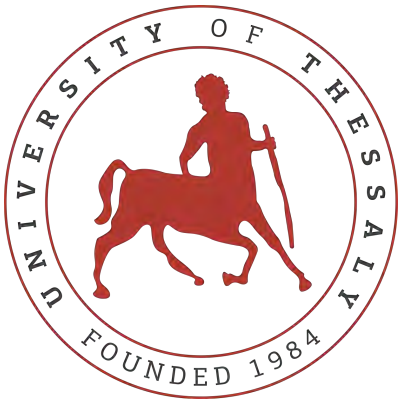
MULTIMODAL GESTURE RECOGNITION WITH THE USE OF DEEP LEARNING



University of Thessaly

Department of electrical and computer engineering

Alexios Gidiotis

supervised by
Dr. Gerasimos Potamianos
Dr. Antonios Argyriou

October 4, 2017

Abstract

Deep recurrent neural networks have recently proved to be very powerful machine learning algorithms. They can have a very large number of trainable parameters which makes them very good at modeling long sequences of information in a very natural way. Recurrent neural networks are now widely used in many machine learning applications and have shown great potential in complicated and difficult tasks.

The recognition of long sequences of gestures in a continuous way have long been a point of interest in the literature. Different approaches include probabilistic models, convolutional neural networks, as well as recurrent neural networks

The aim of this Thesis is to create a state-of-the-art, end-to-end model that uses multiple modalities to detect gestures in a continuous sequence of speech and hand movements performed by the user. In this approach we examine the possibility of recognizing continuous sequences without any segmentation or preprocessing. We also focused on integrating multiple modalities in order to make them work in a cooperative way and produce better results than any unimodal approach. This way we are able to make use of multiple streams of information that might be complementary to each other and combine them without over-fitting the training set. Our work includes experiments on the ChaLearn human gestures dataset. The main focus lies on the audio and skeletal modalities, but this approach can easily be extended to integrate other modalities as well. We also want to demonstrate the advantages of deep recurrent neural network architectures over other sequential algorithms, such as hidden Markov models.

Περίληψη

Τα βαθιά νευρωνικά δίκτυα αποτελούν σήμερα τον πιο διαδεδομένο αλγόριθμο μηχανικής μάθησης. Ο μεγάλος αριθμός παραμέτρων τους δίνει την δυνατότητα να μοντελοποιούν περίπλοκες ακολουθίες δεδομένων με πολύ φυσικό τρόπο. Αυτό έχει ως αποτέλεσμα να χρησιμοποιούνται κατα κόρον για την επίλυση ενός αριθμού προβλημάτων μηχανικής μάθησης.

Η αναγνώριση ακολουθιών από χειρονομίες με συνεχή τρόπο είναι ένα σημείο ενδιαφέροντος για τη βιβλιογραφία. Οι διαφορετικές προσεγγίσεις περιλαμβάνουν πιθανοτικά μοντέλα, συνελικτικά νευρωνικά δίκτυα, καθώς και επαναλαμβανόμενα νευρωνικά δίκτυα.

Σκοπός αυτής της εργασίας είναι η δημιουργία ενός μοντέλου που χρησιμοποιεί έναν συνδυασμό μεθόδων βαθιάς μάθησης για την ανίχνευση χειρονομιών σε μια συνεχή ακολουθία βίντεο. Σε αυτή την προσέγγιση εξετάζουμε τη δυνατότητα αναγνώρισης συνεχών ακολουθιών χωρίς οποιαδήποτε τμηματοποίηση ή προεπεξεργασία. Έχουμε επίσης εστιάσει στην πολυτροπική αναγνώριση ούτως ώστε να πάρουμε όσο το δυνατόν καλύτερα αποτελέσματα. Με τον τρόπο αυτό μπορούμε να χρησιμοποιήσουμε πολλαπλές ροές πληροφοριών που θα μπορούσαν να είναι συμπληρωματικές μεταξύ τους και να τις συνδυάσουμε με αποτελεσματικό τρόπο. Το έργο μας περιλαμβάνει πειράματα σχετικά με την βάση δεδομένων ChaLearn 2013. Σε αυτή την εργασία χρησιμοποιήσαμε κατά κύριο λόγο σκελετική πληροφορία και ομιλία αλλά αυτή η προσέγγιση μπορεί εύκολα να επεκταθεί για να ενσωματώσει και άλλες μορφές πληροφορίας. Θέλουμε επίσης να τονίσουμε τα πλεονεκτήματα των νευρωνικών δικτύων σε σχέση με άλλους αλγόριθμους μηχανικής μάθησης, όπως τα κρυφά Μαρκοβιανά μοντέλα.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In machine learning we often wish to turn a sequence of data into a sequence of classes. Well known examples include handwriting recognition, part-of-speech tagging, keyword spotting, and semantic analysis. What all of this examples have in common is that they take as input a sequence of data, for example a waveform or a sequence of frames, and try to predict a sequence of classes that model the input data. One approach that has been widely used is to segment the input sequence into logical pieces and try to classify each piece into a certain class. Another way to do this is to try and predict a sequence of classes that best models the input data. That is a very natural way to solve the problem. In fact this is probably the way that the human brain actually solves this kind of problems.

Both approaches have previously been studied with a variety of methods being proposed and they both have shown reasonable results. The second approach though seems to be a lot more promising in the long run. When we deal with the problem of Natural Language Processing (NLP) or video processing, the events occurring in the data sequence are rarely independent from one another. This means that if we are able to capture this correlation we are more likely to make accurate predictions on the sequential input. This is a big win in most real-world cases.

In previous years probabilistic models like Hidden Markov Models (HMMs) have given very good results in many problems and they were indeed the most widely used method in many tasks like NLP and keyword spotting. They are able to model inputs that have sequential structure and produce very interesting results using probability distributions and hidden states. The model can go from one hidden state to the next with a transition probability and each state has a probability to produce output.

Recurrent Neural Networks (RNNs) are similar to HMMs in the way that they are able to model data with sequential structure. They also have a hidden state and are able to go from one hidden state to another as well as produce outputs. Although at first they seem to be very similar in the way they model data, they are indeed very different. For example, at each time step the current state of the network may depend on both the input and the previous hidden state which allows it to develop very complex dynamics. The purpose of this

thesis is to develop a model that uses RNNs for a task that was previously solved with HMMs and explore the advantages of these more powerful models.

While RNNs are in theory very good at modeling sequences, in practice they face some limitations. Long Short-Term Memory (LSTM) is an RNN architecture designed to overcome such limitations. LSTM networks are very good at remembering information for a very long time. This property makes them capable of bridging very long time lags between relevant input and target events. In many real-applications LSTM networks have proven to be superior to simple RNNs. For that reason we are going to use LSTMs instead of simple RNNs throughout this thesis.

In this thesis we will be using deep LSTM networks to address the problem of gesture recognition. In order to more effectively solve the problem of gesture recognition we will be utilizing multiple input modalities that will be able to cooperate with one another and produce good classification results. The focus of this thesis is the implementation and training of multiple LSTM networks, one for each modality, and finally the integration of the different networks in one more powerful model. This model is going to take as input unsegmented sequences of data and try to spot the different gestures performed.

In our experiments we will be using the proposed model in order to solve the problem of multimodal gesture recognition on the especially challenging ChaLearn 2013 data set [7]. This particular data set includes a large number of labeled training and evaluation examples of different users performing a number of athropological gestures. The data set comes with a number of different streams including audio, video and skeletal information and is especially suitable for multimodal recognition tasks.

1.1 Contributions

We are going to use LSTM units with memory cells to build our networks [16]. This units have the nice property that they remember information for a long time. We combine LSTM with bidirectional RNNs [26] resulting in bidirectional LSTMs (BLSTMs), which process the input data both forward and backwards. This makes it easier to train powerful LSTM models with relatively few data without over-fitting. In order to make training even easier we add residual connections between layers [14] that are known to speed up the training procedure by propagating the full gradient many layers deep. This combination of residual bidirectional LSTM blocks has given us very promising results in a variety of tasks.

In order to train the networks directly for the sequence labeling tasks with unknown input-label alignments we employed Connectionist Temporal Classification (CTC) [11]. We approach the task with an end-to-end machine learning model. The trained model is able to perform alignment and recognition at the same time. That way we can get rid of the long pipelines of different models that perform these tasks separately. Our model is able to take as input raw data sequences and produce a sequence of class predictions.

These properties make the training of the network much simpler as well.

We can simply just train the model with long sequences of unsegmented data and provide only loosely aligned label sequences without any information about the precise timings of the events. Also no background model was necessary in order to model out-of-vocabulary events.

Also in order to make full use of the multiple modalities available, such as audio, skeletal information, RGB and depth, we propose an architecture that integrates previously trained unimodal networks and forces them to cooperate in order to improve gesture spotting results. The multi-modal recognition of gestures has previously been proposed both for neural network architectures and HMMs and has given much better results than unimodal approaches. In this thesis we propose a way of combining multiple BLSTM networks in an effective way. This is done by another network that takes as input the outputs of the unimodal networks, merges them and uses them to make the final predictions. We worked with audio and skeletal data in our experiments but the proposed method can also be used to integrate other different modalities. The proposed architecture demonstrates the power of RNNs and their ability to learn to perform complicated tasks with very little hand-engineering.

Another advantage that the proposed integration method has is that the different modalities can also have different alignments. For example, when a gesture is performed, the alignment of the gesture itself is different from the alignment of speech. The single modality sub-networks are trained to model the data in a way that fits each specific modality. Then the high level network takes the different outputs and combines them in order to make the final predictions in a multi-modal way.

Finally, a big part of our work was regularizing the models proposed. BLSTM networks with millions of trainable parameters can very easily overfit the training set and thus generalize very poorly. As a result they tend to perform very well on the training set but very poorly with unseen data. We used dropout [27] along with weight constraints and Gaussian noise as a way to better regularize the different networks and avoid over-fitting.

1.2 Overview of Thesis

The chapters in this thesis are grouped into roughly two parts. Chapters 2-6, include background material about the methods used in the thesis. In chapters 7-9 we discuss the experimental setup and the proposed model.

Chapter 2 is a brief introduction to supervised learning and particularly sequence modeling. Probabilistic methods that were widely used in the past are briefly presented here in order to compare them to the deep learning methods that are used in this thesis. In chapter 3 we provide background material for feed-forward and recurrent neural networks. Neural networks are the main focus of the thesis and here we present the basic concepts of neural network architectures and training. Chapter 4 describes LSTM networks and especially the bidirectional LSTM (BLSTM) networks that we will be using in our experiments. Chapter 5 introduces connectionist temporal classification (CTC) and the equations behind it as well as the forward-backward algorithm that is

used for the training of networks with CTC.

In Chapter 6 we describe the methodology proposed in the thesis. This chapter includes the details about feature extraction, network architectures as well as the method used for the integration of the different modalities. In this chapter we also present the ChaLearn dataset that was used in our experiments.

Chapter 7 includes the training of the model along with the optimizations and hyper-parameters we have been using. Here we evaluate the performance of the proposed method and compare it with the results from different probabilistic methods. Concluding remarks and directions for future work are given in chapter 8.

Chapter 2

Learning Sequences

This chapter provides the background for supervised sequence learning. Section 2.1 introduces briefly the basic concepts of supervised learning in general. Section 2.2 covers the basics of sequence modeling and recognition while section 2.3 presents various machine learning tasks that involve the modeling of data sequences. Section 2.4 presents basic probabilistic algorithms that have been used to solve this kind of problems. This algorithms do not involve deep learning and are presented here in order to contrast them with the deep learning approach of this thesis.

2.1 Supervised Learning

Machine learning problems are generally split into three categories. Supervised learning refers to the problems where a set of input-target pairs is provided for training. Reinforcement learning refers to problems where no specific target is given for each input but instead a positive or a negative reward is provided for training. Finally unsupervised learning includes problems where no training signal is provided at all and the algorithm attempts to find the structure of the data by observing them.

The problem of sequence recognition that we are studying in this thesis falls into the first category and thus we will briefly present the basic concepts here. A standard supervised learning task consists of a training set S that includes input-target pairs (x, y) where x is an element of the input space X and y is the ground truth corresponding to this input element. A machine learning algorithm learns to predict the target y when given the input x. In order to achieve this the algorithm uses the training set to minimize some task specific error metric. An error metric is usually a metric the measures the distance between the algorithm outputs given an input and the target value. The process of minimizing the error metric on the training set is generally referred to as learning. Once a machine learning algorithm has successfully learned from the training data, it can correctly predict targets when given other similar inputs.

Different algorithms use different ways of minimizing the error metric. For example, for most regression tasks we use the squared Euclidean distance be-

tween the algorithm outputs and the target vectors as an error metric and regression algorithms minimize this metric by iteratively adjusting their parameters based on the training inputs. Neural networks use the derivatives of the error to incrementally adjust the algorithm parameters in order to optimize some objective function on the training set.

The ability of an algorithm to transfer performance from the training set to unseen data is usually referred to as generalization. Generalization is a key concept of machine learning in general and will be discussed in more detail later on in this thesis.

2.2 Modeling Data Sequences

In many real world tasks we find that the data exhibit some temporal structure. This means that events occur in a sequence and there is usually some dependence between them. Also many times the order in which the events occur is important for the task at hand. We often want to turn one sequence into another. For example we might want to take English words and translate them into words in Greek. Or we might want to turn a waveform into a sequence of phonemes and words, which is what happens in speech recognition. Sometimes we want to take a sequence of inputs and try to predict the next term. Take for example a sequence of nucleotides and try to predict the next one.

When dealing with these tasks we need models that can capture the sequential structure of the data and make use of it in order to produce outputs. Often the outputs at one time-step depend not only on the inputs at that particular time-step but also on inputs at previous time-steps. So we need algorithms that are able to maintain a state, visible or hidden, and remember events that happened in the past.

Since the problem we are studying in this thesis is strictly supervised we are going to focus on sequence recognition tasks. Of course these are not the only kind of problems that have sequential structure but they cover quite a lot of tasks. Because we are focusing on supervised recognition we are going to assume that there is always a training set of data sequence-target pairs. The classes that we want to recognize are part of an alphabet of symbols $\{E1, E2, ..., En\}$ which is a-priori specified and finite.

The labeling of the training data may also differ from one task to another. Some tasks require precise alignment of the labels while for others this might be unimportant. In this thesis we focus on labels that are loosely aligned and the approach proposed here outputs only the final sequence of labels and not the precise time when they occurred.

The type of label sequence required can also depend on the task at hand. Based on the task we can have three different types of sequence labeling. The type of sequence labeling can affect both the training algorithm and the error metrics it minimizes.

The most restrictive case is sequence classification. In this case we assign a single class to a sequence of inputs and so the length of the label sequence

is strictly one. Examples of sequence classification are the recognition of an individual keyword in a part of speech or the recognition of an action performed in a video part.

The second type of sequence labeling is segment classification. This applies to the cases where we need to predict multiple classes for a single input sequence. For example we might require to classify each phoneme in a sequence of spoken words. If we know in advance the exact positions where the different phonemes occur then we can run a classification algorithm for each segment. This approach is referred to as segment classification and has shown very good results in a number of tasks. The downside of this approach is that it requires the input to be pre-segmented, which is not always an easy thing to do.

The third and most general case is called temporal classification. Here no assumptions can be made about the label sequences except that their length is less than or equal to the length of the input sequences. These tasks require an algorithm that is able to decide where in the input sequences the classifications should be made. Temporal classification is very useful in cases where the alignment between inputs and label sequences is unknown. The learning procedure of these algorithms is slower and much more difficult than segment classification. On the other hand these methods are applicable in many more real world cases where we only know the classes that occur in an input sequence and we have no information about their exact timings.

2.3 Applications of Sequence Recognition

Here we are going to present briefly some machine learning tasks which require specific algorithms that can model sequential structures and in the next section we are going to introduce some algorithms besides neural networks that were used to address them.

One of the most obvious tasks where the data have temporal structure is natural language processing (NLP). NLP tasks include keyword spotting, understanding speech, machine translation and speaker diarization. In all of these tasks the inputs are usually sequences of sound pressures. For different tasks we expect the algorithms to produce different outputs. For example, in keyword spotting we want to know if certain keywords from a pre-defined vocabulary of keywords occur in a sequence of speech. We do not really care about the other out-of-vocabulary words that might occur and so we need to ignore them. Discriminative algorithms are better suited for keyword spotting as the keywords are usually a small proportion of the speech sequence. Natural language understanding is slightly different from keyword spotting. Here we want to recognize a sequence of phonemes and turn them into word entities. This means that the algorithms cannot ignore anything because in speech the meaning of a word entity greatly depends on the previous and following words. Generative models are very good at modeling long sequences of speech and can capture long term dependences. In machine translation we want to turn a sequence of words from one language to another. Of course this is easier said than done because different languages may have much different

structure. Speaker diarization is the process of partitioning an input audio sequence into homogeneous segments according to the speaker identity. This requires algorithms that are able to learn general information about speech and can discriminate between different speakers.

Handwriting recognition is quite similar to NLP. In this task though the inputs are images and frames instead of sound pressures. Again we might need to spot specific keywords and characters from a pre-defined vocabulary or we might want to understand unconstrained sequences in order to perform some other task. What makes things slightly different is that frames and videos have spatial as well as temporal structure which may require a combination of different machine learning algorithms in order to get sufficiently good results.

DNA sequencing and analysis is another field where sequential machine learning algorithms are proving to be quite effective. Here the inputs lie in a different domain than speech and handwriting. Instead we need to order, analyze and decode sequences of nucleotides drawn from a very small set of four bases $\{A, G, T, C\}$. These bases form very long chains that encode genes. This task has numerous applications in molecular biology, medicine and forensics. Because DNA strands are huge we need algorithms that are able to capture extremely long term dependences in an efficient way.

2.4 Probabilistic Methods and Hidden Markov Models

Here we are going to present different models, that are not neural networks, which have been used in the past in order to solve the problem of sequence modeling. These include memoryless models, like autoregressive models, and models that are able to remember information using a hidden state. We are going to focus on the latter because they are significantly more complicated and much more interesting. A model with a hidden state can store information in its hidden state for a long time. Usually in this kind of models a probability distribution is inferred and we are able to predict the hidden state by observing outputs that follow this probability distribution. Because we assume that the outputs follow a probability distribution, these models are called probabilistic. There are mainly two different models that have these properties. The first is a Linear Dynamical System (LDS) and the other is a hidden Markov Model (HMM).

A Linear Dynamical System is a generative model that has a linear hidden state and is able to produce observations. The LDS may also have driving inputs that affect the hidden state. The outputs of the LDS depend on the current hidden state and the next hidden state depends on the current hidden state and the inputs. A nice property of this model is that the distribution over the hidden state given the observations is Gaussian which can be computed efficiently.

A hidden Markov Model is another hidden state, probabilistic model that uses a discrete distribution rather than a Gaussian one. The HMM can be in exactly one hidden state each time. The transition from one hidden state to

the next is probabilistic and controlled by a transition matrix and we assume that the observed outputs follow a probability distribution which is usually a mixture of Gaussians.

Given a data set $D = \{x_1, ..., x_N\}$ and assuming that $x_n \in R^d$ we can define a Gaussian mixture model (GMM) by making each of the K components of a mixture a Gaussian with parameters μ_k and Σ_k. Each component is a multivariate Gaussian with its own parameters $\theta_k = \{\mu_k, \Sigma_k\}$.

$$p_k(x|\theta_k) = \frac{1}{(2\pi)^{d/2} \mid \Sigma_k \mid^{1/2}} \exp^{-\frac{1}{2}(x-\mu_k)^t \Sigma_k^{-1}(x-\mu_k)} \tag{2.1}$$

When using HMMs to model data sequences we generally have to deal with three distinct problems. These are the evaluation, the decoding and the learning problem. With the evaluation problem we are given an observed sequence $O = O_1O_2...O_t$ and a model $\lambda = (A, B, \pi)$ and try to find the probability that the model produced this sequence $P(O \mid \lambda)$.

$$P(O \mid \lambda) = \sum_Q P(O \mid Q, \lambda)P(Q \mid \lambda) \tag{2.2}$$

In the decoding problem we are trying, given an observed sequence and a model, to find the most probable sequence of hidden states that produced the observed sequence. This is done with dynamic programming and the Viterbi algorithm.

$$Q^* = q_1q_2q_3..q_t : Q^* = argmax_Q P(Q \mid O, \lambda) \tag{2.3}$$

Finally we have the learning problem. In the learning problem we need to find the model parameters that give the model a high probability of producing the observed sequence. For HMMs these parameters can be estimated with the Baum-Welch algorithm. The algorithm uses the Expectation Maximization algorithm (EM) in order to find the maximum likelihood estimation of the model parameters given the set of observed feature vectors.

$$\lambda^* = argmax_\lambda P(O \mid \lambda) \tag{2.4}$$

It was because of these algorithms that HMMs took over the sequence recognition field and they were the mainline in most applications from speech recognition to keyword spotting, diarization and bioinformatics. In this thesis we are mainly going to use HMMs as a benchmark in order to see how well our approach performs versus the standard HMM approaches. We actually have implemented a model that uses HMMs and we are going to compare it with the model proposed in this thesis.

One of the fundamental limitation of HMMs is that the model can only be at one hidden state at each time. This means that in order to remember information it generated a long time a go, an HMM needs exponentially as many states. For example if an HMM needs to convey 100 bits of information from the first half of an utterance to the second it will require as many as 2^{100} hidden states. That limitation is what brings us to Recurrent Neural Networks. We are going to describe them in more detail in the next chapter.

Chapter 3

Neural Networks

This chapter provides background material for neural networks. We are going to focus on neural networks that are used for supervised classification tasks and especialy RNNs as we are going to use them in this thesis. Section 3.1 introduces the basic concepts of simple feed-forward neural networks. We are going to discuss different feed-forward architectures used for classification tasks. We will also describe hidden unit activations, the optimization function and back-propagation training. In section 3.2 we will review RNNs and their application to sequence labeling. Also we are going to extend back-propagation training to work with RNNs. Here we present simple RNN networks and in the next chapter we are going to introduce LSTM networks. In section 3.3 we are going to discuss various topics that are essential for training neural networks. These topics include generalization and optimization as well as other issues like vanishing and exploding gradients.

3.1 Feed-forward Neural Networks

Artificial Neural Networks (ANNs) are mathematical models that are able to process information in a non linear way. When neural networks were first introduced, researchers believed that this was the way that the human brain worked. Of course now we know that they bear little resemblance to real biological neurons.

An ANN consists of small processing units, which are called neurons (Figure 3.1), that are able to take inputs and apply some function on them.

Here we present the basic function of a neuron that takes N inputs and a bias term b and produces an output v. With w_n we denote the weights of the incoming connections.

$$v = \sum_{n=1}^{N} w_n x_n + b \tag{3.1}$$

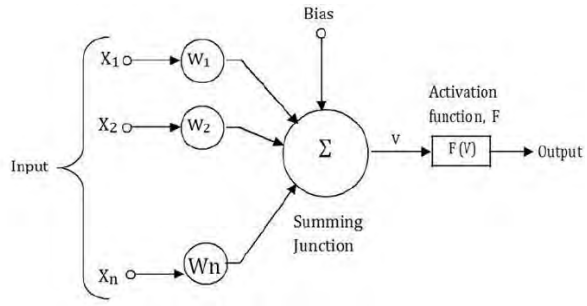


Figure 3.1: **Basic Neuron**

3.1.1 Activation Functions

If we want the output of the neuron to be non-linear we can apply a different activation function to the output v. The most common activation function for hidden neurons is the logistic sigmoid. With z we denote the output of a single hidden neuron.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3.2}$$

Other common activation functions are the hyperbolic tangent (tanh) and the rectified linear (ReLU).

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

$$relu(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases} \tag{3.4}$$

All of these activations are used in practice. Usually the choice of activation function is architecture specific and has to be decided when designing a new model. For example ReLUs are usually preferred for feed-forward networks over sigmoids because they are much faster to compute. On the other hand sigmoids and hyperbolic tangents are usually a good choice for RNNs. Also there are other activation functions that are basically variations of the ones presented here. In Figure 3.2 are presented some commonly used activation functions.

3.1.2 Forward Pass

A deep feed-forward network is usually constructed from many layers of neurons and is able to learn complicated functions. The layers of neurons in a neural network are called hidden layers. A simple feed-forward network usually has an input layer, an output layer and one or more hidden layers (Figure 3.3). The neurons of each hidden layer are also called hidden units. The weights connecting the hidden units of a layer with the previous and the next layer are the networks' learnable parameters. A network with many hidden units

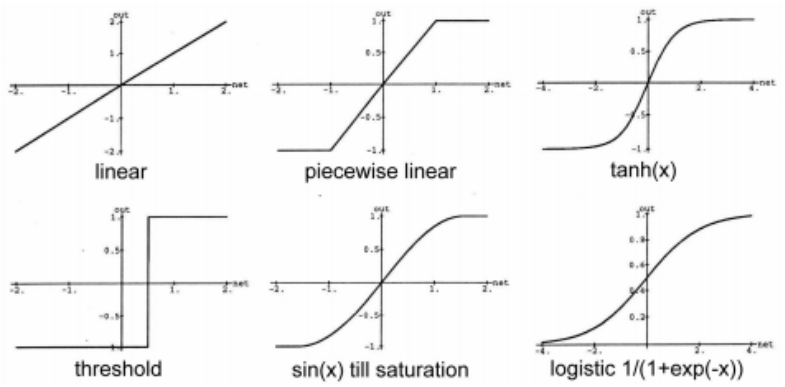


Figure 3.2: **Activation functions** [10]

has a lot of learnable parameters and is able to learn more complicated patterns. Experiments have shown that deeper networks with less hidden units per layer have a larger capacity and are able to learn better features than shallow networks with more hidden units per layer.

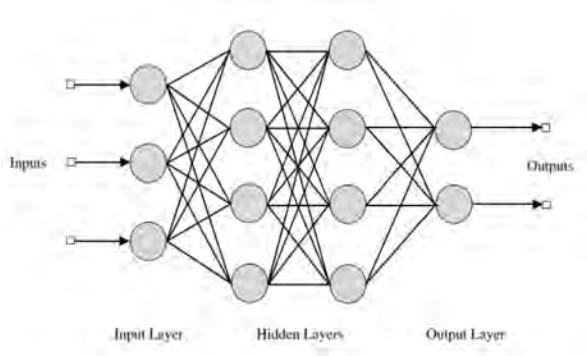


Figure 3.3: **Feed-forward neural network**

Once a network is trained we can get outputs by forward-propagating the inputs through the network. For the network shown in Figure 3.3 the forward pass includes the following steps. Here with $z_{j,i}$ we denote the output of the i_{th} hidden unit of the j_{th} layer and with x_n the n_{th} input. Let us assume the hidden units also have sigmoid activations. With $a_{j,i}$ we denote the output of the i_{th} unit of the j_{th} layer after the sigmoid activation. With $w_{j,i,n}$ is denoted the incoming weight connection of the i_{th} unit of the j_{th} layer with the n_{th} unit of the $j-1_{th}$ layer and $b_{j,i}$ is the bias of the i_{th} unit of the j_{th} layer.

$$z_{1,i} = \sum_{n=1}^{N} w_{1,i,n} x_n + b_{1,i} \tag{3.5}$$

$$a_{1,i} = \sigma(z_{1,i}) \tag{3.6}$$

$$z_{2,i} = \sum_{n=1}^{N} w_{2,i,n} a_{1,n} + b_{2,i} \tag{3.7}$$

$$a_{2,i} = \sigma(z_{2,i}) \tag{3.8}$$

$$z_{3,i} = \sum_{n=1}^{N} w_{3,i,n} a_{2,n} + b_{3,i} \tag{3.9}$$

3.1.3 Optimization Objective

The most common use of neural networks is for multi-class classification. We usually want the network to take an input vector and predict one of L discrete classes. In order to do this we add an L-way softmax after the final layer. The softmax will turn the outputs into a probability distribution over the L classes where the most probable class is predicted. With $softmax(z)_i$ is denoted the i_{th} element of the L-way softmax. The softmax turns an L-dimensional vector of arbitrary real values and turns it into a probability distribution where all values sum up to one.

$$softmax(z)_i = \frac{e^{z_i}}{\sum_{l=1}^{L} e^{z_l}} \tag{3.10}$$

Because the output of the network is a softmax unit, we need an appropriate error metric also called a cost function. The cost function measures how different is the current output from the desired output of the network. When the output of the network is a probability distribution, the appropriate cost function is the negative log probability of the right answer. This function is called cross-entropy. With t_j we denote the j_{th} target value while y_j is the current output for the j_{th} training example.

$$E = -\sum_{j=1}^{J} t_j log(y_j) \tag{3.11}$$

When training a neural network, what we do is optimize the cross-entropy cost function. The way that we do this is by using the gradient of the cost-function to iteratively adjust the parameters of the network in order to minimize the cross-entropy. Minimizing the cross-entropy is exactly equivalent to maximizing the probability of getting the correct answer. An important detail in order to use the cross-entropy cost function is to have an appropriate target value.

The target value is an L dimensional vector that has one at the position of the right answer and zeros at the positions of all the wrong answers. This way we can very easily compute the cross-entropy of every output-target pair.

3.1.4 Back-propagation

The cross-entropy cost function is differentiable and has a very simple derivative. The optimization of the cost function can be efficiently done with gradient descent methods. The cross-entropy term defined above is a sum over all input-target pairs in the training set. As a result the derivatives of the objective function is also a sum of derivative terms. When referring to the derivatives of the objective function in our equations, we implicitly mean the derivatives of a single input-target pair.

Back-propagation [24][31] is a method used to efficiently calculate the gradient. This is often referred to as the backward pass of the network. Back-propagation is simply a repeated application of the chain rule for partial derivatives. We first calculate the derivative of the objective function with respect to the output unit and the derivative of the softmax output with respect to the input of the softmax unit. Then we apply the chain rule in order to calculate the derivative of the objective function with respect to the input of the softmax unit. Here y_j is the output of the j_{th} unit while z_j is the input of the j_{th} unit.

$$\frac{\partial E}{\partial y_j} = \frac{y_j - t}{y_j(1 - y_j)} \tag{3.12}$$

$$\frac{\partial y_j}{\partial z_j} = y_j(1 - y_j) \tag{3.13}$$

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j} = y_j - t \tag{3.14}$$

We now continue to apply the chain rule, going backwards through the hidden layers. When we want to back-propagate the derivative of the j_{th} unit to the i_{th} unit, which belongs to the previous layer, we get the following equations. By repeatedly applying the chain rule we can back-propagate the error derivatives all the way through the network.

$$\frac{\partial E}{\partial y_j} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial y_i} = \sum_j w_{i,j}\frac{\partial E}{\partial z_j} \tag{3.15}$$

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial w_{i,j}} = y_i\frac{\partial E}{\partial z_j} \tag{3.16}$$

Now that we have a good way of computing the error derivatives for every weight, we can specify a learning procedure that adjusts the weights based on these derivatives. The learning procedure simply adjusts each weight towards the direction where the derivative of that weight is the steepest.

There are also two issues one needs to take into account when training a neural network. The first is how often to update the weights and the second is how much to update. A number of different optimizers have been developed that optimize all these hyper-parameters and apply the learning rule for the whole training set. We are going to briefly present some of these optimizers in section 3.3.

3.2 Recurrent Neural Networks

In the previous section we have discussed feed-forward neural networks that only have connections from one layer to the next. Recurrent neural networks (RNNs) also have recurrent connections between units of the same layer. This architecture allows RNNs to have a hidden state which makes them very good for modeling data sequences. An RNN is able to change its hidden state based on the inputs and produce outputs based on the current hidden state as well as the inputs.

In the previous chapter we presented HMMs that are able to model sequences of data. We have also shown that a key weakness of HMMs is that they can only be at one, out of a finite number, state at each particular time step. On the other hand RNNs have a distributed hidden state which allows them to store a lot of information efficiently. This means that several different units are active at once, and as a result the network can remember several different things at once. Also RNNs are non-linear models that are able to develop very complicated dynamics and change state in a non-linear way. Neural networks in contrast to HMMs are not stochastic models. RNNs are deterministic models which means that if you know the current hidden state and the input you can predict the next hidden state.

RNNs can exhibit many different behaviors. They can oscillate in a certain state or settle to point attractors. They can also behave chaotically. These different behaviors make RNNs good at performing many different tasks. In this thesis we are going to focus on their ability to recognize very long data sequences in a more efficient way than HMMs. We are going to briefly describe the way RNNs are able to model sequences and we are going to present a variation of standard back-propagation that can effectively compute the error derivatives with respect to weights for a recurrent neural network. Finally we will present a different version of RNN that can process data both in a forward and a backward order. These are called bidirectional RNNs and have proved to be really effective at many machine learning tasks.

3.2.1 Forward Pass

Here we are going to focus on a simple RNN that has only one hidden layer with recurrent connections. We can think of this RNN as an extension of a feed-forward network with one hidden layer for every time step. We will assume that there is a time delay of one time step in using each connection. Now we can see the recurrent net as a layered net that uses the same weights at every time step.

The RNN begins at an initial state at time zero. Then it uses the inputs and the weights of its recurrent connection, for each time step, to get to a new state. The inputs are forward propagated through the network at each time step and the previous state is propagated using the recurrent connections. This way the current hidden state depends on both the inputs at that particular time step and the previous hidden state.

It is also possible for the network to produce outputs at each time step.

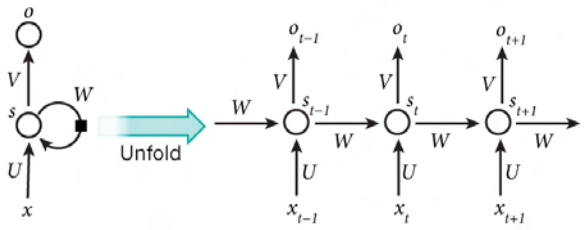


Figure 3.4: **RNN unfolded through time** [19]

Sometimes we may want to have an output at every time step, for example when tracking a missile or driving a car, or we may want to get an output at the end of a specific sequence. In the networks that we are using for our experiments we want the latter as we would like for the network to produce an output class every time it recognizes a gesture.

3.2.2 Back-propagation Through Time

Because of the weight constraints of recurrent connections, we can effectively use back-propagation to find the error derivatives with respect to the hidden weights for each time step and go all the way back to the initial state.

Back-propagation through time (BPTT) [31] is a well known algorithm that can efficiently calculate the error derivatives with respect to the hidden weights for RNNs. BPTT is both conceptually simple and efficient in computation time (though not in memory). Like standard back-propagation, BPTT repeatedly applies the chain rule. The difference with standard back-propagation is that, for recurrent networks, the objective function depends on the activation of the hidden layer not only through its influence on the output layer, but also through its influence on the hidden layer at the next time-step.

$$\frac{\partial E}{\partial w_{i,j}} = \sum_{t=1}^{T} \frac{\partial E}{\partial z_j^t} \frac{\partial z_j^t}{\partial w_{i,j}} = \sum_{t=1}^{T} y_i^t \frac{\partial E}{\partial z_j^t} \tag{3.17}$$

3.2.3 Bidirectional RNNs

Often in sequential modeling tasks, we want to have access to future context as well as past. For example when we are trying to recognize a certain phoneme, we would like to know the phonemes coming at the next time steps as well as the previous ones. Bidirectional RNNs [26][1][13] offer an elegant way to do this. The basic idea is to process the data sequence both forwards and backwards. This is done by two separate hidden layers that are connected to the same output. This way the network has access to both future and past context when processing each time step of the input sequence. BRNNs have given promising results in a number of problems including speech processing.

In the forward pass both hidden layers are presented the same sequence but in opposite orders. The output is updated once both networks have finished processing the whole sequence. For the backward pass we use again BPTT.

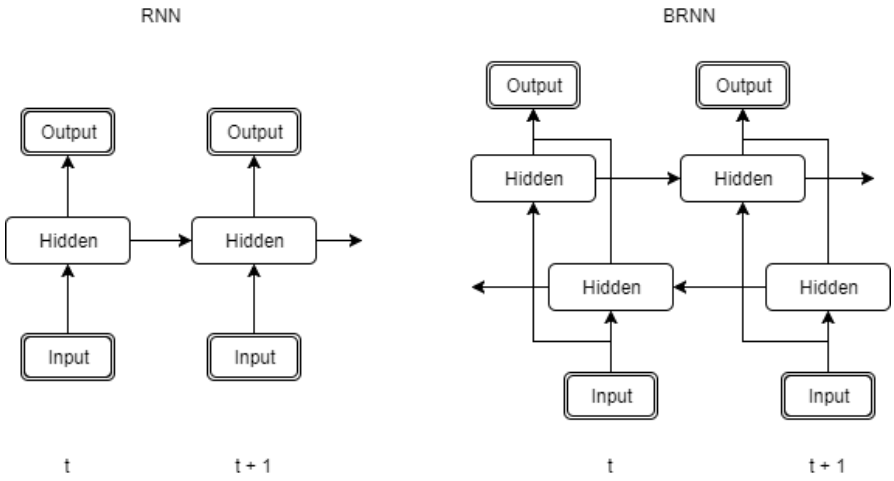


Figure 3.5: **RNN and BRNN**

We compute the derivatives for the output and then back-propagate them, using the chain rule, to calculate the error derivatives with respect to hidden weights for both networks in the layer.

Although BRNNs at first glance violate causality there are many temporal tasks where the model is expected to produce outputs after processing the whole sequence. This means that BRNNs are a feasible solution for all of these tasks and have been widely used.

Another aspect of BRNNs is that since the network processes the data both ways, this acts as a strong regularizer for the network. Indeed BRNNs have shown improved generalization with new data and they do so with significantly less training data than simple RNNs. Bidirectional networks are a little less prone to over-fitting the training set than simple RNNs which, as we are going to see in the next section, is a useful property to have when training a neural network.

3.3 Training Neural Networks

Until this point we have discussed the back-propagation algorithm for feed-forward and recurrent neural networks as an efficient way to compute the error derivatives with respect to the weights for each hidden unit. However in order to train a neural network effectively and efficiently, several issues have to be resolved.

3.3.1 Optimizers

Once the error derivatives have been computed for all weight connections, we need to update these connections towards the direction of the steepest gradient descent. Before we can update the weights though we need to answer two questions. How often to update the weights and how much to update them.

Usually when we are dealing with large amounts of data we cannot simply go through all the examples and then update the weights. We usually process the data in mini-batches of some size. The size of the mini-batch is an important hyper-parameter that we need to carefully choose before the training. A very large batch size is more robust but it can be quite expensive to compute the error derivatives. On the other hand if the batch size is very small we can end up updating the weights in the wrong direction. A careful selection of the batch size is key to an efficient training.

Also we have to decide how much to update the weights every time. This is largely controlled by the learning rate. A number of different optimizers have been developed that are able to use complicated weight update rules and use a number of hyper-parameters in order to optimize the learning process. Some of the most widely used optimizers are SGD, RMSprop, adadelta, adam and nadam.

SGD

Stochastic gradient descent is the simplest optimizer that performs gradient descent learning on mini-batches. The learning procedure is mainly controlled by the learning rate and can also use more advanced methods like momentum, that uses the gradient to change the velocity of the weight update instead of the position, and learning the rate decay over time.

RMSprop

RMSprop [30] is a learning algorithm that keeps a moving average of the squared gradient for each weight.

$$MeanSquare(w,t) = 0.9MeanSquare(w,t-1) + 0.1\left(\frac{\partial E}{\partial w(t)}\right)^2$$

Then it divides the gradient by $\sqrt{MeanSquare(w,t)}$. RMSprop can also be combined with momentum [15] and adaptive learning rates.

Adadelta

Adadelta [36] is another method that dynamically adapts over time and uses a pre-defined set of hyper-parameters to make the training more efficient.

Adam

Adam [18] is another optimizer that is based on the idea of RMSprop and uses adaptive estimates to find the learning step. Adam also pre-defines the hyper-parameters for efficient training.

Nadam

Nadam [5] is essentially a version of Adam that uses the Nesterov momentum. For this purpose we first take a step towards the direction of the steepest

gradient and then estimate the error and then re-adjust the weights towards the correct direction. The Nesterov momentum is known for accelerating the training process.

3.3.2 Regularization and Generalization

Although the learning of the parameters of a neural network is performed on the training set, the real objective is to achieve a good performance on previously unseen data. The issue of whether training set performance carries over to the test set is referred to as generalization, and is of fundamental importance to machine learning.

Neural networks usually have a large number of learnable parameters which allows them to capture complicated dynamics. At the same time a neural network with many learnable parameters is very good at learning the sampling error of the training set as well. This is usually called over-fitting and leads to a model that performs very well on the training set but very poorly on unseen data. In order to address this problem we can either try to design a neural network that does not have too many trainable parameters or use a number of different techniques to regularize a large network. Generally very deep neural networks with many learnable parameters are preferred as they can learn a lot more complicated dynamics. Here we are going to discuss some of the many ways to regularize large neural networks and prevent over-fitting.

Dropout

Dropout [27] is an effective way to prevent the hidden units of a network from co-adapting too much. The key idea is to randomly set a number of hidden connections to zero during each iteration over the training set. This significantly reduces the ability of the network to learn the sampling error and greatly reduces over-fitting.

When properly applied, dropout allows us to use much larger networks that are able to learn complicated patterns without over-fitting the training set. When choosing the dropout percentage for each layer one must be careful though. A large percentage of dropout can be a great regularizer but can also lead to under-fitting and result to networks that are untrainable.

Dropout can also be applied to RNNs in order to reduce over-fitting. It is particularly helpful because RNNs tend to overfit the training set quite easily. With RNNs we can also choose to apply dropout to either the feed-forward or the recurrent connections. It is usually preferable though to apply dropout only to the feed-forward connections and not the recurrent connections [35]. Dropout effectively corrupts the information we want to carry from one time step to the next and this makes the training very hard.

Weight Constraints

Another way that we can regularize a neural network is to constrain the weights to lie in a circle. We set an upper bound on the L2 norm of the weights

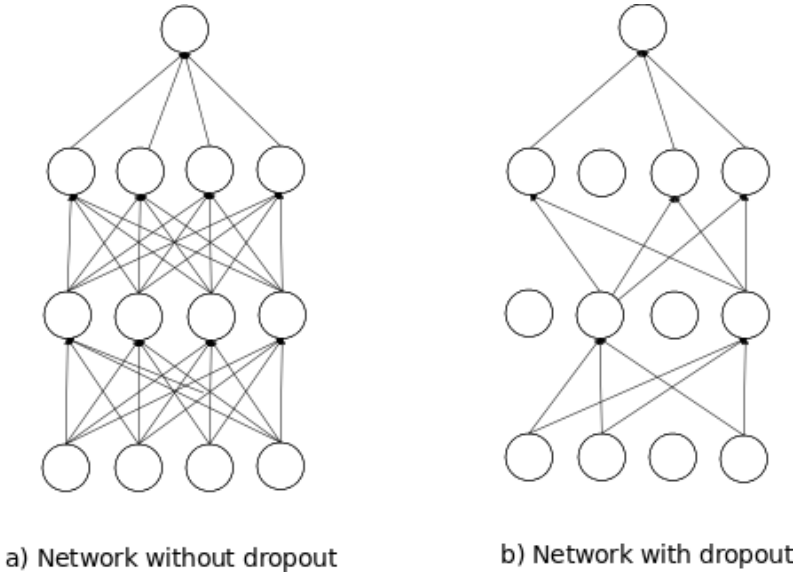


Figure 3.6: **Dropout thinned neural network**

and we normalize the weights when a weight update violates this constraint. This forces the network to learn weights that have reasonable values and thus generalize better. Because we do not penalize the weight update it is possible to train the network with a reasonable learning rate without under-fitting.

In most of our experiments we constrained the norm of the weights to to have a maximum value of 10. The weight constraints work especially well when combined with dropout [9].

Early Stopping

One of the simplest yet most effective ways to address over-fitting is to use early stopping along with a validation set. The validation set is not used for training but instead for validating the performance of the model on unseen data during the training and choosing the hyper-parameters that better optimize the network.

We monitor the validation at the end of every epoch. At the beginning of the training both the training and the validation loss will steadily decrease. When we observe that the validation loss stops decreasing for a number of epochs while the training loss continues to decrease this usually means that the network starts over-fitting the training set. At this point we stop the training and we keep the set of weights that performs best on the validation set. Early stopping is an effective way to prevent over-fitting and it has the advantage of adding no overhead to the training process.

Gaussian Noise

Adding Gaussian noise to the inputs of the network during training works as a strong regularizer for the network. It can artificially increase the size

of the training set with new examples that are similar but not identical to each other. This forces the network to try and learn to denoise the inputs by learning useful patterns.

The added noise, like most regularization techniques, tends to decrease performance on the training set, but increase it on the validation and test set where no input noise is added. The effect of the noise is controlled by the variance of the noise. The variance is a hyper-parameter that is data set specific and needs to be determined using the validation set. Larger values of noise have a stronger effect on the inputs, but may halve the networks ability to learn. In our experiments we used the validation set in order to determine the best possible value for the Gaussian noise. In some of our models we used noise with variance between 0.5 and 0.6 the variance of the original data, while for others we did not use any noise.

Chapter 4

Long Short-Term Memory

In the previous chapter we discussed how recurrent neural networks are good at modeling time series and mapping sequences of inputs into outputs. Unfortunately standard RNNs are pretty bad at remembering things for a long time and thus their ability to model data sequences is limited to very short sequences. The reason for that is the fact that the influence of a given input of a hidden layer can either vanish or explode when it is propagated many time steps in the network. In practice this problem, that is often referred to as the "vanishing gradients" problem [16][3], makes it difficult for standard RNNs to deal with delays of more than a dozen time steps between the input and the target.

Many methods have been proposed that deal with the "vanishing gradients" problem. Some of these methods include non-gradient training [3], while others propose careful initialization of the hidden weights in order to create a pool of loosely coupled oscillators that are able to "echo" inputs over long time delays [17][25].

By far the most effective method so far is the Long Short-Term Memory (LSTM) architecture [16]. In this chapter we review the background material of LSTM architectures and training, which is the main RNN architecture that we are using throughout this thesis. Section 4.1 introduces the basic LSTM unit and how this architecture overcomes the "vanishing gradient" problem. In section 4.2 we will introduce the bidirectional LSTM (BLSTM) architecture which further improves the LSTM architecture. Finally in section 4.3 we present the residual bidirectional LSTM block which is the basic building block we are using in all of our networks presented in this thesis.

4.1 LSTM architecture

The LSTM architecture consists of layers of recurrent memory blocks called LSTM units. The basic LSTM unit contains one or more linear, self-connected memory cells and three multiplicative units, the input, output and forget gates, that provide read, write and reset operations for the cells.

The memory cells have the ability to remember information for a long time and they are controlled by binary control signals. This allows LSTM networks

to overcome the "vanishing gradient" problem. As long as the input gate remains closed the memory cell continues to remember its current value and the activation cannot be overwritten by current inputs. When the output gate opens, the network can access the value that was stored inside the memory cell, even after a very long time.

Figure 4.1 shows a typical LSTM unit with one linear, self-connected unit called the "error carousel". The LSTM networks are constructed just like simple RNNs with LSTM blocks instead of recurrent units. The LSTM layers can be combined with other types of layers such as convolutional and dense layers. An LSTM network can be effectively trained with back-propagation through time as described in chapter 3.

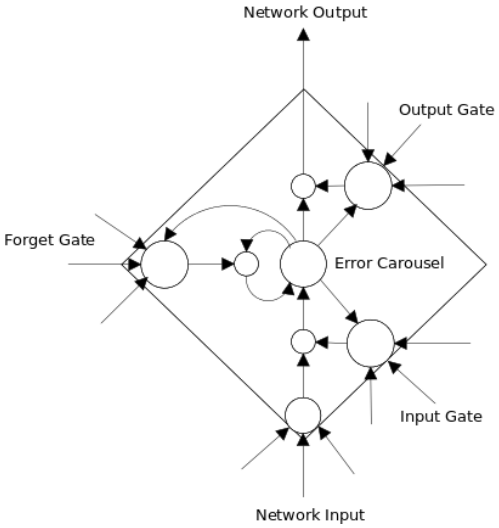


Figure 4.1: **Typical LSTM unit**

Ever since LSTM units were introduced, back in 1997, they have been successfully applied to various problems from speech recognition [12], protein structure prediction [29] and sequence to sequence learning[28].

Currently LSTM architectures are widely used for various machine learning tasks where there is a need to model long range contextual dependencies. On the other hand, if a data sequence is preprocessed and segmented into a number of shorter sequences, the use of LSTM architectures becomes redundant. When dealing with the problem of gesture recognition, if a video sequence is preprocessed and segmented into a series of distinct sequences, then it might be possible to use simple RNNs in order to recognize these short sequences. In this thesis though we are proposing an end-to-end approach where we do not pre-process or segment our sequences. As a result the proposed model needs to be able to deal with very long sequences of data and this makes the use of an LSTM architecture necessary.

4.1.1 LSTM equations

Here we provide the equations for the forward pass and the gradient calculation of an LSTM hidden layer. The exact error derivatives with respect to the hidden weights can be efficiently calculated with back-propagation through time (BPTT) [13]. As previously, $w_{i,j}$ is the weight of the connection from unit i to unit j. The input to unit j at time t is denoted α_j^t and the value of the same unit after applying the activation function z_j^t. The equations presented here refer to a simple memory block. The subscripts ι, ϕ and o refer to the input gate, forget gate and output gate respectively. With s_c^t we denote the state of cell c at time t. With f, g and h we denote the activations of the gates, input and output functions respectively. Also I is the number of inputs, H is the number of cells in the hidden layer and K is the number of outputs.

Forward Pass

Input Gates

$$\alpha_\iota^t = \sum_\iota w_{i,\iota} x_i^t + \sum_h w_{h,\iota} z_h^{t-1} + \sum_c w_{c,\iota} s_c^{t-1} \tag{4.1}$$

$$z_\iota^t = f(\alpha_\iota^t) \tag{4.2}$$

Forget Gates

$$\alpha_\phi^t = \sum_\phi w_{i,\phi} x_i^t + \sum_h w_{h,\phi} z_h^{t-1} + \sum_c w_{c,\phi} s_c^{t-1} \tag{4.3}$$

$$z_\phi^t = f(\alpha_\phi^t) \tag{4.4}$$

Cells

$$\alpha_c^t = \sum_c w_{i,c} x_i^t + \sum_h w_{h,c} z_h^{t-1} \tag{4.5}$$

$$s_c^t = z_\phi^t s_c^{t-1} + z_\iota^t g(a_c^t) \tag{4.6}$$

Output Gates

$$\alpha_o^t = \sum_o w_{i,o} x_i^t + \sum_h w_{h,o} z_h^{t-1} + \sum_c w_{c,o} s_c^t \tag{4.7}$$

$$z_o^t = f(\alpha_o^t) \tag{4.8}$$

Backward Pass

$$\epsilon_c^t = \frac{\partial C}{\partial z_c^t} \tag{4.9}$$

$$\epsilon_s^t = \frac{\partial C}{\partial z_s^t} \tag{4.10}$$

Cell Outputs

$$\epsilon_c^t = \sum_k w_{c,k} \delta_k^t + \sum_h w_{c,h} \delta_h^{t+1} \tag{4.11}$$

Forget Gates

$$\delta_o^t = f'(\alpha_o^t \sum_c h(s_c^t)\epsilon_c^t \tag{4.12}$$

States

$$\epsilon_s^t = z_o^t h'(s_c^t)\epsilon_c^t + z_\phi^{t+1}\epsilon_s^{t+1} + w_{c,\iota}\delta_\iota^{t+1} + w_{c,\phi}\delta_\phi^{t+1} + w_{c,o}\delta_o^t \tag{4.13}$$

Cells

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

Forget Gates

$$\delta_\phi^t = f'(\alpha_\phi^t) \sum_c s_c^{t-1}\epsilon_s^t \tag{4.15}$$

Input Gates

$$\delta_\iota^t = f'(\alpha_\iota^t) \sum_c g(\alpha_c^t)\epsilon_s^t \tag{4.16}$$

4.2 Bidirectional LSTM

In order to enhance the performance of simple LSTMs, the bidirectional LSTM (BLSTM) was proposed [13]. Much like the bidirectional RNN, the bidirectional LSTM layer consists of LSTM units that have the ability to process the data sequence both forward and backwards. The bidirectional layer assumes that the input data can be divided into finite segments and performs a forward and a backward pass before producing an output. This process introduces a small delay between the input event and the output produced by the network and seems to violate causality, but in fact human listeners do exactly the same thing. Words that mean nothing at the first place start making sense when combined with future context.

In a bidirectional network the forward and the backward pass are combined in order to produce results. As a result the complete network is able to make more accurate predictions. For example the forward sub-network is usually more accurate than the backward one but there are some cases where the backward sub-net corrects mistakes made by the forward one. Also because the weights in both directions are tied together, the bidirectional network is better regularized than a unidirectional one and can be effectively trained with a lot less data. BLSTMs consistently outperform LSTMs on sequence classification tasks and are less prone to over-fitting the training set.

4.3 Residual Connections for the Enhancemet of BLSTMs

Deeper neural networks are more powerful than shallow ones. Increasing the number of hidden layers leads to networks with more complicated dynamics that can learn complicated features. Nevertheless, increasing the depth of a

neural network is not as easy as stacking many layers. Very deep feed forward networks start exhibiting "vanishing gradients". This makes the learning procedure of the network very difficult from the beginning of the training. With RNNs the problem only becomes worse due to the nature of recurrent networks. As mentioned earlier, LSTM networks deal with the "vanishing gradient" problem by remembering information over long time lags but if the architecture becomes deeper, LSTM networks also have difficulty converging.

Residual learning [14] was first proposed to deal with the "vanishing gradients" problem on very deep convolutional networks. The deep residual framework introduced shortcut connections skipping one or more stacked layers. Residual connections effectively deal with the "degradation" problem and allow very deep networks to converge. The shortcut connections simply perform identity mapping, and their outputs are added to the outputs of the stacked layers. Identity connections are not counted as trainable parameters and do not increase the computational complexity of the network. The network can still be trained with back-propagation using the same optimizers described in the previous chapter. Figure 4.2 shows a simple feed-forward residual block.

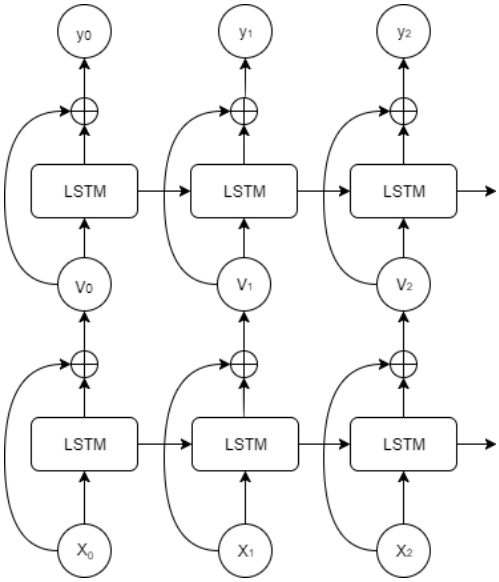


Figure 4.2: **Residual block**

In this thesis we effectively combine BLSTM layers with residual connections. The use of residual BLSTM blocks allowed us to effectively train deep BLSTM networks that have a greater capacity than shallow ones without running into degradation problems.

Chapter 5

Connectionist Temporal Classification

The recurrent neural networks that we described in the previous chapters have one major weakness. In order to train them to perform sequence recognition and classification we need to provide them with sequence, label pairs. To do this we need to pre-process our input sequences and segment them into finite sequences that include only one class instance each. In the task of gesture recognition this means that we need a pre-processing step where all gesture sequences are carefully segmented into parts where only one gesture is performed. Of course this task is not trivial and requires a lot of effort in order to correctly pre-segment all the input data. If the segmentation of the data is not correct this can cause problems to the whole recognition process. In addition, the preprocessing step increases the overall complexity of the model at run time. For all these reasons we decided that it is preferable to explore a different end-to-end approach that does not require pre-segmented data and allows us to train our models using sequences of labels instead.

Connectionist temporal classification (CTC) [11] is basically an output layer that allows the neural network to perform temporal classification. RNNs with CTC are able to learn both the alignment and the recognition task at once while being trained with sequences of labels. We have found that BLSTM networks with CTC are able to outperform models based on HMMs on the task of temporal classification of gesture sequences.

Section 5.1 introduces the basic concepts behind connectionist temporal classification. Section 5.2 briefly describes the forward-backward algorithm which is an efficient way of computing the conditional probabilities of label sequences. In section 5.3 we describe the CTC cost function and in section 5.4 we present a method for decoding CTC outputs.

5.1 Temporal Classification

As it was mentioned earlier, neural networks trained to minimize the standard objective functions require data, target pairs for the training. In other words we require separate targets for every segment in the input sequence. This

creates a couple of problems. First of all, the input sequence needs to be pre-segmented in order to provide the targets. Second, the network only sees the one segment when performing classification and as a result the long term aspects of the sequence must be modeled externally. In order to predict a sequence of labells, a number of pre-processing and post-processing steps is required.

Motivated by these problems, in this thesis we propose a different method that can be used for end-to-end temporal classification. Connectionist temporal classification (CTC) allows the network to make predictions at any point in the input sequence, as long as the overall sequence predicted is correct. This way the network itself learns the alignment of the labels and removes the need for pre-segmented data. Moreover, CTC learns the total output probability of the complete sequence thus eliminating the need for external post-processing.

In the early stages of our work in this thesis we experimented with frame-wise classification neural networks. In order to perform frame-wise classification we pre-segmented the input data with the help of an activity detector and then trained our model to classify each segment separately. As a result the network was unable to discriminate between real gesture instances and out-of-vocabulary instances and thus resulted in a lot of "false positives" and poor accuracy. In order to deal with this problem we would need to explicitly model out-of-vocabulary instances along with a post-processing step to combine the predictions into sequences. On the other hand CTC just solves these problems and makes it easier to implement a well-performing model. A CTC network outputs a series of high probability spikes at any time in a given sequence it recognizes a gesture instance. As a result the rate of false positives is much lower in a CTC network than a frame-wise classification network. Figure 5.1 shows the difference between frame-wise classification and CTC for the same speech signal.

5.1.1 Mapping outputs to labels

In practice for a sequence labeling task where we have to recognize N classes, CTC consists of a softmax layer that has $N + 1$ units. The first N units output the probability of each class while the last unit is responsible for the probability of the "blank" label. In more detail the activation y_l^t of unit l at time step t is the probability of observing class l at time t given an input sequence of length T. If we put together all of these probabilities we get a probability distribution over all possible sequences of length T. Then the probability of observing a particular sequence π given an input sequence x of length T and a training set S is given by the following equation.

$$p(\pi|x, S) = \prod_{t=1}^{T} y_{\pi_t}^t \tag{5.1}$$

The elements π are also referred to as paths which are then mapped into a set of possible label sequences.

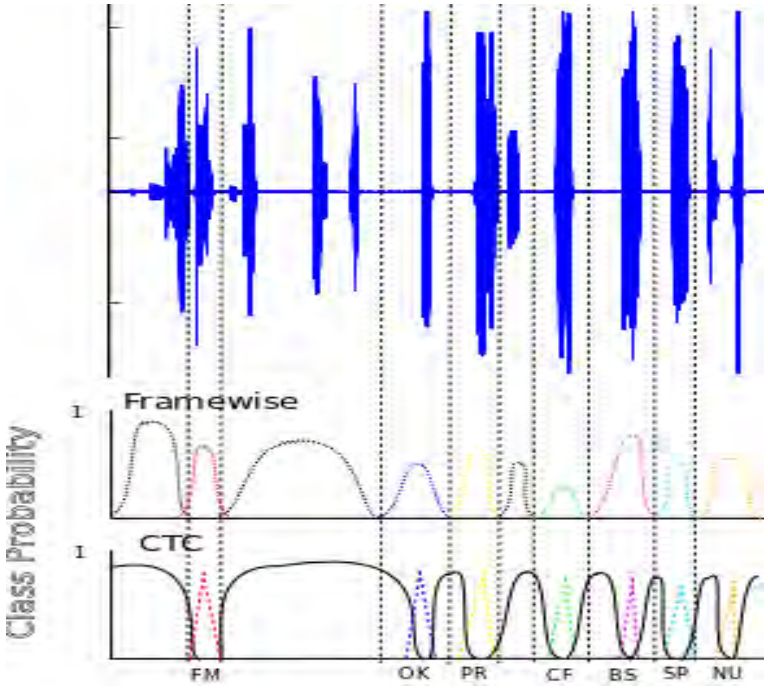


Figure 5.1: **Frame-wise classification and CTC applied to a speech signal**

Here we need to stress the importance of the blank label. Without the blank label the network would keep predicting the same label until the next appeared which is generally not desirable in most situations. Most of the time the network outputs blank labels with short "spikes" of high probability every time a certain class is recognized. This way the network outputs a specific class only if it is really confident for this specific class. As a result CTC networks are more prone to predicting a lot of "false negatives" than "false positives" in contrast to frame-wise networks.

5.2 The Forward-Backward Algorithm

In order to efficiently calculate the conditional probabilities $p(\pi|x)$ of different label sequences we are going to use dynamic programming. The CTC forward-backward algorithm is similar to the forward-backward algorithm for HMMs [23]. The idea behind this algorithm is that in order to compute the sum over paths that correspond to a labeling, we can iteratively compute and sum over paths corresponding to prefixes of that labeling.

For some labeling l, we will define a forward variable $\alpha_t(s)$ which is the sum of all paths with prefixes of length t that are mapped into the length $s/2$ prefix of l.

$$\alpha_t(s) = \sum_{\pi} \prod_{t'=1}^{t} y^{t'}_{\pi_{t'}} \tag{5.2}$$

Now the probability of l can be expressed as the sum of the forward variables with and without the final blank at time T.

$$p(l|x) = \alpha_T(|l'|) + \alpha_T(|l'| - 1) \tag{5.3}$$

All paths start with either the first symbol in l or a blank. We can then get the following initial rules.

$$\alpha_l(1) = y_b^l \tag{5.4}$$

$$\alpha_l(2) = y_{l_1}^l \tag{5.5}$$

$$\alpha_l(s) = 0, \quad \forall s > 2 \tag{5.6}$$

Then we can apply the following recursion rules to find the other forward variables.

$$\alpha_t(s) = y_{l'_s}^t \begin{cases} \sum_{i=s-1}^{s} \alpha_{t-1}(i) & \text{if } l'_s = b \text{ or } l'_{s-2} = l'_s \\ \sum_{i=s-1}^{s} \alpha_{t-1}(i) & \text{otherwise} \end{cases} \tag{5.7}$$

where

$$\alpha_t(s) = 0 \quad \forall s < |l'| - 2(T - t) - 1 \tag{5.8}$$

$$\alpha_t(0) = 0 \quad \forall t \tag{5.9}$$

We can also define a backward variable $\beta_t(s)$ as the summed probability of all paths with suffixes that start at t can be mapped into the suffix of l starting at label $s/2$.

$$\beta_t(s) = \sum_{\pi} \prod_{t'=t+1}^{t} y_{\pi_{t'}}^{t'} \tag{5.10}$$

Now we can get the following rules for initialization and recursion of the backward variables.

$$\beta_T(|1'|) = 1 \tag{5.11}$$

$$\beta_T(|l'| - 1) = 1 \tag{5.12}$$

$$\beta_T(s) = 0, \quad \forall s < |l'| - 1 \tag{5.13}$$

$$\beta_t(s) = \begin{cases} \sum_{i=s}^{s+1} \beta_{t+1}(i) y_{l'_i}^t & \text{if } l'_s = b \text{ or } l'_{s+2} = l'_s \\ \sum_{i=s}^{s+2} \beta_{t+1}(i) y_{l'_i}^t & \text{otherwise} \end{cases} \tag{5.14}$$

where

$$\beta_t(s) = 0 \quad \forall s > 2t \tag{5.15}$$

$$\beta_t(|l'| + 1) = 0 \quad \forall t \tag{5.16}$$

5.3 The CTC Cost Function

In the previous section we described how CTC recurrent neural networks perform temporal classification of data sequences. In order to train the neural network to perform some task we need a cost function, or objective function as it is usually referred to, that we will train it to minimize. Once we have that cost function we are ready to train the neural network using back propagation and one of the optimizers described previously (see section 3.3.1).

Just like other neural network cost functions, like cross entropy, the CTC cost function is defined as the negative log probability of getting all the labels correctly. The cost function C for the entire training set is given by the following equation.

$$C = -\ln\Big(\prod_{(x,z)\in S} p(z|x)\Big) = -\sum_{(x,z)\in S} \ln p(z|x) \tag{5.17}$$

Now if we differentiate we will get the derivative of the cost function with respect to the output y_k^t for some training example (x, z).

$$\frac{\partial C}{\partial y_k^t} = -\frac{\partial \ln p(x|z)}{\partial y_k^t} = -\frac{1}{p(z|x)}\frac{\partial p(z|x)}{\partial y_k^t} \tag{5.18}$$

In order to calculate this derivative we can use the forward-backward algorithm described in the previous section. The summed probability of all paths mapped into z that go through symbol s at time t is the product of the forward and backward variables defined earlier. If we replace z with l in equations 5.2 and 5.10 we get the following

$$\alpha_t(s)\beta_t(s) = \sum_{\pi}\prod_{t=1}^{T} y_{\pi_t}^t \tag{5.19}$$

If we take this one step further and substitute from equation 5.1 we get

$$\alpha_t(s)\beta_t(s) = \sum_{\pi} p(\pi|x) \tag{5.20}$$

Finally if for any t we sum over all s we get

$$p(z|x) = \sum_{s=1}^{|z'|} \alpha_t(s)\beta_t(s) \tag{5.21}$$

This equation is obviously differentiable and if we differentiate with respect to the output, for the paths that go through label k at time t to get

$$\frac{\partial p(z|x)}{\partial y_k^t} = \frac{1}{y_k^t}\sum_{s\in lab(z,k)} \alpha_t(s)\beta_t(s) \tag{5.22}$$

At this point we can apply the chain rule in order to get the derivatives with respect to the unit activation z_k^t for unit k before the CTC output function is applied.

$$\frac{\partial C}{\partial z_k^t} = -\sum_{k'} \frac{\partial C}{\partial y_{k'}^t} \frac{\partial y_{k'}^t}{\partial z_k^t} \tag{5.23}$$

It should be noted that k' ranges over all the output units including the unit corresponding to the blank label. At this point we can recall that y_k^t is actually the output of a softmax unit that has the following nice, simple derivative where $\delta kk'$ is the derivative of the hidden activation.

$$\frac{\partial y_{k'}^t}{\partial z_k^t} = y_{k'}^t \delta kk' - y_{k'}^t y_k^t \tag{5.24}$$

This derivative along with equation 5.22 can then be substituted into equation 5.23

$$\frac{\partial C}{\partial z_k^t} = y_k^t - \frac{1}{y_k^t} \sum_{s \in lab(z,k)} \alpha_t(s)\beta_t(s) \tag{5.25}$$

This is the error signal received by the network during training. This signal can then be propagated back through time, each time applying the chain rule, to calculate the derivatives with respect to the hidden weights for every hidden layer.

5.4 Decoding the CTC Outputs

In every machine learning task our ultimate goal is to use the trained model to make predictions on previously unseen data. We want to use a trained CTC network to classify an unseen input sequence into a sequence of class labels. In order to do this we want to choose the labeling $l*$ that maximizes the posterior probability $p(l|x)$ of observing labeling l given input sequence x.

$$l* = \underset{l}{\operatorname{argmax}}\, p(l|x) \tag{5.26}$$

In terms of HMMs the task of finding the most probable labeling is called decoding. Here we are going to present an approximate method that can be used for decoding the outputs of a CTC network.

5.4.1 Best path decoding

Best path decoding [11] is a straight-forward method and it assumes that the most probable labeling also corresponds to the most probable path. In order to compute the best path decoding we take the most active (probable) outputs at every time step and concatenate them into a sequence of labels.

$$l = \mathcal{B}(\pi*)$$
$$\text{where } \pi* = \underset{\pi}{\operatorname{argmax}}\, p(\pi|x)$$

Best path decoding is very fast to compute since it is just the concatenation of the most probable classes. In some cases though it can lead to errors since best path decoding struggles in cases where a label is weakly predicted for several consecutive time-steps. The following figure illustrates this kind of situation.

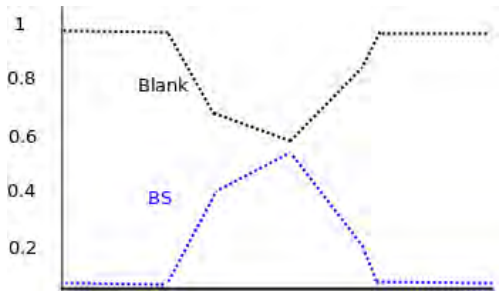


Figure 5.2: **Best path decoding error.**

The best path decoding outputs “blank” because this is the single most probable class. However if we combine the paths corresponding to the labeling ”BS” the probability is actually higher.

When running our experiments for this thesis we found that despite its weaknesses, best path decoding gives us very good results for the task of gesture sequence recognition. As a result it was preferred over other, more sophisticated methods, because of its simplicity and fast implementation.

Chapter 6

Multimodal Gesture Recognition

The term gesture is used for any non verbal action that is performed in order to communicate a message or express a feeling. Gestures can be static, dynamic or a combination of both. They are usually performed using the hands and arms but can also be performed with the head, the face and the rest of the body. Moreover gestures are often language and culture-specific. Some examples of gestures are the cultural and anthropological signs that usually accompany speech and sign language which is generally a visual language.

Gesture recognition unlike speech recognition, is not performed using a single sensory input but rather a combination of different inputs coming from different sensors. In fact human communication and interaction takes advantage of multiple sensory inputs. We are extremely good at receiving inputs from many different sensors and using all of them in order to perceive the world around us. For example when watching a video, a sound is perceived as coming from the speakers lips. In addition our perception may be affected by whether the lips of the speaker are visible or not. At a higher level, gestures that accompany speech usually affect the meaning of the spoken words.

A lot of different gesture recognition tasks have been studied over the previous years, including sign language recognition [4] and human computer interaction [33]. Also a variety of methods have been proposed over the years using both HMMs [22] and deep learning [21]. Based on the formulation of the task at hand, different methods can be best suited for different tasks and most of them have their advantages and disadvantages.

As mentioned previously the purpose of this thesis is to propose a methodology that uses deep learning in order to perform temporal gesture recognition. This methodology incorporates multiple input streams in order to produce an output sequence of labels. This is done by breaking down the problem into multiple uni-modal problems and solving them independently using RNNs. The first RNN is based on speech input while the second one uses skeletal input from a KINECT camera. Once we have well-performing models for each modality we combine the RNNs using a higher level RNN that outputs the final label sequence. At this point it should be noted that although we used

these two modalities in our model, the proposed methodology can easily be extended to combine more and different modalities like RGB and depth video.

In this chapter we present in detail the proposed model. In section 6.1 we described the outlines of the method we used and in section 6.2 there is a presentation of the ChaLearn dataset that was used in our experiments along with all the challenges we had to overcome working with this dataset. In sections 6.3 and 6.4 we go into the details of the different modules that were used for the speech and skeletal modalities respectively. Finally section 6.5 goes through the multi-modal fusion method that was used to integrate the individual modules.

6.1 Temporal Classification of Gesture Sequences

In this thesis our task was to build a model that classifies hand gestures that are performed in a continuous input stream and are accompanied by spoken words. This kind of task is similar to keyword spotting as the gestures we want to recognize come from a small vocabulary and are usually mixed with other out of vocabulary ones.

We decided that in order to build a successful gesture recognition system we would have to incorporate multiple modalities. In this case we chose to use an audio stream along with a skeletal stream. We were confident that an audio keyword spotter would do a very good job at classifying the different keywords but these kinds of systems tend to be very sensitive to background noise and out of vocabulary words. On the other hand we expected that a classifier that uses only skeletal features would not be able to accurately classify all the different gestures but would be less sensitive to distractions. The basic idea was that when these two modalities were forced to work together, they would be complementary to each other and the accuracy of the model would improve.

A number of other methods proposed for the task of gesture recognition [22][32] would segment an input sequence using some activity detector and then train a model that maps each segment into a gesture. This approach is similar to the frame-wise classification task that we described in previous chapters and as a result it has the same disadvantages.

On the other hand our method is a temporal classification model that takes as input whole sequences of input data and maps them into a sequence of labels. The main idea was that a single neural network with CTC output and enough trainable parameters should be able to learn to align and classify the input sequences through training.

The basic framework consists of two RNN sub-networks, one for each modality, that are able to learn temporal representations of the two input sequences. Then these representations are concatenated and propagated through a higher level RNN, with CTC output, that combines them and predicts the most probable labeling. Figure 6.1 illustrates the basic framework.

There is one problem that occurs when we try to implement a multi-modal architecture like this. In most cases the modalities combined will not have the

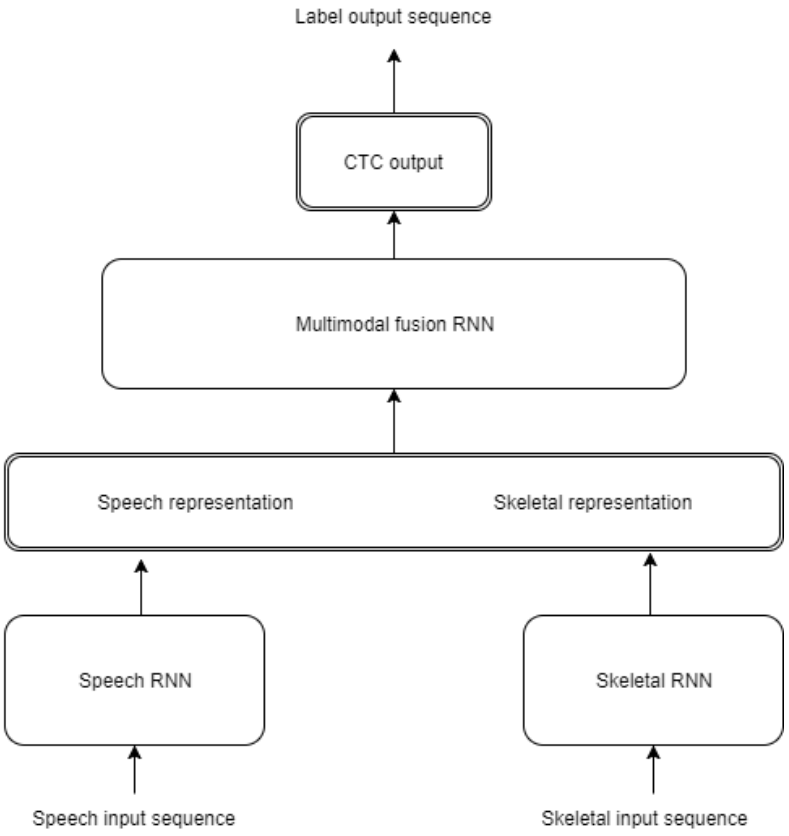


Figure 6.1: **The basic framework**

same alignment. For example a simple hand gesture might be accompanied by a small phrase of speech. As a result the two input sequences will have a much different alignment. This makes it very difficult for a single CTC network to figure out what is the suitable alignment for both inputs. There could be an argument here that a neural network with enough parameters and enough training data could actually figure out, after a lot of training, how to align both sequences. However, in practice we found out that this is not an efficient way to train such a model as it would take a very long time for it to converge and it is very likely to get stuck into local optima unable to progress any further. There is one simple trick that we used in order to overcome this obstacle.

Instead of training one big recurrent network at once, we trained each of the two individual networks separately. We took each of the two sub-networks and using a CTC output we trained each one of them to perform temporal gesture classification. Once each unimodal network has converged we removed the CTC output layers, concatenated their outputs and used them as input for the high level RNN which was then trained using CTC output.

(a) RGB (b) Depth (c) User Mask (d) Skeletal Model

Figure 6.2: **The different modalities of the ChaLearn dataset.**

6.2 The ChaLearn Dataset

At this point we will describe the dataset that we used for our experiments. The ChaLearn Multi-modal Gesture Recognition challenge was organized in 2013. This competition focused on spotting gestures drawn from a certain gesture vocabulary, based on multiple gesture instances performed by different people.

6.2.1 The data

The dataset is split into gesture sequences. In each sequence a single user is recorded in front of a KINECT camera, performing natural communicative gestures and speaking in fluent Italian. Each sequence includes audio, skeletal model, user mask, RGB, and depth streams. In figure 6.2 the different modalities of the dataset are shown.

The skeletal model includes the coordinates of 20 joints of the human body as captured by the KINECT camera. These joints accurately describe the users' pose and movements. The joints can be seen in Figure 6.3

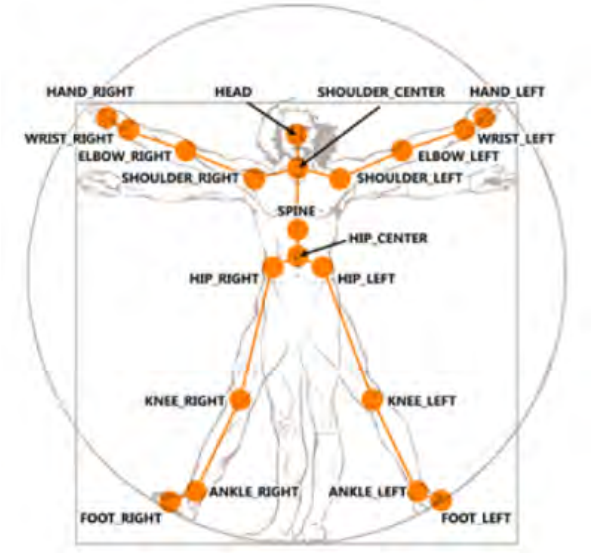


Figure 6.3: **The skeletal joints captured by KINECT** [8]

A total number of 27 users (male and female) perform a total number of 13.858 gestures. The data are split into three sets. The training set contains 393 sequences with a total of 7.754 gestures. The validation set contains 287 sequences with a total of 2.742 gestures and the test set 276 with a total of 2.742 gestures. Each sequence lasts between 1 and 2 minutes and contains between 8 and 20 gesture samples.

When training our models the training signals we used were simple sequences of labels for the gestures performed in each video sequence without any additional information about the specific timing of each event.

6.2.2 Gesture vocabulary

The vocabulary of gestures included 20 Italian cultural gestures each of them corresponding to a specific word or phrase in Italian. While performing a gesture with his or her body movement, the performer also speaks out the corresponding Italian word or phrase. The table bellow shows the 20 gestures along with the corresponding phrases. Figure 6.4 illustrates the different gestures.

Table 6.1: **The different class labels.** We also present the spoken phrases that correspond to these classes.

VA	Vattene
VQ	Vieni qui
PF	Perfetto
FU	E' un furbo
CP	Che due palle
CV	Che vuoi
DC	Vanno d'accordo
SP	Sei pazzo
CM	Cos hai combinato
FN	Non me ne frega niente
OK	Ok
CF	Cosa ti farei
BS	Basta
PR	Le vuoi prendere
NU	Non ce ne piu
FM	Ho fame
TT	Tanto tempo fa
BN	Buonissimo
MC	Si sono messi d'accordo
ST	Sono stufo

6.2.3 The challenges of the task

The ChaLearn taks is especially challenging for multiple reasons. The gestures within each sequence are performed continuously and without a resting pose.

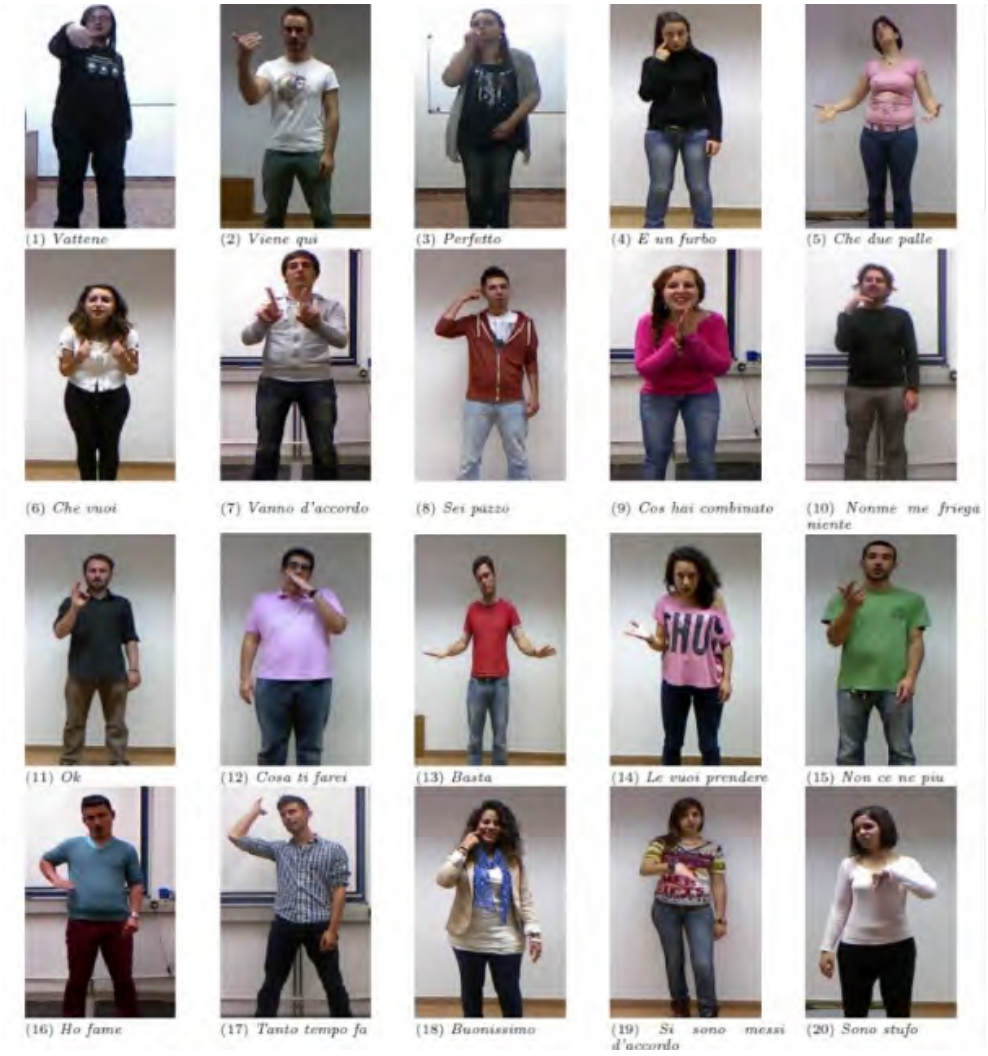


Figure 6.4: **Examples of all gesture classes**

This makes it very difficult to decide when one gesture ends and the next begins. Also many different gesture instances are present in each sequence and there are distracter gestures out of the vocabulary present in terms of both gesture and audio.

There are high inter and intra-class variabilities of gestures in terms of both gesture and audio in the dataset. Moreover the different sequences are recorded in different backgrounds while the users may wear different clothes and have different skin color. There are also variations in the lighting, temperature and resolution of the recorded sequences. To make the task even more challenging some body parts of the users may be occluded and users may speak in different Italian dialects.

6.3 The Speech Model

Building the speech component of the gesture recognition model was very similar to building a keyword spotter network. It is essentially an RNN that takes as input an acoustic signal and outputs the correct label sequence. In most speech recognition and NLP tasks phoneme level models are used but we decided that in our task that wouldn't be necessary. A simple word level model would be sufficient for our task because we only have to recognize keywords from a very small vocabulary.

6.3.1 Feature extraction

The first step in building any machine learning model is feature extraction. Proper input features are key to building a good model. In our speech model we used a standard feature extraction method in speech recognition [34]. The input acoustic data was characterized as a sequence of vectors of 39 coefficients. These coefficients consist of 13 Mel-frequency cepstral coefficients (MFCC) plus energy and their first and second derivatives. The coefficients were computed every 50ms over 25ms long windows. Then a Hamming window was applied, a Mel-frequency filter bank of 26 channels was computed and, finally, the MFCC coefficients were calculated. The sampling period we used is five times longer than the normal period used for frame-wise classification tasks for two reasons. The first reason is that CTC makes it possible to use features extracted with much lower sampling frequency. The other reason is that we were ultimately going to combine the speech model with a skeletal model where the inputs were recorded at 20 frames per second so we wanted the two input streams to have approximately the same length.

6.3.2 Network architecture

The speech that accompanies gestures can either be a single word or a small phrase. The words and phrases can be found in many variations and speeds. There can also be small pauses between words. As we mentioned previously (chapter 4) simple RNNs are having difficulties with dependencies over long time lags so it was essential that we used LSTM units in our networks.

The basic speech network consists of two hidden BLSTM layers both containing 500 memory blocks (in each direction). The memory blocks at each layer are fully connected to themselves as well as the previous and next layer. There are also residual connections that make the training of the network easier. The number of hidden units was selected after experimenting with the validation set. The activations of the memory cells are hyperbolic tangents while for the recurrent connections we used hard sigmoids.

On top of the BLSTM layers we have a dense layer with as many hidden units as the number of different words we need to recognize (44 in our case). These units make up the CTC output layer which is essentially one big softmax at each time step. Also the network has 39 input units (one for each MFCC coefficient).

Due to the challenges of the ChaLearn task we needed to build a network that has enough capacity to model all the different variations of the words in our dataset. For that reason our complete network has approximately 8 million trainable parameters. Figure 6.2 illustrates the network architecture.

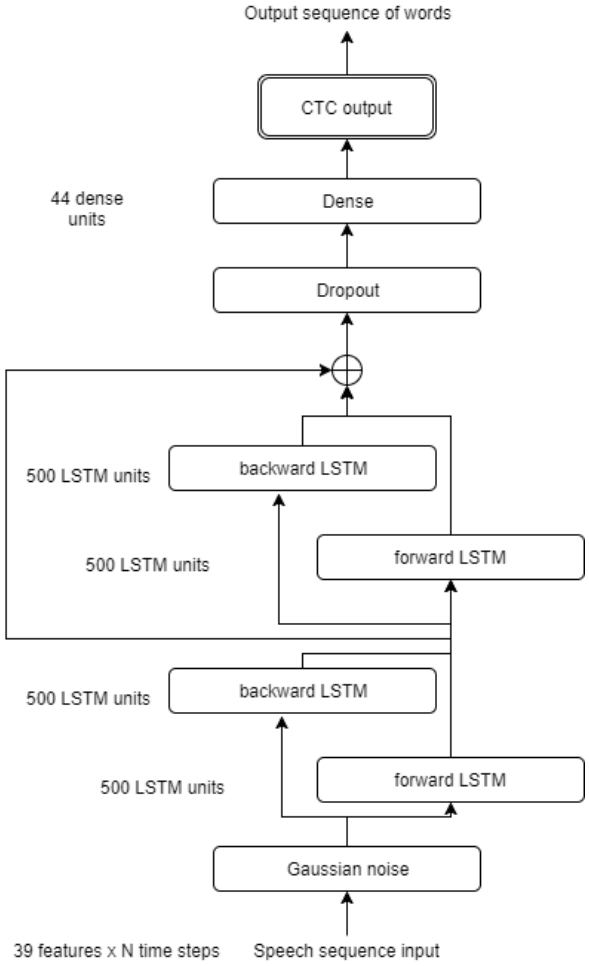


Figure 6.5: **The speech word level BLSTM network**

6.3.3 Regularization

A large neural network with many trainable parameters can very easily learn the sampling error and over-fit the training set. The network that we used is quite large in order to deal with the task at hand but this means that it would very easily over-fit the training set without proper regularization techniques.

In order to address this problem we used a combination of different techniques. We applied dropout on the inputs of the LSTM units in order to corrupt the inputs of each cell. Dropout forces the hidden units in the layer to stop cooperating and learn individually useful features. The dropout value is a hyper parameter that was also chosen after trying different values on the

validation set. We did not apply any dropout in the recurrent connections because we wanted to preserve the signals carried through time. A dropout layer was also added after the second BLSTM layer. This dropout layer corrupts the inputs going into the dense layer.

A large network like the one we are using here needs to be trained with a lot of data in order to generalize properly. The ChaLearn dataset however is not a very large one. A good way to artificially enhance the training set is to add noise to the input data. We applied additive Gaussian noise on the network inputs during training in order to augment the training set. The effect that Gaussian noise has on the training inputs is controlled by the variance of the additive noise which was again decided based o results on the validation set. Like dropout, Gaussian noise is only used during training and is turned off at test time allowing the network to use its full capacity.

The combination of these methods allowed our network to really learn useful features and generalize well on unseen data. Because these techniques are only applied during training, we sometimes observe the strange phenomenon that the validation error is actually smaller than the training error. The specific values of the hyper parameters used during training are provided in the next chapter.

One interesting property of our model is that when training the network we did not use any out of vocabulary examples. The training data included only instances of the gestures we wanted to recognize. When evaluating the model though we used gesture instances mixed up with a lot of out of vocabulary words. Although the network made some insertion errors, we found out that it was successfully ignoring a great deal of out of vocabulary instances without being explicitly trained to do so. Also we found out that the network had learned a lot about phrases and proper syntax just by observing a lot of training examples.

6.4 The Skeletal Model

As we said earlier a neural network that is able to understand speech, although powerful, would have some limitations in the task that we are trying to solve. Using a neural network that is able to understand another modality as well that is complementary to speech would help us deal with such limitations. The biggest problem we had to solve is the many out-of-vocabulary words and gestures that occur along with the gestures and keywords that we want to recognize. In fact the speech model by itself is powerful enough to sufficiently model the different variations of speech in our dataset and predict the correct class in almost 97% of the examples. The biggest weakness it has is that it predicts a large number of false positives along with the correct classes.

The skeletal neural network was designed to help our model deal with that problem. We wanted a skeletal model that is able to understand the basic movements and help our model decide whether a specific event corresponds to a real gesture or an out of vocabulary one. When both modalities agree in their predictions then there is probability that the event is an actual gesture.

When the two models make very different predictions then we most likely have an out of vocabulary event.

6.4.1 Skeletal features

Building a neural network that is able to recognize gestures just by observing the movements of some skeleton joints is by itself a non-trivial task. In contrast to the speech model where there are widely used features that are proven to give very good results, for the skeletal model we had to engineer the features that would help us classify the gestures.

These features needed to be descriptive enough to allow our network to model all the gestures and generalize properly. If we wanted to use the exact position of the joints then the network would very easily learn the sampling error and over-fit the training set. As a result it would generalize very badly to unseen data. Instead we engineered new features that would more accurately model the different gestures but also be invariant enough to allow for good generalization. We wanted the features to be invariant to a number of translations. For example certain gestures might be performed with either hand and at different speeds. Moreover the distances between the joints and the position of the user would differ from user to user and in some cases certain body parts were not visible. All these variations add extra difficulties to the task of modeling skeletal movements.

We focused on features that could fully describe the user's body pose and movements. The distances and angles of the different joints of both hands from the hip center and the shoulder center were measured in order to describe the body pose. In addition to the pose descriptors we wanted to use dynamic features that described the movements of the user. In order to do this we computed the velocity of the hand joints for each frame. The velocity was computed by taking the Euclidean distance between the position of the joint for two consecutive frames. This velocity was given by the following formula. Where x_i and y_i denote the coordinates of joint i while x'_i and y'_i denote the coordinates of the same joint at the next time step.

$$v_i(x, y) = \sqrt{(x'_i - x_i)^2 + (y'_i - y_i)^2}$$

We also approximated the rest position of the hands for each gesture and measured the distance of the hands from the rest position for each frame. The rest position was approximated by averaging the position of the hands for the frames where the user was not moving. We assumed that the hands of the user would usually have low velocity near the rest position.

The skeletal feature vector we used in our experiments consists of 22 features that we found to yield the best performing model. These features are the input for the LSTM network at each time step of the sequence. We knew that the skeletal model by itself would have difficulties when trying to accurately predict some gestures but we found that it was good enough for our method as it would ultimately be combined with the speech model.

6.4.2 Network architecture

The architecture of the skeletal network is quite similar to the one we used for speech. The main difference with this network is that it has much fewer LSTM units per layer which means that it has less trainable parameters and thus smaller learning capacity. When designing the skeletal model we didn't want it to have as many parameters as the speech model which is going to do most of the classification work. Figure 6.6 illustrates the skeletal model. The structure of this network is similar to the speech network but this time we have fewer hidden units per layer and lower input dimensionality.

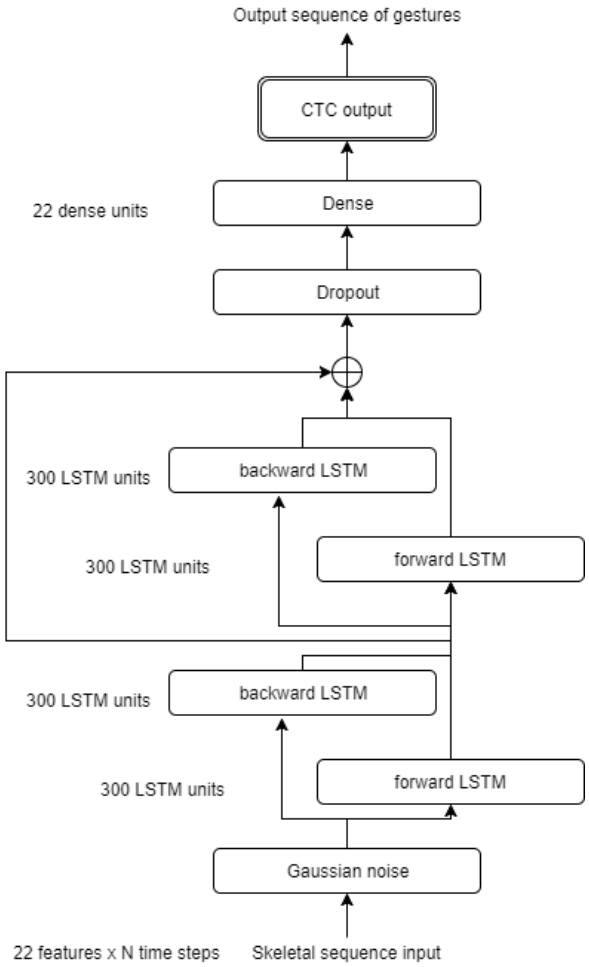


Figure 6.6: **The skeletal BLSTM network**

Each of the two bidirectional LSTM layers has 300 LSTM units in each direction. These units have hyperbolic tangent activations and hard sigmoid recurrent activations. Just like the speech network all BLSTM layers are fully connected to themselves as well as the previous and the next layer and we also have skip connections from the first BLSTM layer to the output of the second BLSTM layer.

On top of the BLSTM layers we again use a fully connected layer with

as many hidden units as the number of different that we have in our dataset (20 for this network). These units make up the CTC output layer which is essentially one big softmax at each time step. The input units for this network are 22 because this is the dimensionality of the skeletal feature vectors we are using.

The total number of trainable parameters for this network is approximately 3 million parameters which is significantly less than the speech model. As we mentioned earlier this network should work as a complementary to the speech network and not as a stand alone model and so it does not require to have as many learnable parameters.

6.4.3 Regularizing the skeletal model

Although this network is significantly smaller than the one used for speech recognition, there is still a need to regularize it in order to prevent it from overfitting. We used dropout in order to prevent the LSTM units from co-adapting too much combined with weight constraints that prevented the weights from becoming too large. We also added Gaussian noise to the inputs of the network in order to artificially augment the training set. The values used for dropout, max-norm constraints and the variance of the Gaussian noise were selected after experimenting with the validation set.

6.5 Multimodal fusion

Once we have trained both unimodal networks we had to find an efficient way to combine the two models and make them work together. Combining two different models in a single neural network that will yield increased performance is not always an easy task. In order for a multi-modal network to have better performance than the unimodal networks we need to make sure that the two individual networks are in some way complementary to each other. A good indicator for this is when the different networks make very different errors when used to predict the test set.

In our case we experimented with both unimodal networks and found out that they exhibit very different behaviors and make different errors. In fact the neural network that was used for the audio modality was having very good classification results but was also outputting a lot of false positives. On the other hand the skeletal network was suffering from a lot of false negatives due to the fact that the skeletal stream cannot adequately model certain gestures from our vocabulary. These are usually the gestures that involve very small hand movements.

It was obvious that these two models could be combined in a way that can boost the overall performance. The main idea was that these two networks could be used as feature extractors that will output a vector representation for each modality. Then these vector representations would be used as input for a higher level LSTM network with CTC output that would combine them in order to make the final temporal classification. We expected that a high

level network with enough capacity would be able to exploit the strengths of both networks and produce improved results.

Although this approach seemed very promising we knew from the beginning that the proposed architecture would have quite a lot of trainable parameters. On the other hand the ChaLearn data set is not a very large one and it was obvious that the training data were not enough in order to train such a large network without severely over-fitting the training set. That is the main reason we decided to train the two networks individually and then combine the converged networks. This way we make sure that each individual network has already learned useful parameters before fusing them into one large network.

There is one last trick that we used in order to efficiently train the combined model. Before training the combined network we froze the weights of the two sub-networks and trained only the parameters of the high level, fusion network.

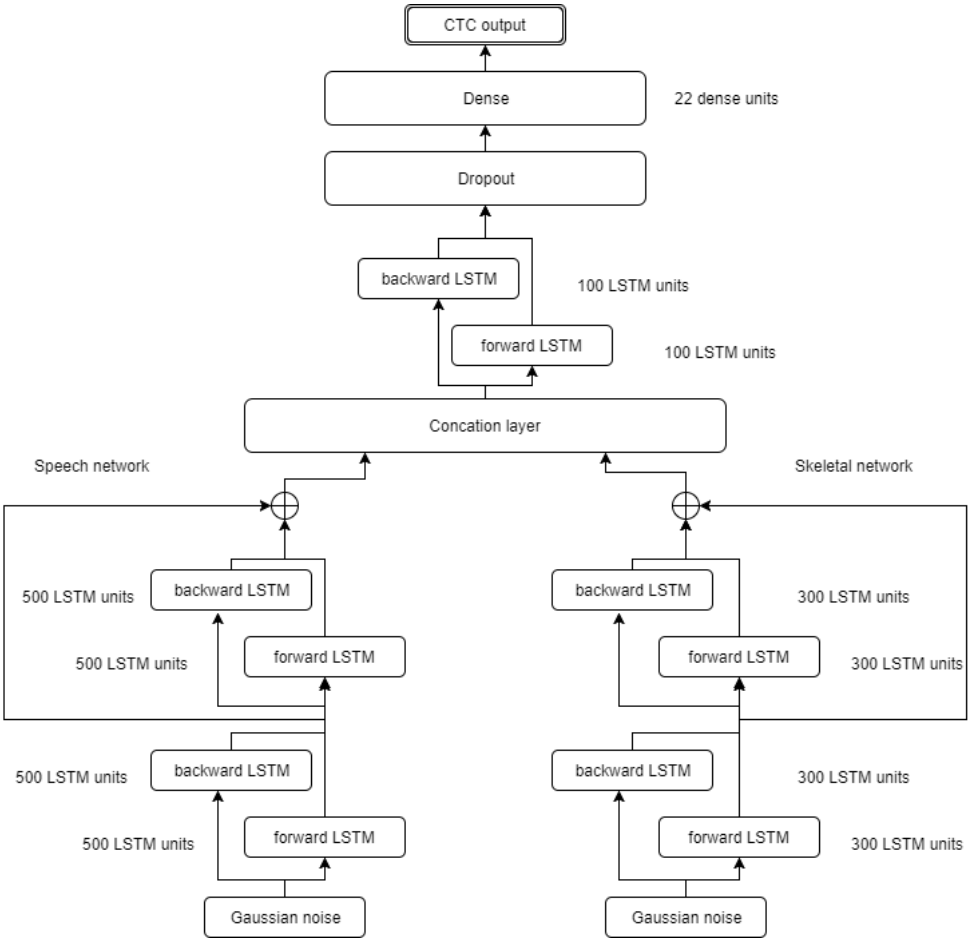


Figure 6.7: **The complete BLSTM network**

6.5.1 Fusion network architecture

The architecture that we used for the fusion network is quite simple. We first removed the CTC output layer from the two trained sub-networks, froze the trainable weights and concatenated the outputs of the top BLSTM layers into a large merge layer. The merge layer had the same sequence length that both sub-networks have. At each time step we get a feature vector which is the concatenation of the output vectors of the two sub-networks.

The merge layer was used as input for another BLSTM layer with 100 LSTM cells in each direction. The activations of these cells were hyperbolic tangents while the recurrent activations were hard sigmoids. This is the layer that is responsible for combining the features extracted from the two sub-networks.

On top of the BLSTM layer we added a dense output layer with a 22-way softmax that implements the CTC output. The complete network has approximately 13 million parameters although only 2 million of them are trainable at this stage. The rest of the parameters were learned at the previous stages and are no longer trainable. Figure 6.7 illustrates the complete network architecture.

6.5.2 Regularization

In order to further regularize the complete model we used the same combination of techniques that were proposed for the two sub-networks. We used dropout at the input of the BLSTM units while leaving the recurrent connections without dropout. We also constrain the BLSTM weights to have a max-norm of 10 in order to stop them from becoming too large. Another dropout layer was added before the final dense layer in order to corrupt the inputs of that layer.

The dropout values and max-norm parameters were chosen after experimenting with the validation set in order to get the best performing model possible. One might notice that we used a combination of regularization methods at all components of the network. This was key in order to get good results because the network architecture we implemented has quite a lot of trainable parameters and the ChaLearn data set is not that large. It is always preferable to get more data if it is possible in order to train large models that are able to generalize well without over-fitting.

Chapter 7

Training and Experiments

In chapter 6 we described in detail the architecture and implementation of the different components of our model. Training our model was a long and difficult process that required many months of fine-tuning. LSTM networks that are trained over long, high dimensional time sequences usually require quite a lot of training and a number of different techniques to converge effectively. In this chapter we are going to present the training process as well as the experimental results.

In section 7.1 we present in detail the hyper parameters of our implementation. These hyper parameters include batch size, optimizers, learning rate and dropout. Section 7.2 describes the training procedure along with all the challenges that had to be overcome. Finally in section 7.3 we provide results from the performance of our model on the ChaLearn Challenge data set.

7.1 Training hyper parameters

When training a neural network one must select values for a number of hyper parameters. The tuning of this hyper parameters usually takes a lot of experimenting through trial and error and requires some degree of experience. All of these hyper parameters are crucial in order to train a successful neural network. In our work we selected the values for these hyper parameters after running a lot of experiments with the training and validation set. This process took quite a lot of time as we had to train three large recurrent networks many times in order to decide which was the best set of hyper parameters.

7.1.1 Sequence length

The first value that we had to select is the length of the input sequences. LSTM networks are capable of processing sequences of different lengths but in our case we decided that it would be much easier if we used sequences of fixed size. The maximum sequence length in our data set was approximately 1800 frames so we set the sequence length to be that long. Sequences that were shorter than the maximum length were padded with zeros while sequences that happened to be longer were truncated to maximum length.

It is obvious that the sequence length greatly affects the complexity of the model. An LSTM network takes a lot of time to process very long sequences which means that our approach would take significantly longer to train on much longer sequences. A different approach would be to keep all sequences at their initial length and just add a termination symbol at the end in order for the network to start predicting values.

7.1.2 Batch size

One other important hyper parameter is the batch size used for the training. Batch size is the number of examples the network needs to process before updating the hidden weights. This value affects the training in two ways. The most obvious effect is that larger batch sizes greatly accelerate training but also require a lot of memory.

One less obvious effect of the batch size has to do with the training algorithm itself. When the error derivatives are computed over large batches we get a better approximation of the error derivative over the full set and thus the weight updates are more efficient. On the other hand computing the error derivatives over small batches can often lead to less accurate weight updates and make convergence very difficult. In our implementation we were forced to use a batch size of two examples because we were limited by the GPU memory.

7.1.3 Optimizer

In chapter 3 we presented the standard method of training neural networks that uses back-propagation to compute the error derivatives and then iteratively updates the hidden weights in order to minimize some loss function. The optimization problem is usually solved in practice using Stochastic Gradient Descent or some more advanced optimization algorithm. The optimization algorithm used is also a training parameter and needs to be carefully selected and parameterized in order to efficiently solve the optimization problem.

Our implementation uses the Adam (adaptive moment estimation) is a method for efficient stochastic optimization that uses the first-order derivatives. The algorithm computes adaptive learning rates for different parameters from estimates of the first and second moments of the gradients. Adam is closely related to AdaGrad [6] and RMSProp and has proved to be very efficient for many optimization problems including recurrent neural networks.

The optimization process is controlled by four parameters. These parameters are namely the learning rate α, the two exponential decay rates for the moment estimates, $\beta 1$ and $\beta 2$ and a fixed bias term ϵ. In our experiments we used the values proposed by the original paper for all parameters except for the learning rate which we fine tuned. The values used for these parameters were $\beta 1 = 0.9$, $\beta 2 = 0.999$ and $\epsilon = 10^{-8}$.

Learning rate

Learning rate is usually one of the most important parameters when training any neural network. It needs to be sufficiently small in order for the optimization to converge but not too small as it would take a very long time until convergence. We found that the optimal value for the learning rate was 10^{-3} for both the unimodal sub-networks and the multi modal network. We did not use any learning rate decay because we did not find it necessary.

Gradient clipping

Although the LSTM networks with hyperbolic tangent activations that we used usually eliminate the exploding gradients problem we found out that we can accelerate the convergence of the models by clipping the gradients to a small value. We clipped all gradients forcing them to have a a maximum value of 0.5 and a minimum value of -0.5. By doing so we avoid very large gradient values that can cause the training loss to fluctuate and slow down the training procedure.

7.1.4 Initialization

When recurrent neural networks were first proposed they were found to be really hard to train because of the vanishing gradients problem. Early studies showed that careful initialization is key to solving this problem. LSTM networks along with advanced optimization algorithms solved this problem and reduced the need for advanced initialization techniques.

Nevertheless recent studies have shown that proper initialization can still be advantageous for LSTM networks. Large LSTM networks where the hidden weights are initialized with reasonable values can be trained much more efficiently and usually outperform randomly initialized networks. We initialized all the hidden weights of our models with a random distribution in the range $[-0.05, 0.05]$. The small initial weights allow the neural networks to converge faster with gradient descent.

7.1.5 Normalized inputs

Input normalization is a well studied method that has proved to be advantageous for any gradient based optimization method including neural networks. The main idea is that by normalizing the input features to have zero mean and unit variance over the training set we help the algorithm converge. 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 [20].

In order to normalize the training set we first calculate the mean

$$m_i = \frac{1}{|S|} \sum_{x \in S} x_i \tag{7.1}$$

and standard deviation

$$\sigma_i = \sqrt{\frac{1}{|S|-1}\sum_{x \in S}(m_i - x_i)^2} \tag{7.2}$$

for every component of the input vector. Then the normalized input vectors x' are computed as follows

$$x'_i = \frac{x_i - m_i}{\sigma_i} \tag{7.3}$$

Here we should note that test and validation sets should be normalized with the mean and standard deviation of the training set.

7.1.6 Regularization

As it was mentioned in the previous chapter regularization is such a crucial part off the implementation of a neural network that it is necessary to spend a lot of time experimenting and trying to choose the appropriate parameters for each regularization method that was used.

Dropout

Dropout was used at all parts of the model in order to prevent over-fitting. More specifically we applied dropout at the inputs of each LSTM layer and the inputs of the dense layers. We did not use any dropout on the recurrent connections of the LSTM layers because this would reduce their ability to remember values for many time-steps.

We dropped 60% of the LSTM units and 60% of the dense units of each network in order to get the best performing model. One can see that the dropout percentage is rather large in order to successfully prevent the networks from over-fitting.

Gaussian noise

Gaussian additive noise was applied to the inputs of both the skeletal and the speech model in order to enhance the training set. By adding random Gaussian noise to the data we create noisy training examples that prevent the network from learning the sampling error. It is especially useful for large networks that can very easily learn the sampling error and over-fit. Noisy data are definitely less effective than real data but it is not always easy to find more real training data.

The additive Gaussian noise is sampled from a Gaussian distribution with unit mean and its effect on the data is controlled by the variance of the distribution. In our experiments we used Gaussian noise with a variance of 0.5. One must be careful when choosing a value for the variance of the noise. If the variance is too low then the noise has very little effect on the data. A very high variance can corrupt the data and make the network untrainable.

Max-norm constraints

Max-norm regularization has been found to be very effective when combined with other regularization methods and especially with dropout [27]. By constraining the norm of the incoming weight vector at each hidden unit to have a fixed constant upper bound we prevent the weights from becoming too large. The constant that we used was 3 which is usually a reasonable value for most neural networks. Max-norm regularization was used in every layer except for the output CTC layer.

7.2 Training process

In this section we will present the complete training process of our model. Training the complete model is a long process and it can be split into three parts which are the training of the speech network, the training of the skeletal network and the training of the combined network. Because this process can usually take days to converge, before we start training the networks we need to set up some check-points.

7.2.1 Checkpoints

Check-points are an essential part for the training of very large neural networks. Especially when it comes to large recurrent networks, the training procedure can take a very long time. For this reason we need to be able to save the process, ideally at the end of each epoch, so that we can easily resume the training from that point in case something goes wrong.

For our model we decided that we wanted to save the best parameters at the end of each epoch in order to be able to easily roll back to the last stable state. More specifically we check at the end of every epoch to see if our training metrics have improved. In case this is true we save the weights to an external file and proceed with the next epoch. If the metrics have not improved then we keep the previous best set of weights. This checkpoints allowed us to roll back to the most recent stable state in a number of occasions.

7.2.2 Early stopping

Early stopping is a very easy but really useful mechanism that greatly increases training efficiency. It allows us to prevent a model from severely over-fitting without any additional overhead. We monitor the validation metrics every 5 epochs and we stop the training process when we find that the validation loss did not decrease for more the 10 iterations. This way we are able to stop the learning when we see that the model starts over-fitting the training set.

7.2.3 Training the models

Speech model

The training as we said earlier was performed in three parts. We started by training the speech network first. First we extracted the input features from the raw waveforms and split the training videos using 80% of the training videos for the training set and the rest 20% for the development validation (dev-validation) set. The validation videos were kept out of the training procedure and were just used to evaluate the trained models. We split both the training and the dev-validation set into mini-batches of 2 sample sequences each along with the corresponding label sequences.

The speech network was trained on mini-batches using the Adam optimizer for approximately 150 iterations while evaluating with the dev-validation set every 5 iterations. The training was stopped when the validation loss hadn't decreased for more than 50 epochs. The training of the speech network took approximately 2 days to converge on an Nvidia 1060 gtx. When the training was finished we evaluated the model on the validation set of the challenge that was kept unseen up to this point.

Skeletal model

Similar to the speech model was the training of the skeletal model. Again the first step was to extract the 22 skeletal features that were described in the previous chapter. Once the features were extracted from the data files for each video we split the training set again using 80% for the training set and the rest 20% for the development validation (dev-validation) set. The validation videos of the challenge were used in order to evaluate the trained network while the test videos were not used at this part of the training. Once again the training and dev-validation set were split into mini-batches of 2 sequences along with the corresponding targets.

The training procedure was exactly the same that was used for the speech network and was described in the previous section. This model took more than two days of training in order to converge on the gpu. Once the model was trained the validation sequences were used in order to evaluate the performance.

In Table 7.1 are presented the training and validation set metrics for both the skeletal and the speech networks. As we can see the speech network significantly outperforms the skeletal network while both networks make a lot of insertion errors and have poor accuracy. We observed the output predictions of both models and found out that they were making different errors which means that when combined they are likely to improve the overall performance.

Table 7.1: **Speech and skeletal model performance.** The validation set of the challenge was used in order to test the different networks. We used 20 distinct classes for the skeletal model while for the speech model we used 42 distinct words.

Model	Accuracy	Correct	Deletions	Insertions	Substitutions
Speech	35.7%	98%	22	4894	133
Skeletal	-7.36%	62%	96	2342	1150

As it was mentioned previously the skeletal model uses 20 distinct classes while the speech model uses 42 different keywords. In table 7.2 the words corresponding to the different class labels.

Table 7.2: **Class labels and corresponding keywords.**

Class label	Words
VA	Vattene
VQ	Vieni qui
PF	Perfetto
FU	E' un furbo
CP	Che due palle
CV	Che vuoi
DC	Vanno d'accordo
SP	Sei Pazzo
CN	Cos'hai combinato
FN	Non me ne frega niente
OK	Ok
CF	Cosa ti farei
BS	Basta
PR	Le vuoi prendere
NU	Non ce n'e piu
FM	Ho fame
TT	Tanto tempo fa
BN	Buonissimo
MC	Si sono messi d'accordo
ST	Sono stufo

Combining the models

After both models have reached convergence we proceeded on combining them. We removed the dense output layers from both models in order to use them as feature extractors for the larger network and we added the top LSTM layer along with a CTC output layer on top of them. We had decided that it would be easier to just train the final LSTM layer since the previous layers have already learned reasonable parameters. For that reason the weights of the LSTM layers for both models were frozen and were not updated any further. Then we started training the top layer.

Much like the training of the two sub-networks, the complete network was trained using the Adam optimizer on mini-batches of two multimodal sequences each for approximately 50 epochs. The multimodal input sequences included both the audio feature vector and the skeletal feature vector and the training targets were the same sequences of class labels that were used for the skeletal network. The performance of the model was evaluated every 5 epochs on the validation set and the training was stopped when the validation loss did not decrease for 5 iterations.

The training of the complete network was much quicker than the two sub-networks and it only took 20 hours to converge. The reason for this is that the trainable parameters at this step are much fewer than the previous models. Also the feature extractors have already learned good parameters and thus make it easier for the model to find the optimal region of the weight space.

7.3 Evaluating the model

In order to evaluate the model we used the same setup that was used for the ChaLearn Challenge. The validation set of the challenge was kept separate and was used to evaluate different trained models in order to optimize the different hyper-parameters. The validation set consisted of approximately 280 video sequences with a total of 2700 gestures performed. The video sequences in this set were performed by different users than the training set and also included out-of-vocabulary gestures and speech mixed in among with the gestures and keywords. The test set of the challenge consisted of approximately 270 sequences that were similar to the validation set.

When evaluating the model we gave it as input multimodal sequences of features extracted from the validation sequences and let the model predict the correct label sequence. Then the model predicted the most probable class at each time step. We used a probability threshold of 0.88 on these predictions in order to remove the predictions with low probability and grouped the remaining predictions into the final output sequence. Finally we compared the predicted sequences with the target sequences and measured the overall performance of the model. For all experiments, the basic measures used to evaluate performance were the label accuracy and the label error rate (LER). The probability threshold was determined using experiments on the validation set. We wanted to have a nice balance between the insertion and deletion errors in order to get the best possible accuracy.

This process was repeated every time we trained a complete model. One can imagine that this iterative procedure took quite a long time in order to get the best possible set of hyper-parameters. Once all the hyper-parameters were determined we proceeded on evaluating the final model on the test set of the challenge that was unseen up to this point. Once again we repeated the above procedure and compared the predicted sequences with the target sequences and measured the performance.

The complete model clearly outperformed both the speech and the skeletal models by quite a large margin. The model is able to successfully combine

the two streams and make them cooperate in order to greatly improve the classification accuracy. When observing the output sequences it is clear that the two networks are doing quite a good job at cooperating with one another. The insertion errors that were affecting the performance of the two uni-modal networks were drastically reduced and as a result the classification accuracy greatly improved.

In table 7.3 we present the validation set performance of the complete model as well as the error counts. These are compared with the performance of the two uni-modal networks in order to measure the overall improvement.

Table 7.3: **Multimodal results compared with the uni-modal results.** These are the results on the validation set with the same 20 distinct classes used by the skeletal model.

Model	Accuracy	LER	Correct	Deletions	Insertions	Substitutions
Speech	35.7%	64.3%	98%	22	4894	133
Skeletal	-7.36%	-107.36%	62%	96	2342	1150
Multimodal	79.1%	20.9 %	92.9%	155	593	83

In table 7.4 we present the overall accuracy and label error rate on both the validation and the test set. As we can see there is a slight improvement on the test set which means that our model does not over-fit at all and is able to generalize properly.

Table 7.4: **Multimodal accuracy and label error rate (LER) on the validation and test set.**

Set	Number of sequences	Accuracy	LER	Correct
Validation set	286	79.1%	20.9 %	92.9%
Test set	274	81.2%	18.8 %	93.3%

7.4 Comparisons with different approaches

In this section we are going to compare our model with other approaches on the same problem. Most of the top performing approaches on the ChaLearn Challenge were utilizing a combination of multimodal features with HMM or random forest frameworks over previously segmented gesture intervals.

The first-ranked team (IV AMM) [32] uses a feature vector of combined audio and skeletal features. A simple algorithm that detects end-points based on joint coordinates is applied to segment continuous data sequences into candidate gesture intervals. A Hidden Markov Model is trained with 39-dimensional MFCC features processes the audio features and generates prediction scores for each gesture class. A skeletal feature classifier that is based on Dynamic Time Warping is used to provide complementary information. The predictions generated by the two classifiers are combined to produce a weighted sum for late fusion. Finally a single threshold is employed to discriminate between meaningful gestures and out-of-vocabulary (OOV) instances.

The second-ranked team (WWEIGHT) [7] combines audio and skeletal information, using both joint spatial distribution and joint orientation. They find potential gestures using a high audio-energy detection method and use a log-spaced audio spectrogram as well as the joint positions and orientations above the hips as their input features. For the training of the models they employed a random forest (RF) and a k-nearest neighbor (KNN) model. Finally they average the posteriors from these models and convert them to predicted sequences with simple heuristics.

The third-ranked team (ET) [2] combine the output decisions of two models. The features used are based on the skeleton information and the audio signal. They perform an unsupervised search for possible gestures and extract MFCC features from these intervals. Using these features, they train a random forest (RF) and a gradient boosting classifier. The second model uses simple statistics (median, var, min, max) on the first 40 frames for each gesture to build the training samples. The predictions are made using a sliding window. The outputs of the two models were fused using a weighted average of the outputs.

Table 7.5: **Our approach in comparison with the top performing approaches of the ChaLearn challenge.** We include recognition accuracy, label error rate as well as the type of model used in each approach and the modalities utilized.

Approach	Rank	Modalities	Type of model	Accuracy	LER
Our	-	Audio, Skeletal	LSTM, CTC	81.2%	18.8%
iva.mm	1	Audio, Skeletal	HMM, DTW	87.2%	12.8%
wweight	2	Audio, Skeletal	RF, KNN	84.6%	15.4%
E.T.	3	Audio, Skeletal	RF, Boosting	82.9%	17.1%

In table 7.5 we provide the accuracy and label error rate (LRE) of these approaches in comparison with our approach. As one can see our own model fared quite well compared to the top performing teams although our approach is drastically different. Every single one of the top performing approaches used pre-processing and some segmentation method in order to split the sequences into possible gesture intervals. Our approach on the other hand has no need of these pre-processing steps because the classification is performed in a temporal way. Another difference is that our approach uses LSTM networks that are far more scalable than random forests and KNN methods. Especially in real world applications LSTM networks can make better use of large amounts of training data. Although the ChaLearn data set is not that large we were still able to effectively train deep LSTM networks without overfitting.

Chapter 8

Conclusions and Future Work

The aim of this thesis was to investigate state-of-the-art machine learning methods for the problem of multimodal gesture recognition. In particular we focused on applying LSTM architectures with CTC for temporal modeling of gesture sequences. LSTM networks are currently the state-of-the-art approach in most sequence classification tasks and have demonstrated great performance in various problems. They are exceptionally good at modeling long time sequences and can scale much better than traditional machine learning methods like HMMs, especially when large amounts of training data are available. We showed that an approach that is based on LSTM networks can solve the problem of gesture recognition in a very efficient way.

Another main aspect of our work in this thesis was the use of connectionist temporal classification (CTC) that allowed us to train our LSTM networks directly for sequence labeling tasks with unknown input-output alignments. This approach is much more efficient than training using pre-segmented sequences and offers a number of advantages over the methods proposed in other approaches. CTC in general helped us solve the problem of gesture recognition in a very elegant way avoiding the use of long pipelines of different models. This is especially useful in real world applications, where there is a need for an end-to-end modeling approach.

Finally, we demonstrated how multiple modalities combined in a single model can greatly outperform uni-modal approaches and produce excellent results. We also proposed an efficient way of combining LSTM networks that use different modalities in one single end-to-end model. Our mutimodal results were comparable to the top teams that participated in the ChaLearn Challenge, while our approach was much different than most others. We believe that LSTM networks with CTC will be able to produce even better results when given a lot more training examples. Deep LSTM networks with millions of trainable parameters will be able to scale better and outperform HMM models and random forest classifiers in real world applications where there is an abundance of labeled data.

In the future we would like to scale our approach in a much larger data set in order to further improve classification results. Also, we would like to incorporate more than two modalities in order to further improve performance.

RGB and depth video can be very promising, and we believe that an approach that incorporates these streams will be beneficial.

Another direction that we would like to pursue is the use of unsupervised learning. As we mentioned previously, the ChaLearn data set is not particularly large, and this hinders our ability to train even more powerful models. This is the kind of scenario where unsupervised pre-training with large amounts of unlabeled data can be very beneficial. A model that is pre-trained on large amounts of unlabeled data will be able to learn reasonable initial values for the different parameters. Then the labeled data can be used to fine-tune the pre-trained model in a discriminative way for classification. The pre-trained weights will have values that lie in a reasonable area of the parameter space, and, as a result, they will converge quickly to their optimal values without over-fitting.

Bibliography

[1] P. Baldi, S. Brunak, P. Frasconi, G. Soda, and G. Pollastri, "Exploiting the past and the future in protein secondary structure prediction," *Bioinformatics*, vol. 15, no. 11, pp. 937–946, 1999.

[2] I. Bayer and T. Silbermann, "A multi modal approach to gesture recognition from audio and video data," in *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*. ACM, 2013, pp. 461–466.

[3] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.

[4] H. Cooper and R. Bowden, "Large lexicon detection of sign language," *Lecture Notes in Computer Science*, vol. 4796, pp. 88–97, 2007.

[5] T. Dozat, "Incorporating nesterov momentum into adam," 2016.

[6] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[7] S. Escalera, J. Gonzàlez, X. Baró, M. Reyes, O. Lopes, I. Guyon, V. Athitsos, and H. Escalante, "Multi-modal gesture recognition challenge 2013: Dataset and results," in *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*, 2013, pp. 445–452.

[8] S. Escalera, J. Gonzàlez, X. Baró, M. Reyes, O. Lopes, I. Guyon, V. Athitsos, and H. J. Escalante, "Multi-modal gesture recognition challenge 2013: Dataset and results," in *Proceedings of the 15th ACM on International Conference on Multimodal Interaction*. ACM, 2013, pp. 445–452.

[9] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout networks," 2013.

[10] A. Graves *et al.*, *Supervised sequence labelling with recurrent neural networks*. Springer, 2012, vol. 385.

[11] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 369–376.

[12] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, pp. 6645–6649.

[13] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional lstm and other neural network architectures," *Neural Networks*, vol. 18, no. 5, pp. 602–610, 2005.

[14] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778.

[15] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *arXiv:1207.0580*, 2012.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] H. Jaeger, *Short term memory in echo state networks*. GMD-Forschungszentrum Informationstechnik, 2001, vol. 5.

[18] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014.

[19] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[21] P. Molchanov, S. Gupta, K. Kim, and K. Pulli, "Multi-sensor system for driver's hand-gesture recognition," in *Automatic Face and Gesture Recognition (FG), 2015 11th IEEE International Conference and Workshops on*, vol. 1, 2015, pp. 1–8.

[22] V. Pitsikalis, A. Katsamanis, S. Theodorakis, and P. Maragos, "Multimodal gesture recognition via multiple hypotheses rescoring," in *Gesture Recognition*. Springer, 2017, pp. 467–496.

[23] L. R. Rabiner, "A tutorial on hidden markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[24] D. E. Rumelhart, G. E. Hinton, J. L. McClelland *et al.*, "A general framework for parallel distributed processing," *Parallel distributed processing: Explorations in the microstructure of cognition*, vol. 1, pp. 45–76, 1986.

[25] M. Salmen and P. G. Ploger, "Echo state networks used for motor control," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on.* IEEE, 2005, pp. 1953–1958.

[26] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.

[27] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[28] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.

[29] W. Talloen, D.-A. Clevert, S. Hochreiter, D. Amaratunga, L. Bijnens, S. Kass, and H. W. Göhlmann, "I/ni-calls for the exclusion of non-informative genes: a highly effective filtering tool for microarray data," *Bioinformatics*, vol. 23, no. 21, pp. 2897–2902, 2007.

[30] T. Tieleman and G. Hinton, "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude," *COURSERA: Neural networks for machine learning*, vol. 4, no. 2, pp. 26–31, 2012.

[31] R. J. Williams and D. Zipser, "Gradient-based learning algorithms for recurrent networks and their computational complexity," *Backpropagation: Theory, architectures, and applications*, vol. 1, pp. 433–486, 1995.

[32] J. Wu, J. Cheng, C. Zhao, and H. Lu, "Fusing multi-modal features for gesture recognition," in *Proceedings of the 15th ACM on International conference on multimodal interaction.* ACM, 2013, pp. 453–460.

[33] H.-S. Yoon, J. Soh, Y. J. Bae, and H. S. Yang, "Hand gesture recognition using combined features of location, angle and velocity," *Pattern recognition*, vol. 34, no. 7, pp. 1491–1501, 2001.

[34] S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, X. Liu, G. Moore, J. Odell, D. Ollason, D. Povey *et al.*, "The htk book," *Cambridge university engineering department*, vol. 3, p. 175, 2002.

[35] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," 2014.

[36] M. D. Zeiler, "Adadelta: An adaptive learning rate method," 2012.