\omega^t) \sum_{c=1}^{C} h(s_c^t) \epsilon_c^t \tag{4.12}$$

States

$$\epsilon_s^t = b_\omega^t h'(s_c^t) \epsilon_c^t + b_\phi^{t+1} \epsilon_s^{t+1} + w_{c\iota} \delta_\iota^{t+1} + w_{c\phi} \delta_\phi^{t+1} + w_{c\omega} \delta_\omega^t \tag{4.13}$$

Cells

$$\delta_c^t = b_\iota^t g'(a_c^t) \epsilon_s^t \tag{4.14}$$

Forget Gates

$$\delta_\phi^t = f'(a_\phi^t) \sum_{c=1}^{C} s_c^{t-1} \epsilon_s^t \tag{4.15}$$

Input Gates

$$\delta_\iota^t = f'(a_\iota^t) \sum_{c=1}^{C} g(a_c^t) \epsilon_s^t \tag{4.16}$$

Chapter 5

A Comparison of Network Architectures

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 2.3.2). It tests an algorithm's ability to segment and recognise the constituent parts of a speech signal, requires the use of contextual information, and can be regarded as a first step to continuous speech recognition.

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 conclusion 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 input 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. 184 of the training set utterances (chosen randomly, but kept constant for all experiments) were used as a validation set for early stopping. 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 was used for the audio data. The input 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 25ms 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. The preprocessing was carried out using the *Hidden Markov Model Toolkit* (Young et al., 2006).

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 number of weights 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 $\mathcal{O}(W)$ 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).

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 chapter.

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. The same training parameters were used 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. 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.

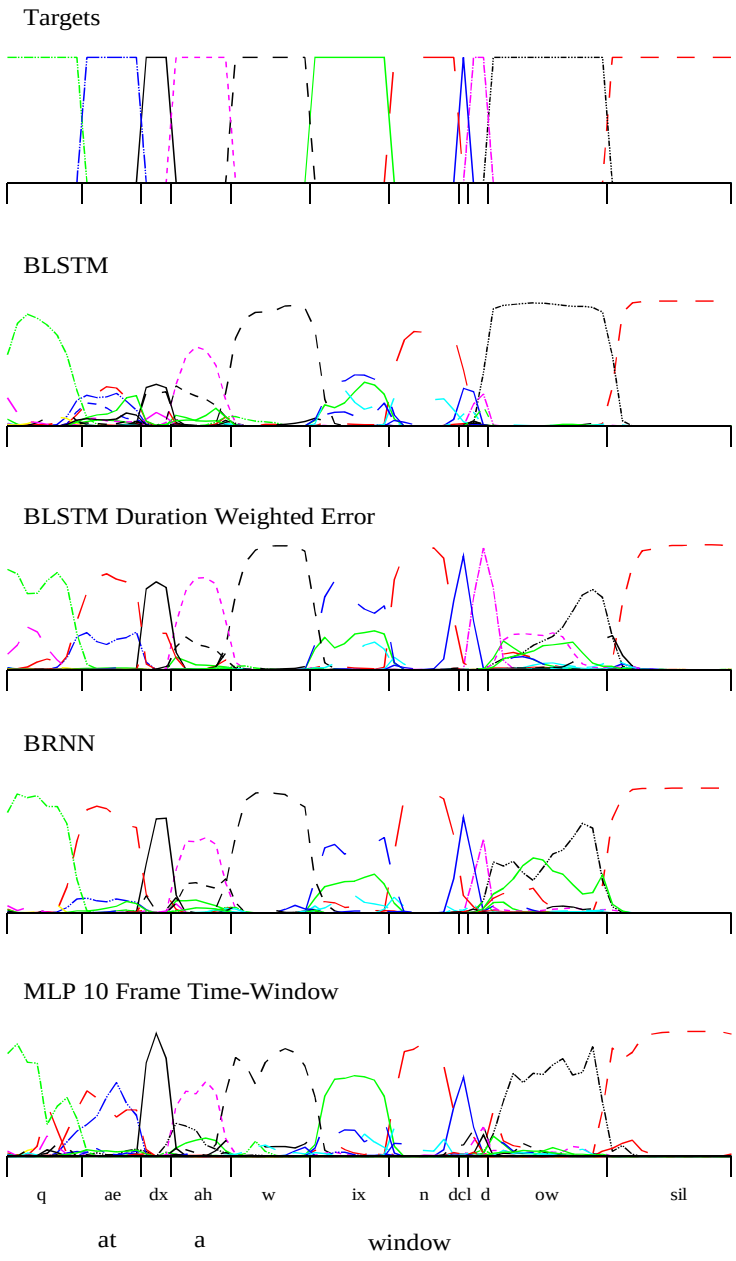

Figure 5.1: **Various networks classifying the excerpt "at a window" from TIMIT.** In general, the networks found the vowels more difficult than the consonants, which in English are more distinct. Adding duration weighted error to BLSTM tends to give better results on short phonemes, (e.g. the closure and stop 'dcl' and 'd'), and worse results on longer ones ('ow'), as expected. Note the more jagged trajectories for the MLP; this is a consequence of not having a recurrent hidden layer, and therefore calculating each output independently of the others.

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.

Note that, other than early stopping, no techniques for improved generalisation were used. It is likely the addition of either input noise (Section 3.3.2.2) or weight noise (Section 3.3.2.3) would have lead to better performance.

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 chapter 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 *transfer learning* or *meta-learning*) which has been widely studied in the neural network and general machine learning literature (see e.g. Giraud-Carrier et al., 2004).

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 eight 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.

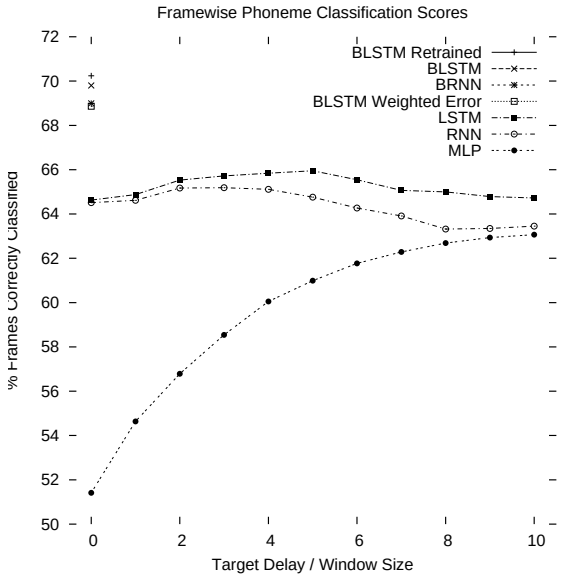

Figure 5.2: **Framewise phoneme classification results on TIMIT.** The number of frames of added context (time-window size for MLPs, target delay size for unidirectional LSTM and RNNs) is plotted along the x axis. The results for the bidirectional networks (which don't require any extra context) are plotted at x=0.

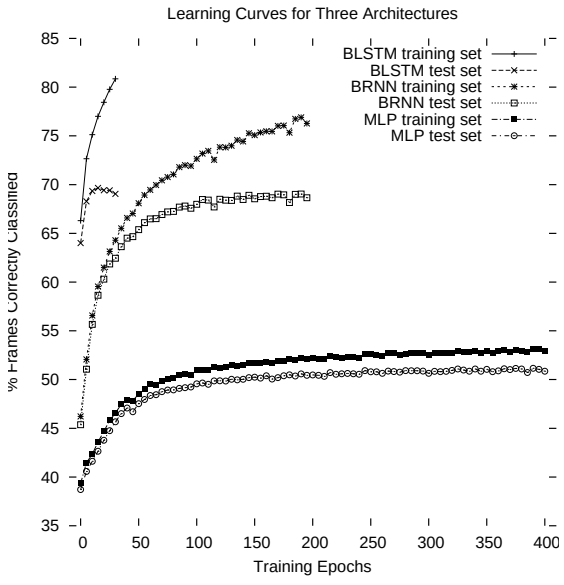

Figure 5.3: **Learning curves on TIMIT for BLSTM, BRNN and MLP with no time-window.** For all experiments, LSTM was much faster to converge than either the RNN or MLP architectures.

Table 5.1: **Framewise phoneme classification results on TIMIT.** The error measure is the frame error rate (percentage of misclassified frames). BLSTM results are means over seven runs ± standard error.

Network	Train Error (%)	Test Error (%)	Epochs
MLP (no window)	46.4	48.6	835
MLP (10 frame window)	32.4	36.9	990
RNN (delay 0)	30.1	35.5	120
LSTM (delay 0)	29.1	35.4	15
LSTM (backwards, delay 0)	29.9	35.3	15
RNN (delay 3)	29.0	34.8	140
LSTM (delay 5)	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

It is interesting to note how much more prone to overfitting LSTM was than standard RNNs. For LSTM, after only fifteen to twenty 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 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 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 the one described by Chen and Jamieson. However this result is questionable as a substantially lower error rate is recorded on the test set than on the training set. Moreover we were unable to reproduce their scores in our own experiments.

In general it is difficult to compare with previous neural network results on this task, owing to variations in network training (different preprocessing, gradient descent algorithms, error functions etc.) and in the task itself (different training and test sets, different numbers of phoneme labels etc.).

Table 5.2: **Comparison of BLSTM with previous network.** The error measure is the frame error rate (percentage of misclassified frames).

Network	Train Error (%)	Test Error (%)
BRNN (Schuster, 1999)	17.9	34.9
RNN (Robinson, 1994)	29.4	34.7
BLSTM (retrained)	21.4	29.8
RNN (Chen and Jamieson, 1996)	30.1	25.8

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 store previous inputs throughout the delay.

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.

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 error 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 error rate of 25.6%, compared to 28.8% with normal errors.

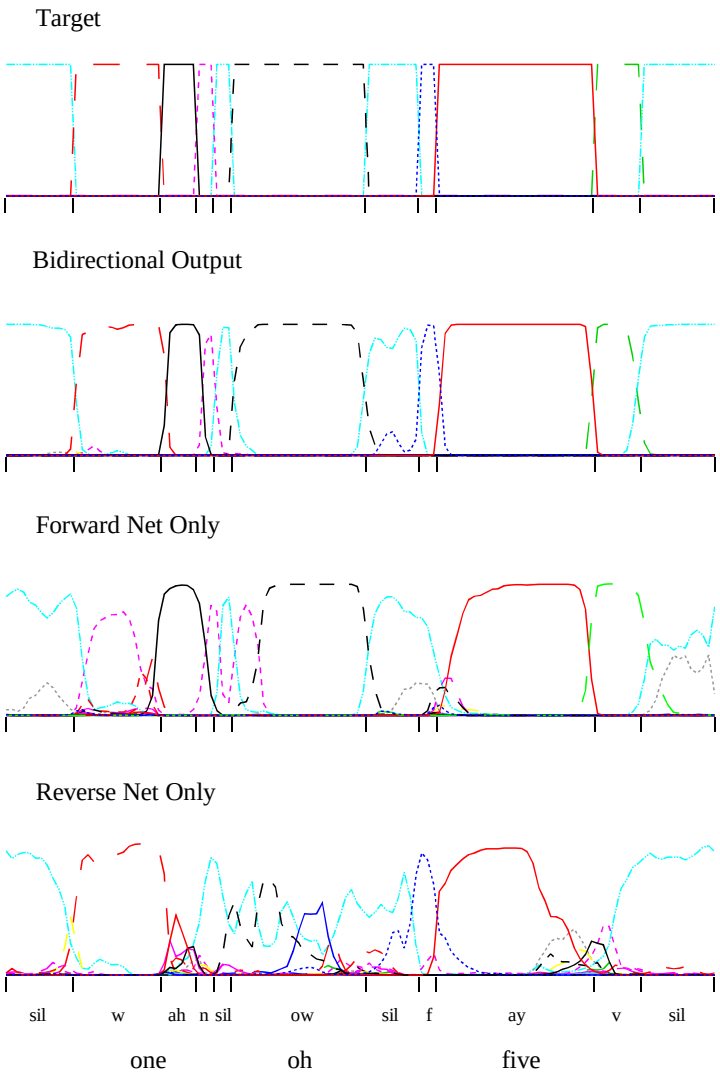

Figure 5.4: **BLSTM network classifying the utterance "one oh five".** The bidirectional output combines the predictions of the forward and backward hidden layers; it closely matches the target, indicating accurate classification. To see how the layers work together, their contributions to the output are plotted separately. In this case the forward layer seems to be more accurate. However there are places where its substitutions ('w'), insertions (at the start of 'ow') and deletions ('f') appear to be corrected by the backward layer. The outputs for the phoneme 'ay' suggests that the layers can work together, with the backward layer finding the start of the segment and the forward layer finding the end.