Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia

COPPE UFRJ

END-TO-END SPEECH RECOGNITION APPLIED TO BRAZILIAN PORTUGUESE USING DEEP LEARNING

Igor Macedo Quintanilha

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia Elétrica, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia Elétrica.

Orientadores: Luiz Wagner Pereira Biscainho
Sérgio Lima Netto

Rio de Janeiro
Março de 2017

END-TO-END SPEECH RECOGNITION APPLIED TO BRAZILIAN PORTUGUESE USING DEEP LEARNING

Igor Macedo Quintanilha

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA ELÉTRICA.

Examinada por:

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2017

Quintanilha, Igor Macedo

End-to-end speech recognition applied to Brazilian Portuguese using Deep Learning/Igor Macedo Quintanilha. – Rio de Janeiro: UFRJ/COPPE, 2017.

XX, 101 p.: il.; 29,7cm.

Orientadores: Luiz Wagner Pereira Biscainho
Sérgio Lima Netto

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia Elétrica, 2017.

Referências Bibliográficas: p. 90 – 101.

1. lvcsr. 2. connectionist temporal classification. 3. deep learning. I. Biscainho, Luiz Wagner Pereira *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia Elétrica. III. Título.

To my parents, my siblings, my closest friends, and to my love, Karina.

Agradecimentos

Uma longa caminhada tem sido feita ao longo desses anos, e eu não conseguiria realizá-la sem estar rodeado de pessoas especiais. São tantas pessoas que vieram, me ajudaram e foram embora, e tantas outras que estão aqui desde o começo da caminhada que fica impossível agradecer individualmente. Gostaria de agradecer a todas essas pessoas especiais que apareceram (e estão) na minha vida e que permitem que eu continue com os meus estudos. A todos vocês, minha eterna gratidão.

Gostaria de agradecer aos meus pais, Simone Macedo Quintanilha e Michel Campista Quintanilha, que nunca deixaram de acreditar em minhas capacidades e até hoje me auxiliam de todas as formas possíveis.

Aos meus irmãos, Yuri Macedo Quintanilha e Juliana Macedo Quintanilha, que são minha fonte diária de inspiração e força, pela garra que mantiveram ao longo de suas caminhadas; pelas pessoas que estão se tornando; e pelo suporte incondicional que me deram.

Agradeço também à minha família oriental, pois sem vocês não chegaria tão longe. Vocês me proveram refúgio, alimento e aconchego nas horas mais necessitadas.

À Karina Yumi Atsumi: você sabe que não posso descrever o quanto lhe agradeço por todos esses anos de companheirismo e amor. Essa dissertação é tão sua quanto minha. Meu eterno obrigado, hoje e sempre.

À Thais Fernandes, por seu bom humor não importando a hora do dia, pelas conversas, pela companhia diária e por me inspirar com a sua determinação de sempre seguir em frente, independente dos obstáculos que surgem.

Ao Luiz Wagner Pereira Biscainho, meu orientador, agradeço imensamente por tudo. Meus pais ficam mais tranquilos sabendo que tenho você para auxiliar meu caminho acadêmico e profissional. Muito obrigado mesmo por toda a paciência, dicas, por escutar meus problemas e por tantas outras coisas que surgiram nesses anos.

Ao Sérgio Lima Netto, meu (co-)orientador, muito obrigado por ter aceitado essa empreitada, por todas as dicas e conselhos que tive e pelo ótimo humor em todas as nossas reuniões.

Ao pessoal do Laboratório de Sinais, Multimídia e Telecomunicações, pelo total suporte que tive ao longo desses anos — vocês são minha segunda (primeira) casa.

Agradeço à Universidade Federal do Rio de Janeiro, ao Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa de Engenharia (COPPE) e em especial ao Programa de Engenharia Elétrica (PEE) pela minha formação. Sem esses centros de excelência eu não seria nada.

E, finalmente, agradeço à Fundação de Amparo à Pesquisa do Estado do Rio de Janeiro (FAPERJ) pelo apoio financeiro.

Ao Matheus, ao Felipe e ao Rafael, que já iam reclamar por não estar nesses agradecimentos, muito obrigado por todos esses anos e por conseguir me tirar dos estudos de vez em quando. Gostaria de ter começado essa amizade anos antes. Ao Roberto, que sempre me acompanhou nessa caminhada, muito obrigado pela paciência e amizade em todos esses anos.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

RECONHECIMENTO DE VOZ PARA O PORTUGUÊS BRASILEIRO UTILIZANDO REDES NEURAIS DE PONTA-A-PONTA

Igor Macedo Quintanilha

Março/2017

Orientadores: Luiz Wagner Pereira Biscainho
Sérgio Lima Netto

Programa: Engenharia Elétrica

Apresenta-se nesta dissertação um sistema de reconhecimento de voz utilizando redes neurais profundas treinadas de ponta-a-ponta baseado em caracteres para o idioma Português Brasileiro (PT-BR). Para isso, foi desenvolvida uma base de dados através de um conjunto de quatro bases (sendo três distribuídas gratuitamente) disponíveis. Foram conduzidos diversos testes variando o número de camadas, aplicando diferentes métodos de regularização e ajustando diversos outros hiperparâmetros da rede neural. O melhor modelo atinge, na nossa base de teste, uma taxa de erro de caractere de 25,13%, 11% maior que a reportada pelos sistemas comerciais. Esse resultado mostra que é possível construir um sistema automático de reconhecimento de voz para o idioma PT-BR utilizando redes neurais treinadas de ponta-a-ponta.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

END-TO-END SPEECH RECOGNITION APPLIED TO BRAZILIAN PORTUGUESE USING DEEP LEARNING

Igor Macedo Quintanilha

March/2017

Advisors: Luiz Wagner Pereira Biscainho
Sérgio Lima Netto

Department: Electrical Engineering

In this work, we present a character-based end-to-end speech recognition system for Brazilian Portuguese (PT-BR) using deep learning. We have developed our own dataset — an ensemble of four datasets (three publicly available) that we had available. We have conducted several tests varying the number of layers, applying different regularization methods, and fine-tuning several other hyperparameters. Our best model achieves a label error rate of 25.13% on our test set, 11% higher than commercial systems do. This first effort shows us that building an all-neural speech recognition system for PT-BR is feasible.

Contents

List of Figures

List of Tables

List of Symbols

$\{0, 1, \ldots, n\}$	A set of all integers between 0 and n, p. 6
$\mathbb{A} \setminus \mathbb{B}$	Set subtraction, *i.e.*, the set containing the elements of $\mathbb{A}$ that are not in $\mathbb{B}$, p. 60
$\boldsymbol{A} \odot \boldsymbol{B}$	Element-wise (Hadamard) product of $\boldsymbol{A}$ and $\boldsymbol{B}$, p. 18
a_i	Element i of the random vector $\mathbf{a}$, p. 21
a_i	Element i of vector $\boldsymbol{a}$, p. 6
$a^{(l)}$	A variable of the l-th layer, p. 6
a_{-1}	The last element of vector $\boldsymbol{a}$, p. 60
$\boldsymbol{A}^T$	Transpose of matrix $\boldsymbol{A}$, p. 5
$\boldsymbol{a}^{(t)}$	A vector at time step t, p. 34
α	Hyperparameter of RMSProp, p. 29
$\alpha_{\{v,m\}}$	Hyperparameters of Adam, p. 29
$A_{i,j}$	Element i,j of matrix $\boldsymbol{A}$, p. 15
$\boldsymbol{A}$	A matrix, p. 6
$\boldsymbol{a}$	A vector, p. 5
$\mathbb{A}$	a set, p. 9
a	A scalar random variable, p. 21
$\mathbf{a}$	A vector-valued random variable, p. 21
a	A scalar (integer or real), p. 5
b	Bias, p. 5
$\boldsymbol{c}^{(t)}$	Cell state at time step t, p. 38

$\boldsymbol{\delta}$	Gradient of the loss w.r.t. the node's output, p. 18
$\frac{\mathrm{d}y}{\mathrm{d}x}$	Derivative of y with respect to x, p. 13
η	Learning rate, p. 27
$\mathbf{E}[\cdot]$	Expectation, p. 9
$f(\boldsymbol{x};\boldsymbol{\theta})$	A model with input $\boldsymbol{x}$ parametrized by $\boldsymbol{\theta}$, p. 5
f^*	Desired function, p. 5
$f \circ g$	Composition of the functions f and g, p. 6
$\boldsymbol{f}$	Forget gate, p. 38
$\boldsymbol{g}$	Block input gate, p. 39
$\boldsymbol{h}^{(t)}$	Hidden state at time step t, p. 34
$\overleftarrow{\boldsymbol{h}}^{(t)}$	Backward hidden state, p. 35
$\overrightarrow{\boldsymbol{h}}^{(t)}$	Forward hidden state, p. 35
$\mathbb{H}$	Hypothesis set, p. 60
h	Hidden unit, p. 5
$\boldsymbol{i}$	Input gate, p. 38
κ	Pre-emphasis factor, p. 56
$\boldsymbol{l}^+$	Augmented label sequence, p. 52
λ	Weight decay factor, p. 25
$\log x$	Natural logarithm of x, p. 50
$\boldsymbol{\mathcal{L}}$	Loss function, $\boldsymbol{\mathcal{L}} : \mathbb{R}^D \to \mathbb{R}$, p. 8
$\boldsymbol{l}$	A label sequence, p. 51
μ	Momentum factor, p. 28
$\boldsymbol{m}$	A mask, p. 25
$\boldsymbol{o}$	Output gate, p. 39
$\frac{\partial f}{\partial \boldsymbol{x}}$	Jacobian matrix $\frac{\partial f}{\partial \boldsymbol{x}} \in \mathbb{R}^{M \times N}$ of $f : \mathbb{R}^N \to \mathbb{R}^M$, p. 14

$\frac{\partial y}{\partial x}$	Partial derivative of y with respect to x, p. 14
$\Phi(\boldsymbol{l})$	A set of CTC paths for $\boldsymbol{l}$, p. 52
$\phi(\cdot)$	Activation function, applied element-wise, p. 5
$\mathrm{P}(\mathrm{a})$	A probability distribution over a variable a, p. 25
$\mathbb{R}$	The set of real numbers, p. 11
$\boldsymbol{r}$	Reset gate, p. 42
$\mathbb{S}$	Training set, p. 9
$\sigma(x)$	Logistic sigmoid, $\frac{1}{1+e^{-x}}$, p. 7
$\boldsymbol{s}$	A signal, $\boldsymbol{s} \in \mathbb{R}^N$, p. 56
$\boldsymbol{u}$	Update gate, p. 42
u	Staircase function, p. 18
$\varnothing$	Blank label, p. 51
$\mathbf{var}(\cdot)$	Variance, p. 21
$\boldsymbol{w}^{\mathrm{in}}$	A window function, $\boldsymbol{w}^{\mathrm{in}} \in \mathbb{R}^L$, p. 56
$\boldsymbol{W}$	Weights of a layer, p. 6
$\boldsymbol{w}$	Weights of a hidden unit, p. 5
$\boldsymbol{x}$	Input, p. 5
$\boldsymbol{y}$	Target, p. 8
$\hat{\boldsymbol{y}}$	Output of the model $f(\boldsymbol{x};\boldsymbol{\theta})$, p. 5

List of Abbreviations

Adam	ADAptive Momentum Estimation, p. 29
AI	Artificial Intelligence, p. 1
AlexNet	Convolutional neural network topology proposed by Krizhevsky *et al.* [1], p. 33
ASR	Automatic Speech Recognition, p. 2
Backprop	Backpropagation, p. 12
Batch Norm	Batch Normalization, p. 22
BLSTM	Bidirectional Long Short-Term Memory, p. 62
BN	Batch Normalization, p. 22
BRNN	Bidirectional Recurrent Neural Network, p. 35
BRSD	Brazilian Portuguese Speech Dataset, p. 73
CLM	Character level Language Model, p. 64
CNN	Convolutional Neural Network, p. 30
CNTK	Microsoft Cognitive Toolkit, p. 76
ConvNet	Convolutional Neural Network, p. 30
CPU	Central Processing Unit, p. 27
CSLU	Center for Spoken Language Understanding, p. 73
CTC	Connectionist Temporal Classification, p. 3
DARPA	Defense Advanced Research Projects Agency, p. 71
DCT	Discrete Cosine Transform, p. 57
DNN	Deep Neural Networks, p. 64

FSH	Fisher dataset, p. 66
GMM	Gaussian Mixture Models, p. 47
GPU	Graphics Processing Unit, p. 27
GRU	Gated Recurrent Unit, p. 41
HMM	Hidden Markov Model, p. 2
HOG	Histogram of Oriented Gradient, p. 47
LDC	Linguistic Data Consortium, p. 64, 71
LeNet	Convolutional neural network topology proposed by Lecun *et al.* [2], p. 33
LER	Label Error Rate, p. 62
LN	Layer Normalization, p. 45
LPC	Linear Predictive Coding, p. 48
LSTM	Long Short-Term Memory, p. 38
LVCSR	Large Vocabulary Continuous Speech Recognition, p. 74
MBR	Minimum Bayes' risk, p. 49
MCE	Minimum Classification Error, p. 49
MFCC	Mel Frequency Cepstral Coefficients, p. 56
MILA	Montreal Institute for Learning Algorithms, p. 1
MI	Multiplicative Integration, p. 45
MIT	Massachusetts Institute of Techonology, p. 71
ML	Machine Learning, p. 33
MLP	Multilayer Perceptron, p. 4
MMI	Maximum Mutual Information, p. 49
MSE	Mean Square Error, p. 10
NIST	National Institute of Standards and Technology, p. 72
PHN	Phoneme, p. 73

PLP	Perceptual Linear Predictive, p. 48
ReLU	Rectified Linear Units, p. 8
ResNet	Residual Network, p. 1
RGB	Red, Green, and Blue, p. 29
RMSProp	Root Mean Square Propagation, p. 28
RNN	Recurrent Neural Network, p. 33
SGD	Stochastic Gradient Descent, p. 27
SNR	Signal-to-Noise Ratio, p. 74
SWB	SwitchBoard dataset, p. 66
UBI	Universal Basic Incoming, p. 1
VGG	Visual Geometry Group, p. 33
WER	Word Error Rate, p. 64
WFST	Weighted Finite-State Transducer, p. 64
WRD	Word, p. 73
WSJ	Wall Street Journal *corpus*, p. 64
XOR	eXclusive OR, p. 9

Chapter 1

Introduction

1.1 The rise of Deep Learning

Since 2006, the world has drastically changed, but unfortunately, only a few people have noticed. In that year, a game-changing algorithm was (re)born: deep learning [3]. After that, the artificial intelligence (AI) field has conquered research and industry, from pedestrian recognition and self-driving cars [4, 5] to speech recognition and disease identification [6, 7]. A billionaire market was raised, and big companies are self-adapting and applying AI with deep learning in the oddest scenarios.

People, like Elon Musk, and nations, like Sweden, have been talking about universal basic incoming (UBI) – a monthly paid salary to every citizen, with or without a job – worried that AI will destroy about 1/3 of jobs in the next few decades, leaving a great fraction of population jobless. Specialists say that in the future, jobs will be a privilege for specialized persons.

Studying deep learning topologies and algorithms will for sure increase the chances of having employment in the near/far future. While Microsoft ends the year of 2016 allocating 10,000 of its workers to work with artificial intelligence and Google has donated millions of dollars to MILA lab, in Montreal – Canada, the birthplace of deep learning, Elon Musk with others founded OpenAI, "a non-profit research company responsible to build a safe AI and ensure AI's benefits are as widely and evenly distributed as possible", which raised more than one billion of investment in a few months.

The last couple of years has shown the power of deep learning. In 2015, Microsoft presented a topology (called residual network or ResNet) that surpassed the human capability of image classification on ImageNet – a dataset of millions of images organized in 1000 classes, of which 100 are dog breeds. In the following, Google showed a text-to-speech architecture (based on PixelCNN) capable of synthesizing human speech that received 50% better evaluation in subjective tests than the best previous system of the company. Using deep reinforcement learning, a team of researchers of Deep Mind – a two-year old British start-up bought by Google for U$650 million – managed to win the best Go (Chinese board game) player in the world; and yet, 10 years before, people used to say it was impossible to build an AI able to play Go, due to the (virtually infinite) number of possible movements on the board.

Speech recognition, machine translation, pedestrian detection, image classification, image detection, image segmentation, and gene-related disease identification are some of the areas where deep learning has been responsible for a huge revolution in the last years. Despite researchers' efforts, understanding what is behind this magnificent idea is still a future, and much research is needed. For now, scientists are creating and trying new architectures, improving their results, breaking successive benchmarks and applying this knowledge in other fields far from image classification.

We are in an exciting time for AI-related research.

1.2 Evolution of automatic speech recognition (ASR) systems: from hidden Markov models (HMMs) to end-to-end solutions

Seventy years ago [8], researchers gave the first steps on the automatic recognition of spoken digits; later on, they have developed several systems for continuous speech recognition, built on a small number of possible words. Without a doubt, the first major technology breakthrough in this area was made at the end of 1960, with the development of hidden Markov models [9] (HMM), enabling the combination of acoustic, language and lexicon models in one probabilistic algorithm.

In the 1980s, it was already possible to build a speaker-dependent system capable of recognizing more than 20,000 words. In the 1990s, commercial solutions have arisen, like the famous IBM Via Voice Center, popularized in Brazil.

Although many others HMM-based algorithms have been developed from the end of 1990s to the beginning of 2012, they brought few significant advances. The increasing computational power — along with the development of powerful GPUs — the availability of huge amount of data, and the development of deep learning made possible, in 2012, the deployment of a new system [6, 10] that led to the highest advance over 20 years in the speech recognition area, improving the previous system performance by over 30%. From 2012 on, ASR got back to the spotlight, leading to hundreds of papers in the speech processing area.

More recently, there is a hype among the researchers in applying only one neural-based system to perform the speech recognition, in an end-to-end solution. Graves *et al.* [11] have made the first successful algorithm, proposing the connectionist temporal classification. Moreover, Chorowski *et al.* [12] have successfully applied an encoder-decoder neural-based model with attention mechanism, which had primarily been employed to neural machine translation, to an end-to-end ASR system.

The work in ASR is not done. The Holy Grail is to develop a unique automatic speech recognition system capable of understanding human speech in the vastest scenarios — from a quiet room to a war zone — and in different languages. Fortunately, we are far from this goal.

1.3 Contributions of this dissertation

The main contribution of this dissertation is the first freely available character-based end-to-end ASR solution for the Brazilian Portuguese language we have knowledge of. Furthermore, we have built our own dataset — a preprocessed ensemble of four different datasets (three of which are publicly available, one paid) that were available to us. The best model achieves a character error rate of 25.13%.

1.4 Chapter organization

We give a brief revision of (deep) neural networks in Chapter 2 — from neurons, layers, optimization, initialization methods, and regularizers to convolutional neural networks and recurrent neural networks.

In Chapter 3, we describe the problem behind HMM-based systems and how the connectionist temporal classification (CTC) method has arisen to overcome it. Also, we discuss how we can decode the sequences from CTC-based models, and how we can improve our ASR system with a language model. Then, we go through several end-to-end models that have been proposed over the years. Finally, we discuss our proposed model.

In Chapter 4, we introduce the TIMIT dataset, used to validate our implementation. Also, we present our Brazilian Portuguese dataset, built from an ensemble of four distinct datasets. Next, we conduct several experiments to build our best final model and discuss many design choices.

Finally, in Chapter 5 we conclude our work and point to possible future directions that our work could bring.

Chapter 2

Neural Networks

"Virtually nothing is known about the computational capabilities of this latter kind of machine. We believe that it can do little more than can a low order perceptron."
— Minsky and Papert, *1971*

Created in 1958 and giving hope to the research community as a pillar to construct a real AI system, being disgraced in 1969 by Marvin Minsky and Seymour Papert, gaining its way back as a suitable machine learning algorithm in 1989, and growing at an unseen rate with deep learning since 2006, neural network had its ups and downs, but it is undeniable that it has gained its space recently as a state-of-the-art algorithm for many applications.

This chapter introduces the basic concepts of neural networks (and deep learning).

2.1 Feedforward neural networks

The feedforward neural network, also often called multilayer perceptron (MLP), can be defined as a directional acyclic graph and has the property of being a universal approximator [13, 14]. It is easier to understand than it looks.

This seminal neural network is made of simpler structures called neurons, which are organized in layers, each with one or more neurons. Neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections, as illustrated in Fig. 2.1. The feedforward neural network has an input, connected to all neurons in the first layer, and has an output, connected to every neuron in the last layer. Commonly, the input is called input layer, the output is called output layer, and the layers between them, hidden layers. This network is called feedforward because the information flows only from input to output: there

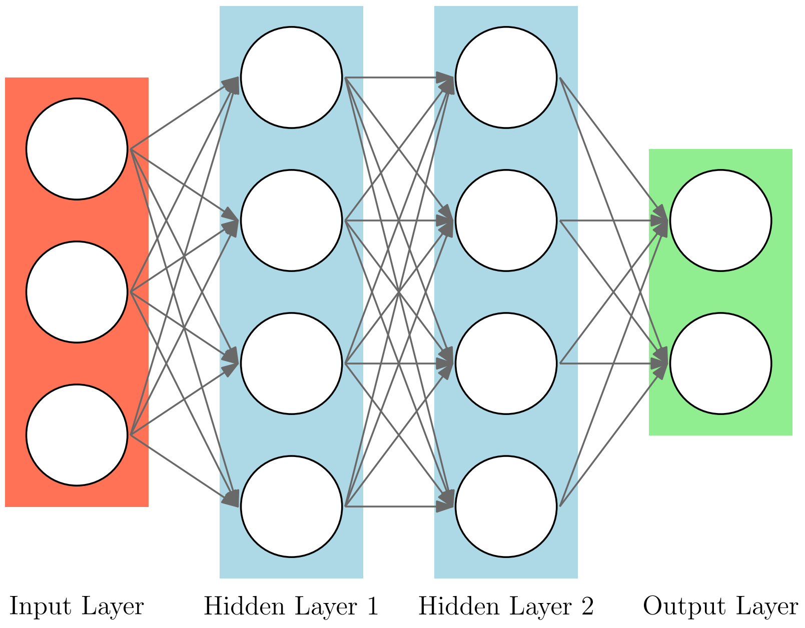


Figure 2.1: A 3-layer feedforward neural network with three inputs, two hidden layers of four neurons each, and one output layer with two neurons. Notice that there are connections between neurons across layers, but not within a layer.

is no information from the output being fed back to the input.

A neuron can be viewed as a node in a graph, and each connection as an edge with its own weight. A generic neuron is illustrated in Fig. 2.2. The information flows from input to output (from one layer to another) through the edges, giving directionality to the graph. Neurons of one layer are only connected to the neurons of the adjacent layers, and there is no feedback connection in this model; thus the graph is acyclic, *i.e.*, starting from one node you cannot reach this point again following the edges.

Mathematically, neurons apply an arbitrary function called activation (of neuron), ϕ, to an affine projection of its inputs, $\boldsymbol{x}$, returning a single output,

$$h = \phi(\boldsymbol{w}^T\boldsymbol{x} + b), \tag{2.1}$$

where $\boldsymbol{w}$ and b are the weights and bias of the neuron, respectively. Here, we will simply denote Eq. (2.1) as $h = \phi(\boldsymbol{x})$ to make some concepts easier to understand. As we said, a feedforward network contains many layers with many neurons. The goal of such network is to approximate some function f^*. A feedforward network defines a mapping $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of those parameters $\boldsymbol{\theta}$ that results in the best approximation to the desired model f^*.

Due to its layered form, we can compute all outputs of a layer at once. The l-th

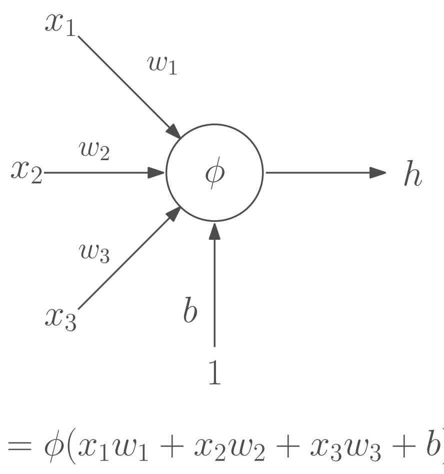


Figure 2.2: Every input (inbound edges) of the neuron (node), x_i, $i \in \{1, 2, 3\}$, is weighted by w_i, $i \in \{1, 2, 3\}$, (given by the value of its edge), summed together with a bias, and transformed by a function ϕ. Another way to look at this is considering the bias as another input with a fixed value of 1.

layer of a network can be described as

$$\boldsymbol{h}^{(l)} = \phi(\boldsymbol{W}^T\boldsymbol{h}^{(l-1)} + \boldsymbol{b}) = f^{(l)}(\boldsymbol{h}^{(l-1)}), \tag{2.2}$$

where

$$\begin{aligned} \boldsymbol{W} &= \begin{bmatrix} \boldsymbol{w}_1 & \boldsymbol{w}_2 & \cdots & \boldsymbol{w}_n & \cdots & \boldsymbol{w}_N \end{bmatrix}, \\ \boldsymbol{b} &= \begin{bmatrix} b_1 & b_2 & \cdots & b_n & \cdots b_N \end{bmatrix}^T \end{aligned} \tag{2.3}$$

are the learnable parameters of the network, *i.e.* $\boldsymbol{\theta}$; ϕ is the activation function applied element-wise; and $f^{(l)}$ represents the overall transformations performed by the l-th layer of the network.

Therefore, the output $\hat{\boldsymbol{y}}$ of an arbitrary L-layer feedforward neural network is a composition of functions (denoted as "$\circ$") applied to the input:

$$\hat{\boldsymbol{y}} = (f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)})(\boldsymbol{x}). \tag{2.4}$$

If every layer $f^{(l)}$, $l \in \{1, 2, \ldots, L\}$, is linear, the composition of linear functions can be reduced to a single linear function,

$$\hat{\boldsymbol{y}} = (f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(2)} \circ f^{(1)})(\boldsymbol{x}) = \boldsymbol{W}'^T\boldsymbol{x} + \boldsymbol{b}'. \tag{2.5}$$

Thus, a feedforward neural network made of linear neurons in arbitrary layers can always be described by a neural network with a single layer. This is not attractive because this network can only distinguish classes that are linearly separable, which limits the power of learning more complex functions. For this reason, it is desirable that each activation is a nonlinear function [15].

2.2 Commonly used activation functions

As mentioned, every activation takes a single value and performs a fixed mathematical operation on it. There are several functions commonly found in practice.

2.2.1 Sigmoid

The sigmoid is described as

$$\sigma(x) = \frac{1}{1+e^{-x}}, \tag{2.6}$$

and has the form shown in Fig. 2.3. It takes a real-valued number and maps it into a range from 0 to 1. The sigmoid function became very popular in the past because it has a plausible interpretation as the fire rate of a neuron: from not firing (0) to fully-saturated firing at an assumed maximum frequency (1). In practice, it has two major drawbacks:

- **Saturation:**. When the sigmoid approaches 0 or 1, its derivative is almost zero. This impacts the gradient computations required by the network optimization method are discussed later.
- **Output with non-zero mean**. Sequential processing of layers in a neural network works by receiving as inputs the activations from previous layers. A sigmoid will always produce non-negative activation values, and these biased inputs fed to the next layer slow down the network training [16].

2.2.2 Tanh

The hyperbolic tangent (represented by tanh) is also shown in Fig. 2.3, and looks like the sigmoid. Indeed, we can express the tanh as a scaled version of the sigmoid:

$$\tanh(x) = 2\sigma(x) - 1. \tag{2.7}$$

Like in the sigmoid neuron, the activation of the tanh neuron also saturates, but its output is zero-centered. This is why the tanh nonlinearity is often chosen in practice instead of the sigmoid.

2.2.3 ReLU

The rectifier linear unit (ReLU) has become very popular recently due to its successful use in different areas [17–19]. It computes the function

$$\phi(x) = \max(0, x), \tag{2.8}$$

and its shape is also shown in Fig. 2.3. The activation is a simple threshold at zero. Krizhevsky *et al.* [1] argued that due to its linear non-saturating form, the training speeds up significantly. Moreover, ReLU performs inexpensive operations compared to sigmoid or tanh. Unfortunately, this kind of activation has a drawback that gained the name of “die ReLU”: if the ReLU learns a large negative bias term for its weights, no matter the inputs, the output will be zero. Besides that, the gradient will also be zero, meaning that the ReLU unit will stay at zero indefinitely.

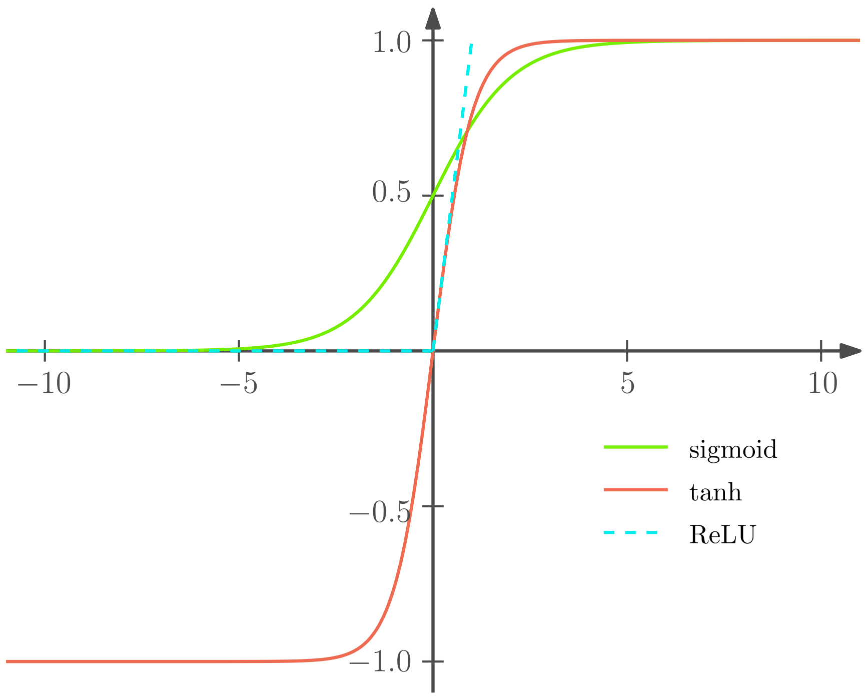


Figure 2.3: The most common nonlinear activation functions in neural networks.

2.3 Loss function

Our objective is to best approximate our model $\hat{\boldsymbol{y}} = f(\boldsymbol{x}; \boldsymbol{\theta})$ to the desired model $\boldsymbol{y} = f^*(\boldsymbol{x})$. We can define a scalar loss function $\mathcal{L}(f(\boldsymbol{x}; \boldsymbol{\theta}), f^*)$, also known as cost function or error function, that measures how well $f(\boldsymbol{x}; \boldsymbol{\theta})$ approximates f^*. Low values indicate better approximations, while high values mean worse approximations. Then, the goal is to find the set of parameters $\boldsymbol{\theta}$ that minimizes the expected loss over the parameters

$$\min_{\boldsymbol{\theta}} \mathbf{E}[\mathcal{L}(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})]. \tag{2.9}$$

Unfortunately, the true data distribution $(\mathbf{x},\mathbf{y})$ is often unknown. What one usually has available is a finite set of fixed data points $(\boldsymbol{x}_i,\boldsymbol{y}_i)$ that is known as the training set $\mathbb{S}$. The solution is approximated by substituting the sample mean loss computed over the training set for the expected loss:

$$\min_{\boldsymbol{\theta}} \mathbf{E}[\mathcal{L}(f(\mathbf{x};\boldsymbol{\theta}),\mathbf{y})] \approx \min_{\boldsymbol{\theta}} \frac{1}{|\mathbb{S}|} \sum_{(\boldsymbol{x}_i,\boldsymbol{y}_i)\in\mathbb{S}} \mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{y}_i). \tag{2.10}$$

Due to the nonlinearity of the model, this minimization problem cannot be solved by the common techniques designed for convex optimization. Instead, a gradient-based method (which follows the steepest descent from each point over the loss hyper-surface) is often preferred. To perform the minimization, all computation in a neural network should be differentiable.

2.3.1 Two-layer feedforward neural network as a universal model for any continuous function

It has been shown that a feed-forward network with a single hidden layer with nonlinear activation containing a finite number of neurons can approximate any continuous function. This is the basic idea behind the universal approximation theorem[1], proved firstly by George Cybenko [13] in 1989 for one kind of activation function and generalized in 1991 by Kurt Hornik [14].

2.4 The forward pass

We have already seen the inner parts of a neural network – neuron, layers, and connections – and how to evaluate its performance, but we have not seen yet how those parts stick together and perform a real computation.

2.4.1 Toy example: Learning XOR

In this example, we will use a feedforward network to predict the outputs shown in Tab. 2.1 — the logic XOR.

The XOR function is the target function $y = f^*(\boldsymbol{x})$, and our goal is to correctly predict the four points $\mathbb{X} = \{[0,\ 0]^T,\ [0,\ 1]^T,\ [1,\ 0]^T, [1,\ 1]^T\}$. Our network provides a function $\hat{y} = f(\boldsymbol{x};\ \boldsymbol{\theta})$, and our learning algorithm will try to adapt the parameter $\boldsymbol{\theta}$ to make f closest to f^*.

[1]Proof of this theorem is beyond the scope of this work.

Table 2.1: Truth table that the neural network will try to predict.

Inputs		Output
0	0	0
0	1	1
1	0	1
1	1	0

We will treat this problem as a regression problem [20] and adopt a mean squared error (MSE) as the loss function to make the math easier. In practical applications, MSE is usually not an appropriate cost function for modeling binary data [15].

After evaluating the training set, the MSE loss is

$$\mathcal{L}(f(\boldsymbol{x};\boldsymbol{\theta}), y) = \frac{1}{4}\sum_{\boldsymbol{x}_i \in \mathbb{X}} (y_i - f(\boldsymbol{x}_i;\boldsymbol{\theta}))^2. \tag{2.11}$$

Now that we have a training set and a loss function, we can choose the form of our model, $f(\boldsymbol{x};\boldsymbol{\theta})$. Suppose that we pick a linear model, with $\boldsymbol{\theta}$ consisting of $\boldsymbol{w}$ and b, such that

$$f(\boldsymbol{x};\boldsymbol{w};b) = \boldsymbol{w}^T\boldsymbol{x} + b. \tag{2.12}$$

We can minimize our cost function $f(\boldsymbol{x};\boldsymbol{w},b)$ in closed form with respect to $\boldsymbol{w}$ and b. Doing some math, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$. The model simply outputs $\frac{1}{2}$ everywhere. This occurs because a linear model is not able to represent the XOR function, as demonstrated in Fig. 2.4. One way to overcome this problem is to use a model that can learn a different feature space in which a linear model can find a solution.

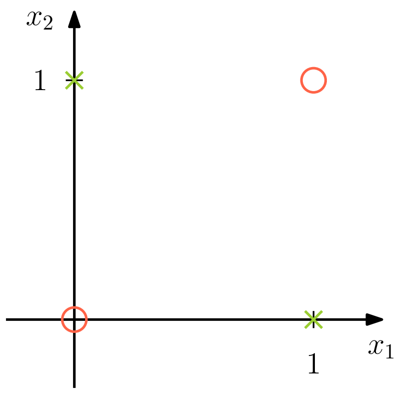


Figure 2.4: XOR is an example of function that cannot be learnt by a linear model, *i.e.* a straight line cannot separate the classes. This kind of function is also known as linearly non-separable function.

For instance, a 2-layer feedforward network with 2 hidden units (neurons) is capable of solving this kind of problem. The first layer will be responsible for mapping the input $\boldsymbol{x}$ into a new set of features $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}; \boldsymbol{b})$. The values of the hidden units are then used as input for the second layer. Hence, the second layer is the output layer of the network. The output layer is still just a linear model, but applied to $\boldsymbol{h}$ rather than $\boldsymbol{x}$. Mathematically, the network contains two functions chained together:

$$\boldsymbol{h} = \phi(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}), \tag{2.13}$$

$$\hat{y} = \boldsymbol{w}^T\boldsymbol{h} + c, \tag{2.14}$$

where $\boldsymbol{W} \in \mathbb{R}^{2\times 2}$, $\boldsymbol{b} \in \mathbb{R}^{2\times 1}$ are the parameters of the first (hidden) layer (containing 2 neurons) and $\boldsymbol{w} \in \mathbb{R}^{2\times 1}$, $c \in \mathbb{R}$ are the parameters of the second (output) layer. Our model is finally described as:

$$f(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}) + c. \tag{2.15}$$

As we wrote before, if we define the first layer as a linear model, then the feedforward network would remain a linear function of its input,

$$\begin{aligned} f(\boldsymbol{x}) &= \boldsymbol{w}^T(\boldsymbol{W}^T\boldsymbol{x}) + \boldsymbol{w}^T\boldsymbol{b} + c \\ &= \boldsymbol{w}'^T\boldsymbol{x} + b', \end{aligned} \tag{2.16}$$

where $\boldsymbol{w}' = \boldsymbol{w}^T\boldsymbol{W}^T$ and $b' = \boldsymbol{w}^T\boldsymbol{b} + c$.

To avoid this problem, ϕ must be a nonlinear function, as described in Sec. 2.2. One could choose ReLU — the default recommendation for modern neural networks [3]. Finally, our complete network is defined as

$$f(\boldsymbol{x}) = \boldsymbol{w}^T \max(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}, \boldsymbol{0}) + c. \tag{2.17}$$

Setting

$$\boldsymbol{W} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}, \qquad \boldsymbol{b} = [0\ 0]^T, \qquad \boldsymbol{w} = [1\ 1]^T, \qquad \boldsymbol{c} = [0\ 0]^T, \tag{2.18}$$

we can calculate the output of our model by propagating the information from input to output, also known as network forwarding. Let $\boldsymbol{X}$ be our batch input containing all four points in the binary input space:

$$\boldsymbol{X}^T = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} . \tag{2.19}$$

The first step is to compute the affine projection of the input[2]

$$(\boldsymbol{W}^T \boldsymbol{X})^T = \begin{bmatrix} 0 & 0 \\ -1 & 1 \\ 1 & -1 \\ 0 & 0 \end{bmatrix} ; \tag{2.20}$$

then, we calculate the value of $\boldsymbol{h}$ by applying the ReLU transformation. This non-linearity maps the input to a space where a linear model can solve this problem, as shown in Fig. 2.5. We finish our calculation by applying the last layer function to the last result:

$$\boldsymbol{w}^T \left(\boldsymbol{W}^T \boldsymbol{X} \right) = \begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix} . \tag{2.21}$$

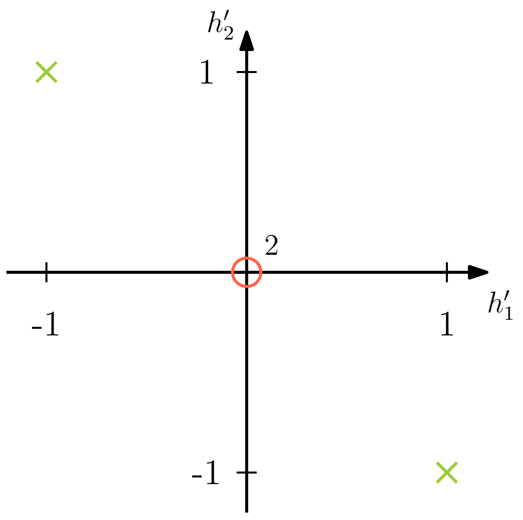
From this expression, the neural network has correctly predicted the right answer for every example in the batch.

In this example, we have demonstrated how the forward pass of a neural network works. In real applications, there might be millions of model parameters and millions of training examples, making it impossible to guess the solution as we did. Instead, a gradient-based optimization is preferred to find the parameters that produce a minimum error. To perform the gradient-based optimization, we must know the gradient of the loss with respect to each parameter. This is done by performing the backpropagation algorithm as discussed below.

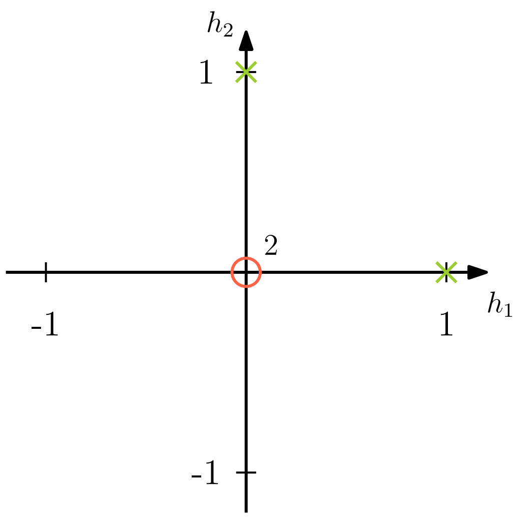
2.5 The backward pass

Backpropagation, also known as backprop, is a highly efficient algorithm responsible for the backward flow of information through a neural network, which will be used by a learning method. People use to associate backprop with the learning algorithm itself, but it is only a method for computing the gradients. Also, backprop is often misunderstood as a particular algorithm for neural networks, but it computes the derivative chain rules through any computational graph [3].

[2]The affine projections for all inputs $\boldsymbol{X} = [\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3, \boldsymbol{x}_4]$ are done at once.



(a) Inputs lying into a straight line after the affine projection performed in the first layer.



(b) After being transformed by the nonlinearity, the inputs lie in a space of features where a linear model can distinguish the correct output.

Figure 2.5: Transformation of nonlinear input space $\boldsymbol{x}$ into a linearly separable feature space $\boldsymbol{h}$.

Definition 2.1. The chain rule

It is worth remembering that the differentiation chain rule states for $z = f(y)$ and $y = g(x)$ that

$$\frac{\mathrm{d}z}{\mathrm{d}x} = \frac{\mathrm{d}z}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}x}, \tag{2.22}$$

which intuitively means that if $f(y)$ depends on y and y depends on x, we can calculate how $f(y)$ changes with x by multiplying how $f(y)$ changes with y and how y changes with x. This is useful in many applications, but turns out to be essential when analyzing and computing how gradients flow through a computational graph.

The chain rule can be extended to a broader context that includes vectors and matrices. Considering $g : \mathbb{R}^M \to \mathbb{R}^N$, $f : \mathbb{R}^N \to \mathbb{R}$, for $\boldsymbol{x} \in \mathbb{R}^M$, $\boldsymbol{y} \in \mathbb{R}^N$, and $z \in \mathbb{R}$, then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}. \tag{2.23}$$

We can denote it in vector form:

$$\frac{\partial z}{\partial \boldsymbol{x}} = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \frac{\partial z}{\partial \boldsymbol{y}}, \tag{2.24}$$

where

$$\frac{\partial z}{\partial \boldsymbol{y}} = \begin{bmatrix} \dfrac{\partial z}{\partial y_1} \\ \vdots \\ \dfrac{\partial z}{\partial y_n} \\ \vdots \\ \dfrac{\partial z}{\partial y_N} \end{bmatrix} \tag{2.25}$$

is the gradient of z with respect to $\boldsymbol{y}$ and

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial y_1}{\partial x_1} & \cdots & \dfrac{\partial y_1}{\partial x_m} & \cdots & \dfrac{\partial y_1}{\partial x_M} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \dfrac{\partial y_n}{\partial x_1} & \cdots & \dfrac{\partial y_n}{\partial x_m} & \cdots & \dfrac{\partial y_n}{\partial x_M} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \dfrac{\partial y_N}{\partial x_1} & \cdots & \dfrac{\partial y_N}{\partial x_m} & \cdots & \dfrac{\partial y_N}{\partial x_M} \end{bmatrix} \tag{2.26}$$

is the $N \times M$ Jacobian matrix of g.

2.5.1 Recursively applying the chain rule to obtain backprop

With the differentation chain rule, generating an algebraic expression for the gradient of a scalar w.r.t. any input is straightforward. Consider a scalar function $f : \mathbb{R}^N \to \mathbb{R}$ composed by L functions $f = (f^{(L)} \circ \cdots \circ f^{(l)} \circ \cdots \circ f^{(1)})(\boldsymbol{x})$, where $f^{(l)} : \mathbb{R}^{N_l} \to \mathbb{R}^{N_{l+1}}$, $N_1 = N$, and $N_{L+1} = 1$. Applying the chain rule, we can obtain the gradient of f w.r.t its input $\boldsymbol{x} \in \mathbb{R}^N$ such that

$$\frac{\partial f}{\partial \boldsymbol{x}} = \left(\frac{\partial \boldsymbol{x}^{(L)}}{\partial \boldsymbol{x}} \right)^T \frac{\partial f^{(L)}}{\partial \boldsymbol{x}^{(L)}}, \tag{2.27}$$

where $\boldsymbol{x}^{(l)}$ is the input of $f^{(l)}$. Reapplying the chain rule,

$$\frac{\partial f}{\partial \boldsymbol{x}} = \left(\frac{\partial \boldsymbol{x}^{(L-1)}}{\partial \boldsymbol{x}} \right)^T \left(\frac{\partial \boldsymbol{x}^{(L)}}{\partial \boldsymbol{x}^{(L-1)}} \right)^T \frac{\partial f}{\partial \boldsymbol{x}^{(L)}}. \tag{2.28}$$

After repeating the same operation until $l = 1$, we have

$$\frac{\partial f}{\partial \boldsymbol{x}} = \prod_{l=1}^{L-1} \left(\frac{\partial \boldsymbol{x}^{(l+1)}}{\partial \boldsymbol{x}^{(l)}} \right)^T \frac{\partial f^{(L)}}{\partial \boldsymbol{x}^{(L)}} = \prod_{l=1}^{L-1} \left(\frac{\partial f^{(l)}}{\partial \boldsymbol{x}^{(l)}} \right)^T \frac{\partial f^{(L)}}{\partial \boldsymbol{x}^{(L)}}. \tag{2.29}$$

By applying the chain rule recursively, we can obtain the derivative of the output w.r.t its input. More than that, we can get the derivative of a complex function by computing several simpler derivatives and multiplying them. However, calculating the Jacobian matrices directly may not be efficient.

Consider for instance a linear model $f(\boldsymbol{x};\boldsymbol{W}) = \boldsymbol{W}^T\boldsymbol{x}$, where $\boldsymbol{W} \in \mathbb{R}^{M\times N}$ and $\boldsymbol{x} \in \mathbb{R}^M$. Finding the Jacobian matrix w.r.t. $\boldsymbol{x}$ is straightforward: $\partial f/\partial \boldsymbol{x} = \boldsymbol{W}^T$. Computing the Jacobian w.r.t. matrix $\boldsymbol{W}$ can be done by reshaping $\boldsymbol{W}$ into a vector

$$\boldsymbol{w} = \begin{bmatrix} W_{1,1} & W_{1,2} & \cdots & W_{1,n} & W_{2,1} & \cdots W_{M,N} \end{bmatrix} \in \mathbb{R}^{MN}, \tag{2.30}$$

resulting in a Jacobian matrix $\partial f/\partial \boldsymbol{w} \in \mathbb{R}^{MN\times N}$ with elements

$$\frac{\partial f_n}{\partial W_{i,j}} = \begin{cases} 0, & n \neq j \\ x_i, & n = j \end{cases}. \tag{2.31}$$

Since our objective is to compute the gradient of a scalar loss function $\mathcal{L}(f(\boldsymbol{x};\boldsymbol{W}), f^*)$ w.r.t. $\boldsymbol{W}$, we can use the chain rule

$$\frac{\partial \mathcal{L}}{\partial W_{i,j}} = \sum_{n=1}^{N} \frac{\partial f_n}{\partial W_{i,j}} \frac{\partial \mathcal{L}}{\partial f_n}. \tag{2.32}$$

Using Eq. (2.31), the gradient becomes much simpler:

$$\frac{\partial \mathcal{L}}{\partial W_{i,j}} = \frac{\partial f_j}{\partial W_{i,j}} \frac{\partial \mathcal{L}}{\partial f_j} = x_i \frac{\partial \mathcal{L}}{\partial f_j}, \tag{2.33}$$

which can be represented in a vectorized form

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{x} \frac{\partial \mathcal{L}}{\partial f}^T. \tag{2.34}$$

As in the example above, training neural networks only requires a scalar loss function. Analogously, for each parameter we only need to calculate the derivative of the output w.r.t. the parameter and multiply it with the gradient of the loss w.r.t. its outputs, avoiding a lot of additional operations.

2.5.2 Computational graph

A computational graph, or computational circuit, is an acyclical directed graph where each node is an operation (referred simply as op), or computation, that transforms its inputs into outputs (see Fig. 2.6). The operations in a graph can be defined arbitrarily. An easy example would be a graph with Boolean operations (logic gates), which we know as a logic graph or logic circuit.

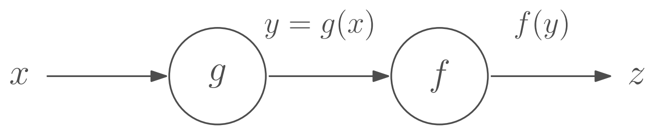
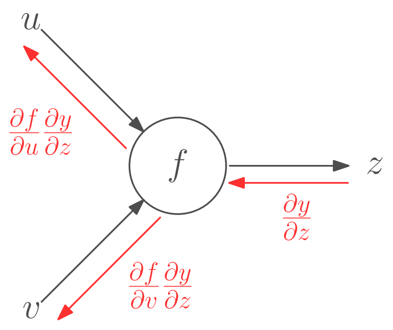


Figure 2.6: An example of computational graph. This graph implements the function $z = (f \circ g)(x)$.

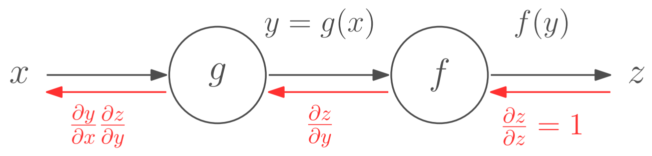
2.5.3 Applying the chain rule to a computational graph

The chain rule can be interpreted in a very interesting way: to calculate the gradient with respect to an input of an op, we simply need the local derivatives and the gradient of the graph's output, represented as y, with respect to the node's output. This concept is illustrated in Fig. 2.7 and an example is shown in Fig. 2.8.



u
$\frac{\partial f}{\partial u}\frac{\partial y}{\partial z}$
f
z
$\frac{\partial y}{\partial z}$
v
$\frac{\partial f}{\partial v}\frac{\partial y}{\partial z}$

Figure 2.7: Calculating the gradient in a computational graph. We only have to calculate the local gradient given the graph's output (y) gradient.



x
g
$y = g(x)$
f
$f(y)$
z
$\frac{\partial y}{\partial x}\frac{\partial z}{\partial y}$
$\frac{\partial z}{\partial y}$
$\frac{\partial z}{\partial z} = 1$

Figure 2.8: An example of backpropagation through the computational graph representation of Eq. (2.22).

Toy example: sigmoid function

As a concrete example, let us compute the derivative of the logistic sigmoid function defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.35}$$

The ops we use to define the graph are entirely up to us. We should define ops which we know the derivatives of. We have defined the graph in Fig. 2.9, and the values on the top of the arrows, in black, are computed by applying the operation to its inputs. The values on the bottom, in red, are the derivatives, calculated by the chain rule: the left term is the local gradient, which multiplies the propagated gradient, on the right.

Knowing some derivatives

$$\frac{\partial}{\partial x}\left(\frac{1}{x}\right) = -\frac{1}{x^2}, \qquad \frac{\partial(x \pm 1)}{\partial x} = 1, \qquad \frac{\partial e^x}{\partial x} = e^x, \qquad \frac{\partial(-x)}{\partial x} = -1, \tag{2.36}$$

the total derivative of the input w.r.t. the output is straightforwardly calculated by applying the chain rule throughout the graph:

$$\frac{\partial \sigma}{\partial x} = e^{-x}\sigma^2(x) = \sigma(x)[1 - \sigma(x)]. \tag{2.37}$$

It should be noted that, since $\frac{\partial \sigma(x)}{\partial \sigma(x)} = 1$, we always assume an initial gradient of 1.

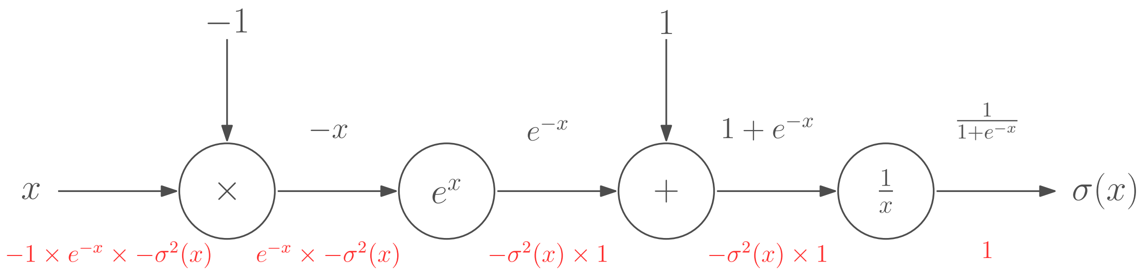


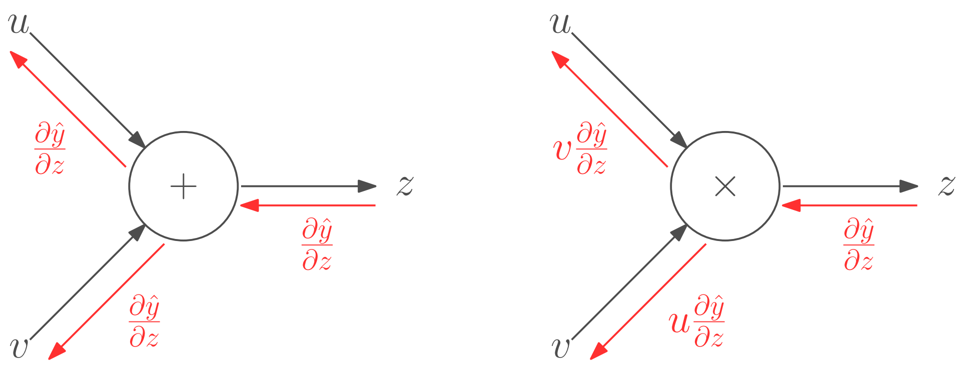
Figure 2.9: Computational graph of sigmoid. The values in black/red on the top/bottom of the arrows indicate the forward/backward calculations. Red arrows indicating the backprop flow are omitted for clarity.

Essential ops

As we saw in a simple example using the logistic function, we were able to calculate the gradients rather easily. That was possible only because we already knew the local gradients of each operation. This is the essential idea: each node changes how the gradient is propagated backwards into the graph; so, as long as we have performed the forward pass (obtain the op's output) and the backward pass (obtain the gradients with respect to its inputs given the gradients with respect to outputs), we can find any gradient desired. Fig. 2.10a shows the adding operation, which acts as a gradient distributor, and Fig. 2.10b shows the multiplying operation, which "swaps" the inputs.

2.5.4 Backpropping through a neural network

As we pointed before, a feedforward neural network can be viewed as an acyclic directional graph — the same way that we have defined a computational graph. We will use everything we have learned of the computational graph to backpropagate



u
v
z
$+$
$\frac{\partial \hat{y}}{\partial z}$
$\frac{\partial \hat{y}}{\partial z}$
$\frac{\partial \hat{y}}{\partial z}$
u
v
z
$\times$
$v\frac{\partial \hat{y}}{\partial z}$
$\frac{\partial \hat{y}}{\partial z}$
$u\frac{\partial \hat{y}}{\partial z}$

(a) Adding operation in a computational graph

(b) Multiplying operation in a computational graph

Figure 2.10: Essential basic operations that we must know in advance. The addition op works as a gradient distributor, while the multiplying op works as a "swap" input.

the signal through the network used in our toy example that predicted the XOR function (Sec. 2.4.1).

First, we need to define the backpropagation through a layer with an arbitrary number of neurons and no activation, which is represented in Fig. 2.11. We have for an arbitrary layer

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \boldsymbol{\delta}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = \boldsymbol{\delta}, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{x}\boldsymbol{\delta}^T, \qquad \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} = \boldsymbol{W}\boldsymbol{\delta}, \tag{2.38}$$

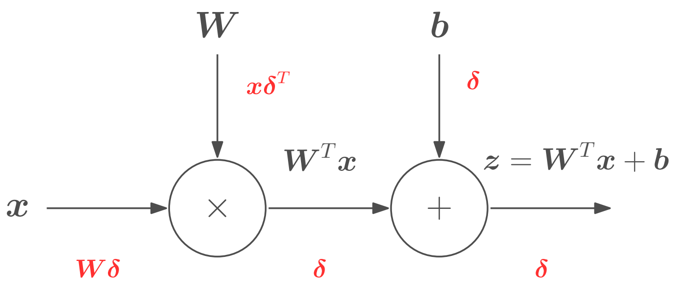
where $\boldsymbol{\delta}$ is the derivative of the loss w.r.t. the output of the computational graph. Then, we need to define the backpropagation through a layer with ReLU activation. Note that the derivative of operation $\phi(x) = \max(x,0)$ applied element-wise is

$$\frac{\partial \phi}{\partial x} = \begin{cases} 0, & \text{for } x \leq 0 \\ 1, & \text{for } x > 0, \end{cases} \tag{2.39}$$

i.e., the step function, u. Next, we can calculate the backprop of a ReLU layer, shown in Fig. 2.12, where $\odot$ denotes the element-wise multiplication (generalizing the multiplicative case for real variables in Fig. 2.10b).

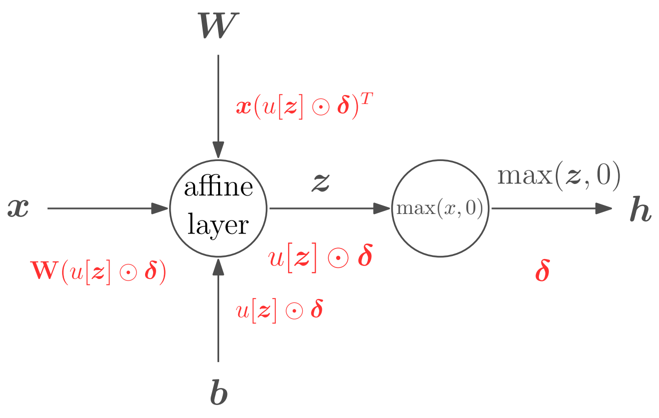
Finally, computing the gradients of the loss w.r.t. any variable of our XOR model (shown in Fig. 2.13) is straightforward:

$$\frac{\partial \mathcal{L}}{\partial y} = \delta, \quad \frac{\partial \mathcal{L}}{\partial c} = \delta, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}} = \boldsymbol{h}\delta, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} = u[\boldsymbol{z}] \odot (\boldsymbol{w}\delta), \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} = \boldsymbol{x}(u[\boldsymbol{z}] \odot (\boldsymbol{w}\delta))^T. \tag{2.40}$$



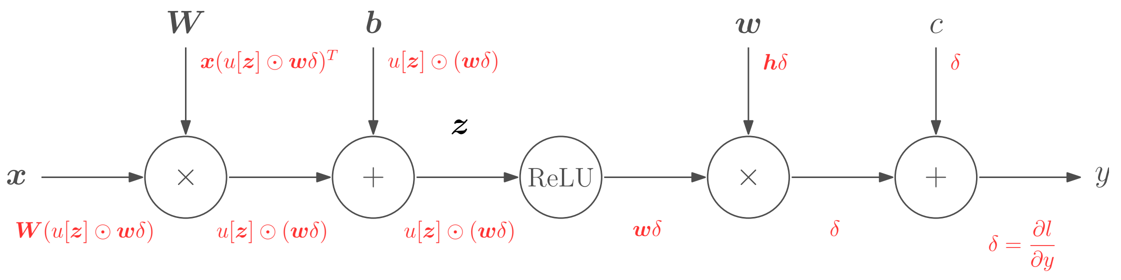
$\boldsymbol{W}$
$\boldsymbol{x}\boldsymbol{\delta}^T$
$\boldsymbol{b}$
$\boldsymbol{\delta}$
$\boldsymbol{W}^T\boldsymbol{x}$
$\boldsymbol{z} = \boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b}$
$\boldsymbol{x}$
$\times$
$+$
$\boldsymbol{W}\boldsymbol{\delta}$
$\boldsymbol{\delta}$
$\boldsymbol{\delta}$

Figure 2.11: Backprop through an affine layer with no activation.



$\boldsymbol{W}$
$\boldsymbol{x}(u[\boldsymbol{z}] \odot \boldsymbol{\delta})^T$
affine layer
$\boldsymbol{z}$
$\max(x,0)$
$\max(\boldsymbol{z},0)$
$\boldsymbol{h}$
$\boldsymbol{x}$
$\mathbf{W}(u[\boldsymbol{z}] \odot \boldsymbol{\delta})$
$u[\boldsymbol{z}] \odot \boldsymbol{\delta}$
$\boldsymbol{\delta}$
$u[\boldsymbol{z}] \odot \boldsymbol{\delta}$
$\boldsymbol{b}$

Figure 2.12: Backprop through a ReLU layer.



$\boldsymbol{W}$
$\boldsymbol{x}(u[\boldsymbol{z}] \odot \boldsymbol{w}\delta)^T$
$\boldsymbol{b}$
$u[\boldsymbol{z}] \odot (\boldsymbol{w}\delta)$
$\boldsymbol{w}$
$\boldsymbol{h}\delta$
c
δ
$\boldsymbol{z}$
$\boldsymbol{x}$
$\times$
$+$
ReLU
$\times$
$+$
y
$\boldsymbol{W}(u[\boldsymbol{z}] \odot \boldsymbol{w}\delta)$
$u[\boldsymbol{z}] \odot (\boldsymbol{w}\delta)$
$u[\boldsymbol{z}] \odot (\boldsymbol{w}\delta)$
$\boldsymbol{w}\delta$
δ
$\delta = \frac{\partial l}{\partial y}$

Figure 2.13: Backprop through the XOR model.

2.6 Deep feedforward neural networks

Stacking more than one hidden layer in a feedforward network sounds a bit awkward because of the universal approximation theorem. As we said, the theorem states that an MLP with one hidden layer with a finite number of neurons can approximate any function, but the theorem does not give an upper bound for the number of neurons. It has been noticed that the number of the neurons in the hidden layer grows exponentially with the complexity of the function that must be learned. The solution, which has an analogy with circuit theory, is stacking more than one layer to turn the solution more tractable in the same way that a multiplier works in a circuit [21]. The rationale behind constructing deeper models is that the first layers will be responsible for learning simple concepts (*e.g.* detecting edges in an image) while deeper layers will learn complex concepts (*e.g.* detecting faces in an image).

Practical experiments have shown that deeper models provide better results than the shallow ones when one has a large number of features at its inputs (*e.g.* raw pixels from an image, speech, and video). However, as every real-life problem, stacking more layers did not work well at first.

2.6.1 Why did stacking layers not work?

In fact, this solution works. In the 1990s and until the beginning of the 2000s, researchers did not have enough computation power to fine-tune the hyperparameters in a viable time.

2.6.2 Unsupervised learning as a bootstrap of deeper models

In 2006, Geoffrey Hinton *et al.* [22], discovered that training an unsupervised generative model called Restricted Boltzmann Machine[3], transferring the learning weights to a feedforward network, and then fine tuning the model with the labeled data enabled stacking more layers. They observed that a better weight initialization makes deeper models able to train. This was the beginning of the deep learning era.

2.6.3 Xavier and He initialization

After that, researchers have noticed that further study was needed to understand why using an unsupervised way of weight initialization worked better than random initialization with the old backpropagation. One of these studies was made by Xavier Glorot and Yoshua Bengio in 2010 [24]. In short, they observed two phenomena:

1. If the weights start too small, then the signal shrinks as it passes through the layers until it is too little to be useful;
2. If the weights start too large, then the signal grows as it passes through the layers until it is too massive to be useful.

Based on those observations, they proposed the Xavier initialization, which ensures that the weights are "just right", keeping the signal within a reasonable range through many layers.

Math behind Xavier

Considering a linear neuron, we will have at any layer[4]

$$h = \boldsymbol{w}^T \boldsymbol{x} = w_1 x_1 + \ldots + w_n x_n + \ldots + w_N x_N. \tag{2.41}$$

[3]More about how it works can be found in [23].

[4]The bias term was ignored to simplify the equations.

Let us denote x_n, w_n, and h the random variables associated to x_n, w_n, and h, respectively. Glorot and Bengio observed that the variance

$$\mathbf{var}(\mathrm{h}) = \mathbf{var}(\mathrm{w}_1\mathrm{x}_1 + \ldots + \mathrm{w}_n\mathrm{x}_n + \ldots + \mathrm{w}_N\mathrm{x}_N) \tag{2.42}$$

should be kept the same throughout the layers. Assuming that both weights and inputs are mutually independent and share the same distribution, we can compute

$$\mathbf{var}(\mathrm{h}) = \mathbf{var}(\mathrm{w}_1\mathrm{x}_1) + \ldots + \mathbf{var}(\mathrm{w}_n\mathrm{x}_n) + \ldots + \mathbf{var}(\mathrm{w}_N\mathrm{x}_N), \tag{2.43}$$

where

$$\mathbf{var}(\mathrm{w}_n\mathrm{x}_n) = \mathbf{E}[\mathrm{x}_n]^2\,\mathbf{var}(\mathrm{w}_n) + \mathbf{E}[\mathrm{w}_n]^2\,\mathbf{var}(\mathrm{x}_n) + \mathbf{var}(\mathrm{w}_n)\,\mathbf{var}(\mathrm{x}_n). \tag{2.44}$$

Now, if both inputs and weights have zero mean, Eq. (2.44) simplifies to

$$\mathbf{var}(\mathrm{w}_n\mathrm{x}_n) = \mathbf{var}(\mathrm{w}_n)\,\mathbf{var}(\mathrm{x}_n); \tag{2.45}$$

then,

$$\mathbf{var}(\mathrm{h}) = N\,\mathbf{var}(\mathrm{w}_n)\,\mathbf{var}(\mathrm{x}_n). \tag{2.46}$$

If we want to keep the variance between input and output the same, that means the variance of the weights should be

$$\mathbf{var}(\mathrm{w}_n) = \frac{1}{N} = \frac{1}{N_{\text{in}}}. \tag{2.47}$$

This is a simplification of the final formula found by Glorot and Bengio [24]. If we go through the same steps for the backpropagated signal, we find that

$$\mathbf{var}(\mathrm{w}_n) = \frac{1}{N_{\text{out}}} \tag{2.48}$$

to keep the variance between the input and output gradient the same. These two constraints are only satisfied simultaneously if $N_{\text{in}} = N_{\text{out}}$. Keeping a compromise, the authors defined

$$\mathbf{var}(\mathrm{w}_n) = \frac{2}{N_{\text{in}} + N_{\text{out}}}. \tag{2.49}$$

In the original paper, the authors made a lot of assumptions (*e.g.* linear neurons and both input and weights i.i.d), but it works. Xavier initialization, also termed Glorot initialization, was one of the big enablers of the move away from unsupervised pre-training.

The strongest assumption made was the linear neuron. It is true that the regions of the traditional nonlinearities considered at the time (tanh and sigmoid) that are explored just after initialization are close to zero, with gradients close to 1. However, for newer nonlinearities, like ReLU, that does not hold.

Tiny modification on behalf of rectified linear units

The ReLU activation has become essential for state-of-the-art neural network models. However, unfortunately, training deeper architectures even with Xavier initialization did not work well when the activations were changed to ReLU, due to the misbehavior around the origin. To surpass that, He *et al.* [19] published in 2015 a paper proposing a new "robust" initialization that mainly considers the ReLU nonlinearities. He *et al.* suggested using

$$\mathbf{var}(\mathrm{w}_n) = \frac{2}{N_{\text{in}}}, \tag{2.50}$$

instead. The rationale is: a rectifying linear unit is zero for half of its inputs, and to compensate for that, we need to double the variance of the weights to keep the variance of the signals constant.

This tiny correction had made possible training deeper models of up to 30 layers when the problems with Xavier initialization started to arise. However, at the same time, the authors have not observed the benefit from training extremely deep models. A 30-layer model was less accurate than a 14-layer model in image recognition. They supposed that either the method of increasing depth was not appropriate, or the recognition task was not complex enough. The performance degradation of very deep models was solved by using shortcut connections that create identity mappings that enable the training algorithm to focus on learning the residual mapping. This kind of network is known as residual network [25], and made possible to construct deep models of up to 1000 layers.

2.6.4 The internal covariate shift problem

Covariate shift refers to a change in the input distribution of a learning model. In deep neural networks, it means that the parameters of each layer below the layer's input change the input distribution. Even small changes can get amplified down to the network, and this effect increases with depth. This change of each layer's input distribution is known as internal covariate shift. It has been long known that the network training converges faster if its inputs are whitened. To address this problem, Sergey Ioffe and Christian Szegedy [26] proposed the batch normalization (BN) algorithm, often termed as batch norm.

Full whitening of each layer's input is too expensive, as it requires computing the covariance matrix and its inverse square root to produce the whitened activations, as well as the derivatives of those transforms for backpropagation. Hence, a normalization is made through each scalar of a D-dimensional input, $\mathbf{x} = [\mathrm{x}_1\ \mathrm{x}_2\ \cdots\ \mathrm{x}_d\ \cdots\ \mathrm{x}_D]^T$, as

$$\hat{\mathrm{x}}_d = \frac{\mathrm{x}_d - \mathbf{E}[\mathrm{x}_d]}{\sqrt{\mathbf{var}(\mathrm{x})}}. \tag{2.51}$$

We also do not want to change what the layer can represent. Therefore, we should make sure that the transformation inserted in the network can represent the identity transform. Ioffe and Szegedy introduced two new variables γ_d and β_d for each activation x_d, which scale and shift the normalized value:

$$\mathrm{y}_d = \gamma_d \hat{x}_d + \beta_d. \tag{2.52}$$

It is worth to mention that these parameters γ_d and β_d are learned jointly with the other parameters of the network, since γ_d and β_d are differentiable, and they can restore the representation of the network. Indeed, by setting $\gamma_d = \mathbf{var}(\mathrm{x}_d)$ and $\beta_d = \mathbf{E}[\mathrm{x}_d]$, we can restore the original activation, if the learning algorithm decides that it is the "best" for the loss.

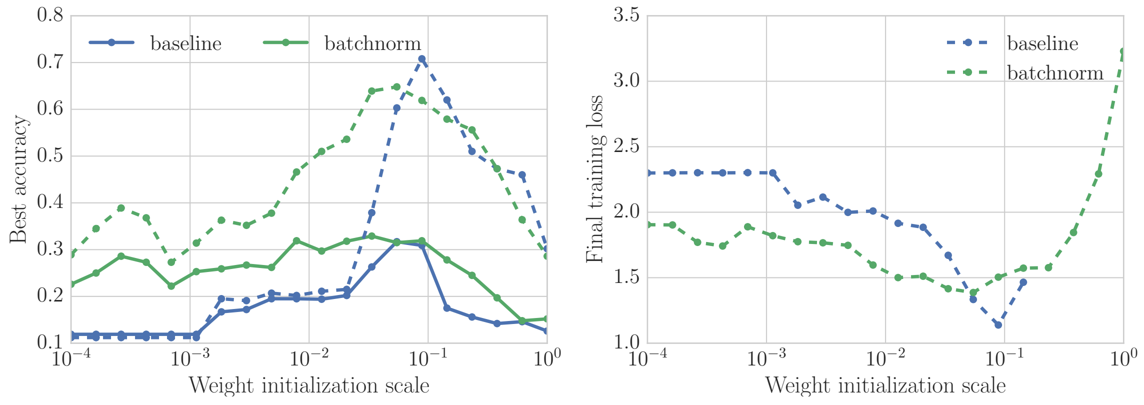
If the entire training set is used at each step of optimization, we might use it also to normalize the activations. However, we will see that this is impractical (Section 2.8.1). Since we use mini-batches in the gradient-based training, the statistics are re-computed at each mini-batch.

Obviously, in the test step (inference) the output must depend deterministically on the input; therefore, it employs statistical parameters that have been previously averaged during the training step.

In traditional deep networks, even with a careful initialization (using Xavier or He strategies), a very high learning rate may result in gradients that explode or vanish, as well as get stuck in poor minima. Batch normalization prevents small changes in the parameters to be amplified into large and suboptimal changes in the gradients [26]. Also, Ioffe and Szegedy [26] have shown that models with batch normalization are more robust to the weights initialization, and make the training faster. The former behavior is illustrated in Fig. 2.14.

Nowadays, batch normalization is essential for deep learning, especially for dense neural networks and convolutional neural networks (see Sec. 2.10). Unfortunately, batch normalization does not scale well for recurrent neural networks (Sec. 2.11) — where the internal covariate shift and vanishing gradients are more severe — but some research has been done [27, 28] to solve this problem.

We have learned how to initialize our deep neural network: if the activations are close to zero and the gradients close to one, we might choose Xavier initialization; if



(a) Accuracy (b) Loss

Figure 2.14: Example of a 6-layer MLP with ReLU activation trained to classify images. Batch normalization was added before each activation. Weights were initialized with samples from a Gaussian distribution with standard deviation equal to the values of x-axis. Dashed lines are the results obtained for the training set, while the solid line are the results for the validation set. Accuracy is the average percentage of examples that are correctly predicted.

we prefer ReLU as the nonlinear activation, we might choose He initialization. For both of them, we must use[5] batch normalization before each activation in order to accelerate the training.

However, even after enabling a deeper model to be trained, we can not assure that it will be better than the shallow one. Deeper models are more prone to overfitting and often do not generalize well. We can mitigate the first problem using one concept from machine learning, early stopping [20], or we may reduce both problems by using some regularization methods, also called regularizers, as addressed in the following section.

2.7 Regularization for deep learning

Regularization is the key to get a better model generalization and prevent overfitting. We can say for sure that there are hundreds of ways to regularize a neural network.

The best regularization method is to increase the training data. Sometimes, increasing the training data can be time demanding, pricey, or both. Instead of generating new real samples, we could generate fake data (*e.g.* flipping images or adding white Gaussian noise in speech), which is often termed as data augmentation.

Creating fake data is not applicable to every task. For example, it is difficult to generate new fake data for a density estimation task unless we have already solved

[5]This is not true if we are using a recurrent neural network.

the density estimation problem.

Even with a great amount of data augmentation, we still can have some problems with generalization: either the input (or augmented) data is not representative enough, or the learning algorithm finds large weights that can cause numerical overflow. Thus, adding one or more regularizers is pretty common in deep learning. All in all, the two regularization methods described below are the most common in the deep learning area [3].

2.7.1 Weight decay

Weight decay means an ℓ_2-norm penalty for the weights. The idea is to constrain the ℓ_2-norm of the weights (or to constrain the Frobenius norm if the weights are matrices), heavily penalizing peaky vectors and preferring smooth weights because peaky weights appear when the network is specializing to the training dataset (overfitting the network), what must be avoided. It can be implemented by adding an ℓ_2 constraint for every weight in the network to the total loss

$$\mathcal{L}_{\text{reg}}(\mathbb{S};\boldsymbol{\theta}) = \frac{1}{|\mathbb{S}|} \sum_{(\boldsymbol{x}_i,\boldsymbol{y}_i)\in\mathbb{S}} \mathcal{L}(f(\boldsymbol{x}_i;\boldsymbol{\theta}),\boldsymbol{y}_i) + \sum_l \frac{1}{2}\lambda \|\boldsymbol{W}^{(l)}\|_{\text{F}}^2, \tag{2.53}$$

where λ is a hyperparameter. Typically, we initially set this variable to zero, and increase it slowly at small steps. Usually, the weight decay is not applied to the biases of the network.

2.7.2 Dropout as an exponential ensemble of thinner networks

Dropout [29] is another regularization method that reduces overfitting, working by randomly dropping units (along with their connections) from the neural network at each training step. Dropout can be viewed as a layer

$$\boldsymbol{h} = \boldsymbol{m} \odot \boldsymbol{x}, \tag{2.54}$$

where $\boldsymbol{m}$ is a mask sampled from a Bernoulli distribution $\mathrm{P}(\mathrm{m}_i = 1) = p$ at every training step, and p is a hyperparameter which represents the probability of dropping out units. A probability of $p = 0.5$ is commonly adopted and seems to work well in a wide range of networks and tasks. Dropout works by preventing the co-adaptation [30] of units – which occurs when units are highly dependent on other units and are not useful without them. By randomly choosing its units, dropout ensures robustness against such co-adaptation, thus preventing overfitting.

Also, dropout can be viewed as an extreme case of an ensemble technique where several models are trained, and at test time the predictions of each model are averaged together. At each training step, the dropout regularizer samples from an exponential number of "thinner" networks that share some weights. The thinned networks consist of all units that survived from dropout. A neural net with n units can be seen as a collection of 2^n possible thinner networks, all sharing weights. At the test time, the whole network is used, and each dropout layer scales its input by a factor of p, *i.e.* $\boldsymbol{h} = p\boldsymbol{x}$, approximating the effect of the averaging predictions of all these thinned networks, ensuring that for each unit the expected output is the same as the actual output at test time. Indeed, in practical scenarios, it is always preferable to use the inverted dropout, which performs the scaling at training time, leaving the network at test time untouched.

2.8 Network optimization

The loss function is a measure that quantifies the quality of our model given any set of hyperparameters $\boldsymbol{\theta}$. The goal of optimization is to find $\boldsymbol{\theta}$ that minimizes the loss function. Unfortunately, this is not a trivial task. Due to the nonlinearities inside the neural network, the problem is not convex, and we can have millions of parameters to set, facing a hard computational problem to solve. Not all hyperparameters of the neural network can be learned automatically by a gradient-based algorithm, such as number of layers, number of hidden units (neurons), activation function, and so on. For this case, we need to make several design choices based on previous successful models or test several models with different parameters. Parameters that can be learned by a gradient-based algorithm, like weights, bias, and batch normalization parameters are called learnable parameters, or just weights.

We need to find a place in the weight space (see next section) where the loss is better than the previous loss. Due to the differentiability of the loss function w.r.t. the weights, we can find the steepest-descent direction in the weight space. The gradient tells us the direction in which the function reaches the steepest rate of increase, but it does not tell us how far along this direction we would step. Choosing the right step size (also called learning rate) is one of the most important hyperparameters that we will have to choose. If we pick a tiny step, we will for sure make progress, but small. On the other hand, if we choose a large step, thinking that the function will descend faster, we may find that it oversteps, reaching a higher loss than before.

2.8.1 Gradient descent

The procedure of repeatedly evaluating the gradient and then updating the parameters is called gradient descent [20]. The vanilla version looks like[6]

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}, \tag{2.55}$$

where L is the total loss evaluated over the training data given our model, and η is the learning rate. This simple update is at the core of every neural network optimization program, and the gradient descent is by far the most common way of optimizing the neural network (although there are other ways to perform this).

In large-scale applications, where we have training data on order of million images, it is wasteful to compute the gradients for the entire training set to perform only a single parameter update. A common approach is to calculate the gradient over mini-batches of the training data. In practice, this works very well, and the explanation behind this is that the training data are correlated by themselves. The gradient will not be the same, but it is a good approximation of the full objective's gradient, and the training gets a much faster convergence, due to the more frequent parameter updates.

In the extreme, we can adopt a mini-batch of one example. This process is called a stochastic gradient descent (SGD). This process is not commonly used because this does not take advantage of the vectorized code optimization of modern CPUs and GPUs. In fact, it can be much more computationally efficient to evaluate the gradient for a mini-batch of 128 examples than for a mini-batch of 1 example 128 times. Usually, as said, SGD applies to the evaluation of one example at a time, but the community uses SGD even when referring to the mini-batch gradient descent. The size of the mini-batch is also a hyperparameter, but it is usually constrained according to memory restrictions (*e.g.* to fit in your GPU memory), or set to some power of 2, like 32, 64 or 128, because many vector operations are accelerated in this way.

2.8.2 Momentum

Sometimes, learning with SGD can be a slow process. The method of momentum [31] was created to accelerate learning by introducing a new variable v, initialized at zero, that plays the role of velocity – the direction and velocity at which the parameters move through the parameter space. This can be accomplished by

[6] $\boldsymbol{\theta}$ here will be used here for simplicity to indicate the learnable parameters.

accumulating an exponentially decaying moving average of the past gradients

$$\begin{aligned} \boldsymbol{v} &\leftarrow \mu \boldsymbol{v} - \eta \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \end{aligned} \tag{2.56}$$

where the new hyperparameter $\mu \in [0,1)$ determines how quickly the contributions of previous gradients decay.

Nesterov momentum

A slight and yet powerful modification introduced by Sutskever [32] is to compute the update velocity on an estimate of future position $\hat{\boldsymbol{\theta}}$:

$$\begin{aligned} \hat{\boldsymbol{\theta}} &\leftarrow \boldsymbol{\theta} + \mu \boldsymbol{v} \\ \boldsymbol{v} &\leftarrow \mu \boldsymbol{v} - \eta \frac{\partial \mathcal{L}}{\partial \hat{\boldsymbol{\theta}}} \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \boldsymbol{v}. \end{aligned} \tag{2.57}$$

This minor modification works better than the standard momentum in practice.

2.8.3 Adaptive methods

Machine learning practitioners have long realized that the learning rate is one of the most difficult hyperparameters to be defined, because it significantly affects the model performance. The momentum-based algorithm tries to mitigate this issue, but at the cost of adding another hyperparameter.

Much work has gone to create methods that can adaptively tune the learning rates, ideally per parameter and with addition of no new hyperparameter. Unfortunately, this goal has not been achieved yet. Adaptive methods were created to overcome the first problem and still require that hyperparameters are set, but are arguably more reliable for a broader range of values than the raw learning rate.

RMSProp

RMSProp is a powerful adaptive learning rate method that has not been published yet[7]. It uses a moving average of squared gradients,

$$\begin{aligned} \boldsymbol{m} &\leftarrow \alpha \boldsymbol{c} + (1-\alpha) \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \right)^2 \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \frac{1}{\sqrt{\boldsymbol{m}} + \epsilon} \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}, \end{aligned} \tag{2.58}$$

[7]People usually refer to [33], slide 29 of lecture 6 of Geoffrey Hinton's Coursera class.

where cache variable $\boldsymbol{m}$ keeps track of a per-parameter smoothed sum of squared gradients, α is a hyperparameter whose typical values are $\{0.9, 0.99, 0.999\}$, and ϵ is a small value that avoids division by zero. The feeling behind RMSProp is that parameters that receive high gradients will have their true learning rate reduced, whereas parameters that receive small updates will have their true learning rate increased.

Adam

Adam [34], or ADAptive Moment estimation, is one of the most recent adaptive learning rate methods proposed. It looks like the RMSProp update, but the first order momentum is used instead of the possibly noisy gradient. The (simplified) algorithm looks like

$$\begin{aligned}
\boldsymbol{v} &\leftarrow \alpha_v \boldsymbol{v} + (1-\alpha_v)\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \\
\boldsymbol{m} &\leftarrow \alpha_m \boldsymbol{m} + (1-\alpha_m)\left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}\right)^2 \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \eta \frac{\boldsymbol{v}}{\sqrt{\boldsymbol{m}}+\epsilon},
\end{aligned} \tag{2.59}$$

where $\alpha_v \in [0,1)$ and $\alpha_m \in [0,1)$ are hyperparameters, and $\epsilon = 10^{-8}$, $\alpha_v = 0.9$, and $\alpha_m = 0.99$ are the recommended values [34].

Adam usually works slightly better than RMSProp and is the recommended algorithm to use when we are searching for the hyperparameters, because Adam is faster than Nesterov. The most common approach is: searching the best model configuration (tuning the hyperparameters) with Adam and training the best model with SGD + Nesterov.

2.9 Sharing weights across time and space

A fully connected network does not scale well to full images. If we take into account that we are interested in applications with images of medium size in an RGB channel ($256 \times 256 \times 3$), and we apply this input into a single fully-connected neuron, we will have $256 \cdot 256 \cdot 3 = 196{,}608$ weights. However, we would almost certainly want to have several such neurons, and the number of parameters will add up quickly. Clearly, the number of parameters would quickly lead to overfitting (and we want to avoid this).

Besides that, fully connected networks are not very adequate for modeling sequences, such as temporal ones. As we saw, the basic principle in an MLP is modeling a function that transforms the input data into the desired output. This limits their applicability in cases where inputs are part of a sequence, and their order implies correlation between them and, most important, between the outputs. A common example of a temporal sequence is a sound signal: samples (or even small sequences of samples) often contain little meaning by themselves, and to truly understand the data the network must be able to analyze not only the current sample but its relationship to past inputs. Commonly adopted solutions to adapt neural networks to sequences include the usage of windows to the sequence into chunks, which are then used as input for the network. This has some limitations, such as:

- Input lengths are fixed, which limits the amount of temporal information the network can take into account.
- The model does not naturally adapt to different input sizes.
- The model cannot, by itself, learn how long it should remember things since it has no memory and the available temporal information depends on the chosen window size, which is set by a human expert.
- At each window, it completely forgets information about previous windows, which might be relevant.

To address these problems, two kinds of networks were invented: convolutional neural network and recurrent neural network.

2.10 Convolutional networks

Convolutional neural networks (CNNs) [35], often termed as ConvNets, have proved very effective as feature extractors [36, 37]. They have been successful in object recognition [25, 38], instance segmentation [39–42], identifying faces [43–45], and powering vision of self-driving cars [5]. Nowadays, ConvNet is one of the most powerful tools for every machine learning practitioner.

Unlike MLPs, ConvNets take advantage of the grid-like topology of the input. They expect and preserve the spatial relationship between neighbors (*e.g.* pixels in images) by learning internal feature representation using small filters.

A typical convolutional network has three stages: convolutional layer, nonlinear function, and pooling layer.

2.10.1 Convolutional layer

The convolutional layer is the core element of the convolutional network. Differently from a fully connected (affine) layer, it takes spatial structure into account.

Not only that, it also employs a great deal of weight sharing, which makes possible to work with large inputs (such as images) with a reasonable amount of parameters.

The convolutional layer owes its name to the operation it performs: convolution. The layer input is no longer a vector, but a 3-D tensor that has C channels, height H and width W. It has, therefore, dimensions $(C \times H \times W)$. The layer has sets of weights called filters, each one with dimensions $(C \times H_f \times W_f)$. The convolution process is characterized by sliding (convolving) the filter across the input, performing at each position an inner product through which we obtain a scalar called activation. If we organize activations according to the position of the filter w.r.t. the image, we obtain an activation map with dimensions $(H_{\text{out}} \times W_{\text{out}})$.

The activation map dimensions depend on the input size, the filter size, and three hyperparameters: stride, padding, and zero-padding. Fig. 2.15 illustrates the convolution procedure. When stride is one, we move the filters one sample at a time. A larger stride produces smaller output volumes spatially. If we applied one convolution layer after another with stride greater than one, the spatial size would decrease at each layer until vanishing out. Zero-padding is used to address this issue: we add a border of zeros to the input just for keeping the output with the same input size after convolution. Typical choices are filters with small kernel sizes (3×3 or 7×7) with stride 2 and zero-padding.

Each convolution between the input and the filter gives rise to an activation map; therefore, if there are F filters, the output has dimension $(F \times H_{\text{out}} \times W_{\text{out}})$. Note that all filters will have the same dimensions, and share the same depth of the input.

Each filter can be trained to look for specific patterns in the input and, when convolved with the input, is expected to produce an activation map that retains spatial information regarding that feature in the image. The number of filters we use represents how many different patterns we are looking for in the input.

Typically, after each convolutional layer, a nonlinear function, such as ReLU, is applied. This stage (convolutional layer + nonlinear function) is sometimes called detection stage.

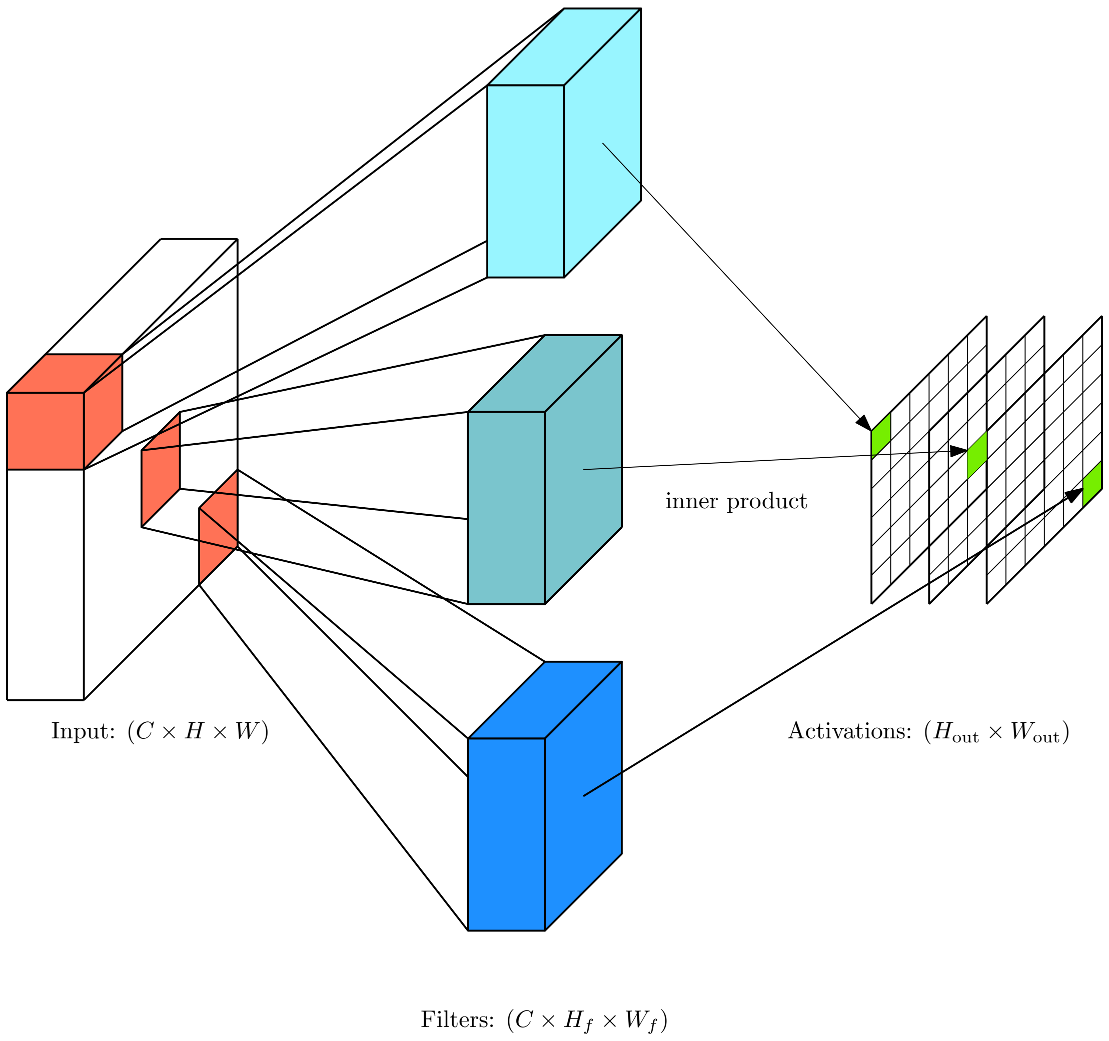

inner product
Input: $(C \times H \times W)$
Activations: $(H_{\text{out}} \times W_{\text{out}})$
Filters: $(C \times H_f \times W_f)$

Figure 2.15: Example of convolutional layer. Three kernels (filters) of size $C \times H_f \times W_f$ are applied to the input, resulting in three activations maps of size $H_{\text{out}} \times W_{\text{out}}$. A single convolution is shown for each filter.

2.10.2 Pooling layer

In a typical ConvNet architecture, it is common to periodically insert a pooling layer in between successive convolutional layers.

The pooling layer reduces the spatial size, reducing the number of parameters of the network, and controls the overfitting. Pooling also helps to make the representation approximately invariant to small translations of the input. Pooling is also defined by a learnable parameter contained in the kernel. Unlike a convolutional layer, the pooling layer has as hyperparameters only the filter size and the stride. It only operates independently on every depth slice of the input. Its most common form is a pooling layer with filters of size 2×2 applied with a stride of 2, and performs the max operation over the activations, reducing the spatial size by 4. The depth dimension remains unchanged. Intuitively, the max pooling layer retains the strongest feature detected. Average pooling is another widely used pooling layer.

An example is shown in Fig. 2.16.

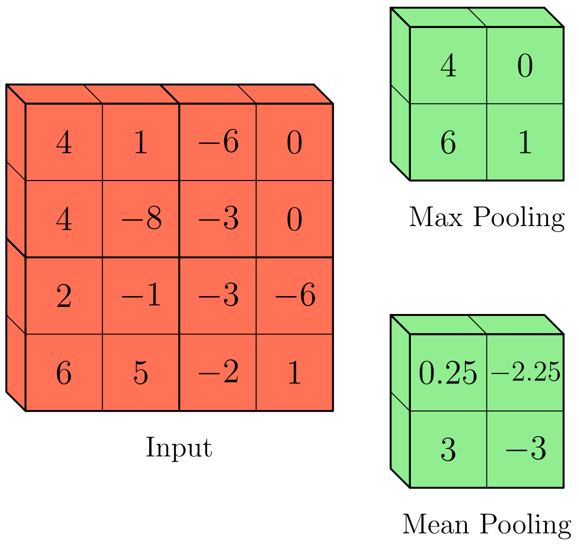


Figure 2.16: Example of pooling layer. The filter has size of $C \times 2 \times 2$ where C is the number of channels of the input, with stride 2. Only the first channel values are shown.

2.10.3 ConvNet architectures

Commonly, ConvNet architectures are composed of several blocks of convolutional layers + batch norm + nonlinear activation with periodical insertions of pooling layer between successive blocks. In the end, a fully connected layer or a global average pooling [46] is appended to generate the output (*e.g.* predictions of the class of an image). There are several hyperparameters to be set like number of layers, filter size of each layer, and stride. There is no default recipe for building a ConvNet model.

Over the years, several models have been proposed and became popular. LeNet [2] was the first of its kind, and AlexNet [1] won by a large margin the ImageNet challenge — the first time that deep learning surpassed other ML algorithms. VGG [47] is an "old" model and yet has been widely used due to its simplicity. ResNets [25] have allowed training models with depth up to 1000 layers, becoming the state-of-the-art in several visual recognition tasks and are the default choice for real applications.

2.11 Recurrent neural networks

Recurrent Neural Networks (RNNs) were created in the 1980s but have just been gaining recent popularity with the advances of deep learning and with the increasing computational power of graphical processing units. One of most attractive features of RNNs is that, in theory, they can learn correlations between samples in a sequence, unlike feedforward neural networks. RNNs have been used as the state-of-the-art model for machine translation [48–50], language modeling [51, 52] and many others.

For simplicity, we refer to RNNs as operating over a sequence that contains vectors $\boldsymbol{x}^{(t)} \in \mathbb{R}^D$ with the time step index t ranging from 1 to T. In practice, RNNs operate on mini-batches of such sequences, with a different sequence length T for each member of the mini-batch. We have omitted the mini-batch indices to simplify notation. To better understanding about recurrent networks, one could read the textbook of Graves [53].

2.11.1 Vanilla RNNs

We can model sequences with recurrent neural networks, whose computation depends not only on the present input but also on the current state of the network, which keeps information on the past. They allow us to operate over sequences of vectors: sequence at the input, at the output, or both. Fig. 2.17 shows a few examples of what RNNs can model.

In general, an RNN can be defined by two main parts:

1. A function that maps the input and the previous state into the current state;
2. A function that maps the current state into the output.

On a vanilla RNN, these functions are represented by

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{W}_{xh}^T \boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}^T \boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h) \tag{2.60}$$

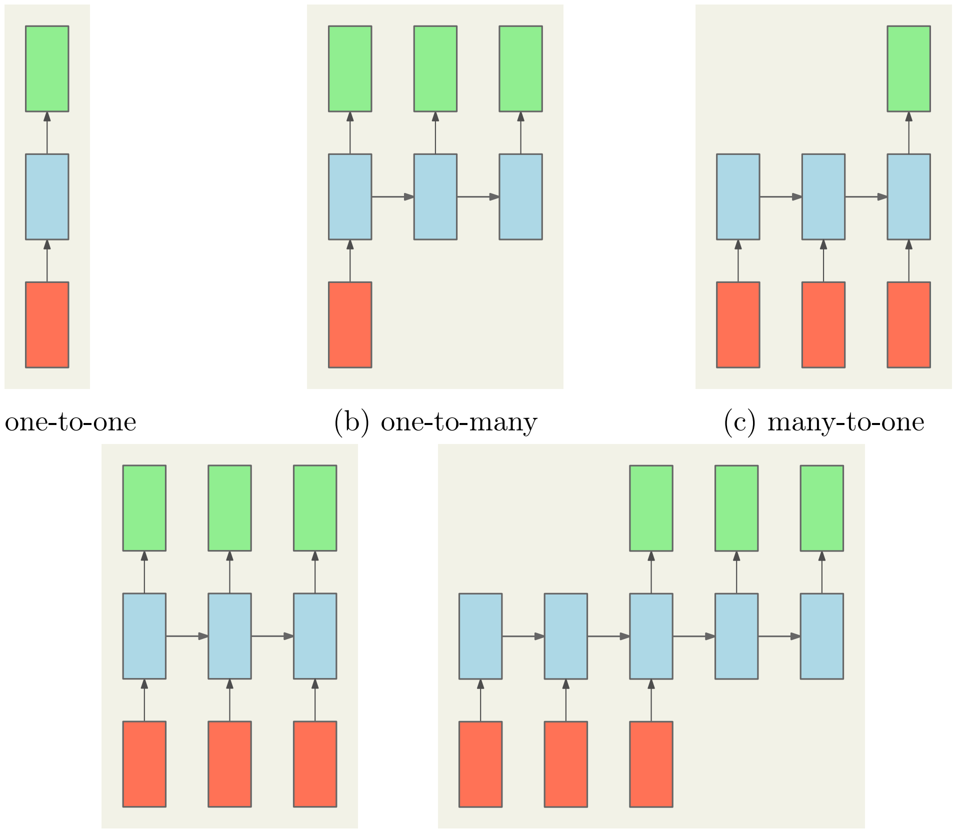
$$\boldsymbol{y}^{(t)} = \boldsymbol{W}_{hy}^T \boldsymbol{h}^{(t)} + \boldsymbol{b}_y, \tag{2.61}$$

where $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$ are the input state and the output at time step t, respectively, $\boldsymbol{W}_{u,v}$ contains the weights that map vector $\boldsymbol{u}$ in vector $\boldsymbol{v}$, and $\boldsymbol{b}_u$ is the bias term of vector $\boldsymbol{u}$. Notice that the same set of weights is applied at each time step, sharing the weights across different sequences.

2.11.2 Bidirectional RNNs

The RNN presented has a causal structure, meaning that the output at time t only depends on the information from the past and the current time.

In many applications, however, the output at time t may depends on future samples. In speech recognition, for example, the correct prediction of the current phoneme may rely on the next few phonemes due to co-articulation. In character prediction, we may need the information of both future and past context because of linguistic rules. This is also true for many other sequence-to-sequence applications.



(a) one-to-one (b) one-to-many (c) many-to-one

(d) many-to-many (e) many-to-many

Figure 2.17: Following the convention of the feedforward network, inputs are in red, outputs are in green, and RNN states are in blue. From left to right: (a) Vanilla neural network without RNN; fixed input size and fixed output size (*e.g.* image classification). (b) Sequence at the output (*e.g.* image captioning, where the input is an image, and the output is a variable-size sentence of words. (c) Sequence at the input (*e.g.* sentiment analysis, where a given sentence at the input is classified as expressing some sentiment). (d) Synced sequences at the input and the output (*e.g.* video classification, where we wish to label every frame).(e) Sequence at the input and sequence at the output (*e.g.* Machine translation, where a sentence of words at the input is translated to another sentence of words in another language). Images adapted from Karpathy's blog[8].

Bidirectional RNNs (BRNNs) aim to address that issue [54]. They combine two RNNs, one that moves forward through time, beginning from $\boldsymbol{x}^{(0)}$, and another that moves backward through time, starting from $\boldsymbol{x}^{(T)}$. Fig. 2.18 illustrates a bidirectional RNN. This allows output $\hat{\boldsymbol{y}}$ to depend on both the past and the future, by concatenating the forward hidden state $\vec{\boldsymbol{h}}^{(t)}$ and the backward hidden state $\overleftarrow{\boldsymbol{h}}^{(t)}$.

Although BRNNs have been extremely successful in several tasks, such as speech recognition [55–57] and handwriting recognition [58], they cannot be applied in real-time applications. Since the output at any time step depends on all future sequences, the information at time t will be available only when all the sequence is processed.

[8]`http://karpathy.github.io`

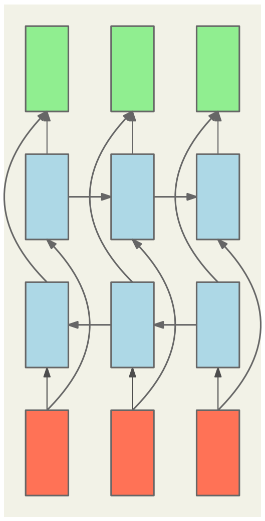


Figure 2.18: Example of a BRNN, which can be understood as two distinct RNNs, one fed at positive time steps (from 1 to T) and the other reversely fed (from T to 1). The output, however, depends on the concatenation of the two hidden states.

2.11.3 Back propagation through time

Despite its apparent recursive nature, we can analyze an RNN as a feedforward network whose depth evolves in time. In order to perform backpropagation, we set a depth to stop unrolling the network and propagate the gradients from there, as shown in Fig. 2.19. As we see from Fig. 2.20, the gradients w.r.t the current state result from both its output $\boldsymbol{y}^{(t)}$ and another gradient that flows through the state on the next step. Since the weights and biases are all shared between different time steps, the total gradient is the summation of the gradients at each time step. The backward computation is, therefore,

$$\begin{gathered}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_y} = \sum_t \frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(t)}}, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hy}} = \sum_t \boldsymbol{h}_t \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^{(t)}}\right)^T, \\
\boldsymbol{\alpha}^{(t)} = [1-(\boldsymbol{h}^{(t)})^2] \odot \left[\boldsymbol{W}_{hy}\frac{\partial \mathcal{L}}{\partial \boldsymbol{y}^{(t)}} + \boldsymbol{\delta}_t\right], \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hx}} = \sum_t \boldsymbol{x}_t \left(\boldsymbol{\alpha}^{(t)}\right)^T, \quad \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_{hh}} = \sum_t \boldsymbol{h}^{(t-1)}(\boldsymbol{\alpha}^{(t)})^T, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_h} = \sum_t \boldsymbol{\alpha}^{(t)}, \quad \boldsymbol{\delta}^{(t-1)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t-1)}} = \boldsymbol{W}_{hh}\boldsymbol{\alpha}^{(t)}, \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}_t} = \boldsymbol{W}_{xh}\boldsymbol{\alpha}^{(t)}.
\end{gathered} \tag{2.62}$$

2.11.4 Vanishing and exploding gradients in recurrent nets

As we saw previously, the RNN was created to overcome the limitations of the vanilla neural network in modeling sequences. One of the claims of RNNs is the

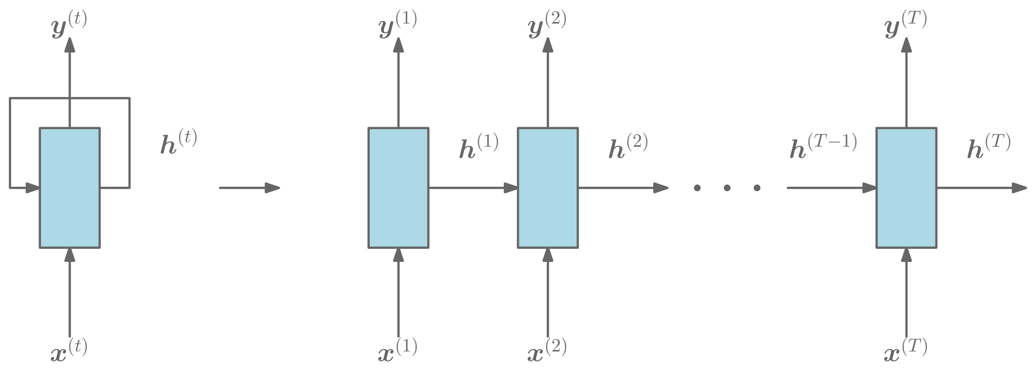


Figure 2.19: Unrolling the RNN in order to calculate the gradients.

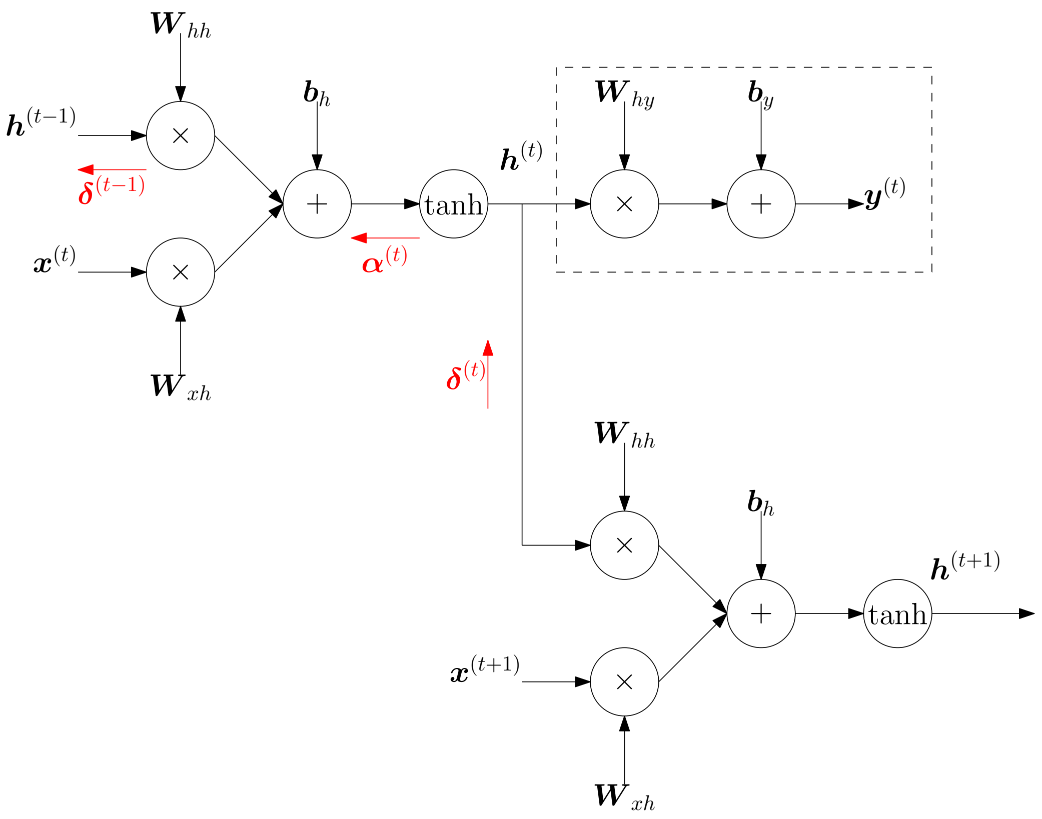


Figure 2.20: Computational graph of an RNN.

idea that they might be able to connect prior information to the present task. One might wonder how much context they are actually able to remember.

Sometimes, we only need to look at recent information to perform the present task. For example, consider a language model trying to predict the next word based on the previous ones. If we seek to predict the last word in "the clouds are in the **[last word]**", we do not need any further context — it is quite probable that the next word is going to be "sky". In such situations, where the gap between the relevant information and the place where it is needed is small, RNNs can learn to use the preceding information.

However, there also circumstances where we require more context. Consider trying to predict the last word in the subsequent text:

"I grew up in Teresópolis, Brazil. I studied [...] I am working [...] I speak fluent **[last word]**."

Recent information ("I speak fluent") suggests that the next word has a high probability of being the name of a language, but if we want to discover which language, we need the context of Brazil, from further back. We can imagine several situations where the gap between relevant information and the point where it is needed becomes very significant. In such cases, RNNs become unable to learn to connect the information, due to vanishing and exploding gradients caused by repeated matrix multiplications [59].

2.11.5 Dealing with exploding gradients: clipping

A simple but highly effective method for dealing with exploding gradient is to rescale the norm of gradient $\boldsymbol{\delta}$ whenever it goes over a threshold. If $\boldsymbol{\delta} \geq$ threshold,

$$\boldsymbol{\delta} \leftarrow \frac{\text{threshold}}{\|\boldsymbol{\delta}\|}\boldsymbol{\delta}. \tag{2.63}$$

Besides being simple, this technique is very efficient computationally, but it introduces an additional hyperparameter, the threshold. The value of the threshold is arbitrary, commonly set as 1, 5 or 10.

2.11.6 Dealing with vanishing gradients: LSTM

Long short-term memory networks – usually just called LSTMs – are a special kind of RNN, capable of learning long-term dependencies. They were introduced almost two decades ago [60] and several different implementations of LSTMs have been proposed.

In the vanilla LSTM, at each time step we receive an input $\boldsymbol{x}^{(t)}$ and the previous hidden state $\boldsymbol{h}^{(t-1)}$; the LSTM also maintains an H-dimensional cell state, so we also receive the previous cell state $\boldsymbol{c}^{(t-1)}$. The relationship between the hidden state, the cell state, and the inputs is controlled by four gates. The learnable parameters of LSTM are an input-to-hidden matrix $\boldsymbol{W}_x \in \mathbb{R}^{D\times 4H}$, a hidden-to-hidden matrix $\boldsymbol{W}_h \in \mathbb{R}^{H\times 4H}$, and a bias vector $\boldsymbol{b} \in \mathbb{R}^{4H}$.

At each time step, we first compute an activation vector $\boldsymbol{a} \in \mathbb{R}^{4H}$ as $\boldsymbol{a} = \boldsymbol{W}_x^T\boldsymbol{x} + \boldsymbol{W}_h^T\boldsymbol{h}^{(t-1)} + \boldsymbol{b}$. We then divide it into four vectors $\boldsymbol{a} = [\boldsymbol{a}_i, \boldsymbol{a}_f, \boldsymbol{a}_o, \boldsymbol{a}_g]^T$, for $\boldsymbol{a}_\star \in \mathbb{R}^H$. We then compute the input gate $\boldsymbol{i} \in \mathbb{R}^H$, the forget gate $\boldsymbol{f} \in \mathbb{R}^H$, the output gate $\boldsymbol{o} \in \mathbb{R}^H$, and the block input $\boldsymbol{g} \in \mathbb{R}^H$ as

$$\boldsymbol{i} = \sigma(\boldsymbol{a}_i), \qquad \boldsymbol{f} = \sigma(\boldsymbol{a}_f), \qquad \boldsymbol{o} = \sigma(\boldsymbol{a}_o), \qquad \boldsymbol{g} = \tanh(\boldsymbol{a}_g), \tag{2.64}$$

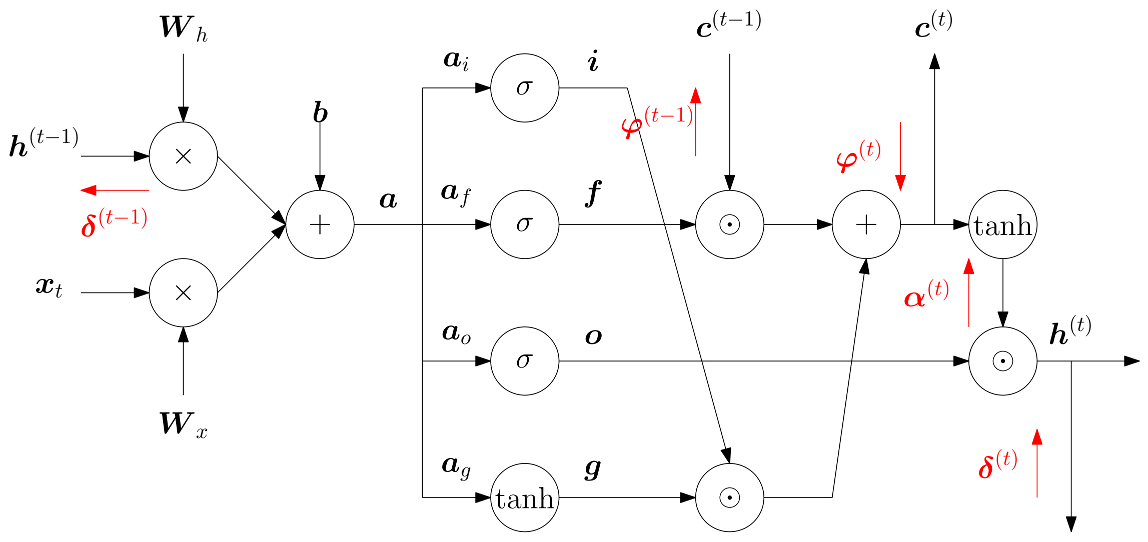


Figure 2.21: Computational graph of LSTM.

where σ is the sigmoid function, applied elementwise. Finally, we compute the next cell state $\boldsymbol{c}^{(t)}$ and next hidden state $\boldsymbol{h}^{(t)}$ as

$$\boldsymbol{c}^{(t)} = \boldsymbol{f} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i} \odot \boldsymbol{g} \tag{2.65}$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{o} \odot \tanh(\boldsymbol{c}^{(t)}). \tag{2.66}$$

The key to LSTMs is the cell state, which runs straight down the entire chain, with only minor linear interactions. The LSTM does have the capacity to remove or add information to the cell state, carefully regulated by the four structures called input gate, forget gate, output gate, and block input.

The first step in our LSTM is to decide what information we are going to throw away from the cell state. This decision is made by the sigmoid layer called forget gate $\boldsymbol{f}$. It looks at $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{x}^{(t)}$, and outputs a number between 0 and 1 for each number in the cell state. A 1 will keep the state unchanged, and 0 will completely erase the state. The next step is to select what new information we are going to store in the cell state. This has two parts. First, the input gate $\boldsymbol{i}$ determines which values we will update. Next, a tanh layer (called block input $\boldsymbol{g}$) creates a vector of new candidate values for $\boldsymbol{c}^{(t)}$ that could be added to the state. In the next step, we combine these two to create an update to the state. Finally, the current hidden state is based on our current cell state (after a tanh operation to map the values between -1 and 1), but filtered by the output gate $\boldsymbol{o}$. The computational graph for all these operations is shown in Fig. 2.21.

Backpropagation through the LSTM

As we saw in vanilla RNN, we can analyze it as a feedforward network whose depth evolves in time. In order to perform backpropagation on the LSTM, we set a depth to stop unrolling the network and propagate the gradients from there. As we see from the graph (Fig. 2.21), the gradients w.r.t. the current hidden state come from both its output $\boldsymbol{y}^{(t)}$ (not shown) and a gradient that flows through the state on the next step, and the gradients w.r.t. the current cell state come from its previous cell state. Since the weights and biases are all shared between different time steps, the total gradient is the summation of the gradients in each time step. The backward computation is, therefore,

$$
\begin{aligned}
\boldsymbol{\alpha}^{(t)} &= \left[\mathbf{W}_{hy}\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(t)}} + \boldsymbol{\delta}^{(t)}\right] \odot \boldsymbol{o} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t)}} &= \left[1 - (\tanh \boldsymbol{c}^{(t)})^2\right] \odot \boldsymbol{\alpha}^{(t)} + \boldsymbol{\varphi}^{(t)} \\
\boldsymbol{\varphi}^{(t-1)} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t-1)}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t)}} \odot \boldsymbol{f} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{i}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t)}} \odot \boldsymbol{g} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{f}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t)}} \odot \boldsymbol{c}^{(t-1)} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} &= \left[\mathbf{W}_{hy}\frac{\partial \mathcal{L}}{\partial \mathbf{y}^{(t)}} + \boldsymbol{\delta}^{(t)}\right] \odot \tanh \boldsymbol{c}^{(t)} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{g}} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}^{(t)}} \odot \boldsymbol{i} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_i} &= \boldsymbol{i} \odot (1 - \boldsymbol{i}) \odot \frac{\partial \mathcal{L}}{\partial \boldsymbol{i}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_f} &= \boldsymbol{f} \odot (1 - \boldsymbol{f}) \odot \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_o} &= \boldsymbol{o} \odot (1 - \boldsymbol{o}) \odot \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_g} &= (\mathbf{1} - \boldsymbol{g}^2) \odot \frac{\partial \mathcal{L}}{\partial \boldsymbol{g}},
\end{aligned}
\tag{2.67}
$$

where $\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}} = \left[\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_i}; \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_f}; \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_o}; \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_g}\right]$. Finally,

$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}} &= \sum_t \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_h} &= \sum_t \boldsymbol{h}^{(t-1)} \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}}\right)^T \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_x} &= \sum_t \boldsymbol{x}^{(t)} \left(\frac{\partial \mathcal{L}}{\partial \boldsymbol{a}}\right)^T \\
\boldsymbol{\delta}^{(t-1)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}^{(t-1)}} &= \boldsymbol{W}_h \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}} \\
\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}^{(t)}} &= \boldsymbol{W}_x \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}}.
\end{aligned} \tag{2.68}
$$

2.11.7 LSTM and beyond

After the original LSTM was introduced in 1997, several variations have been proposed. Forget gates, as described, were introduced by Gers *et al.* [61] and are in the standardized model presented by Graves [53] (which we called vanilla LSTM) due to its effectiveness. It is worth mentioning some other variations.

LSTM with peepholes

Gers *et al.* [61] developed one of the most popular LSTM variants, adding what they called peephole connections. This means that we let the gate layers look at the cell state. Mathematically, we create a new learnable parameter $\boldsymbol{W}_c \in \mathbb{R}^{H \times 4H}$ and

$$\boldsymbol{a} = \boldsymbol{W}_x^T \boldsymbol{x}^{(t)} + \boldsymbol{W}_h^T \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_c^T \boldsymbol{c}^{(t)} + \boldsymbol{b}. \tag{2.69}$$

The peephole connections improve the LSTM's ability to learn tasks that require precise timing and counting of internal states [53]. Note that the peephole connection was added to all gates, but this is not strictly necessary.

GRU

Another LSTM variation couples the forget and input gates. Instead of making the decisions separately, whether forget something or add new information to the cell state, we should make this decision together. That is, we only add new information to the state when we forget something older. This means that

$$\boldsymbol{c}^{(t)} = \boldsymbol{f} \odot \boldsymbol{c}^{(t-1)} + (\boldsymbol{1} - \boldsymbol{f}) \odot \boldsymbol{g}. \tag{2.70}$$

A simpler modification of the LSTM that shares many of the same properties is the gated recurrent unit, firstly used by Cho *et al.* [48]. It combines the forget and input gates into a single update gate $\boldsymbol{u}$ and also merges the hidden state with the cell states, among some other changes. The model requires fewer parameters than the vanilla LSTM.

At each time step, the activation vector $\boldsymbol{a} \in \mathbb{R}^{2H}$ is computed as $\boldsymbol{a} = \boldsymbol{W}_x^T \boldsymbol{x} + \boldsymbol{W}_h^T \boldsymbol{h} + \boldsymbol{b}$. We then divide it into two vectors $\boldsymbol{a} = [\boldsymbol{a}_u,\ \boldsymbol{a}_r]^T$, for $\boldsymbol{a}_\star \in \mathbb{R}^H$, and compute the reset gate $\boldsymbol{r} \in \mathbb{R}^H$ and the update gate $\boldsymbol{u} \in \mathbb{R}^H$ as

$$\boldsymbol{r} = \sigma(\boldsymbol{a}_r), \qquad \boldsymbol{u} = \sigma(\boldsymbol{a}_u). \tag{2.71}$$

Finally, we compute the next hidden state $\boldsymbol{h}^{(t)}$ as

$$\begin{aligned} \tilde{\boldsymbol{h}}^{(t)} &= \tanh\left[\boldsymbol{W}_{x\tilde{h}}^T \boldsymbol{x}^{(t)} + \boldsymbol{W}_{h\tilde{h}}^T (\boldsymbol{r} \odot \boldsymbol{h}^{(t-1)}) + \boldsymbol{b}_{\tilde{h}}\right] \\ \boldsymbol{h}^{(t)} &= (1 - \boldsymbol{u}) \odot \boldsymbol{h}^{(t-1)} + \boldsymbol{u} \odot \tilde{\boldsymbol{h}}^{(t)}, \end{aligned} \tag{2.72}$$

where $\boldsymbol{W}_{x\tilde{h}} \in \mathbb{R}^{D\times H}$, $\boldsymbol{W}_{h\tilde{h}} \in \mathbb{R}^{H\times H}$, and $\boldsymbol{b} \in \mathbb{R}^H$ also are learnable parameters of GRU.

Intuitively, the reset gate determines how to blend the information between the new input and the previous memory, and the update gate determines how much of the previous memory will be kept. If we set the reset gate to 1 and the update gate to 0, we will have the vanilla RNN model. This simpler model has been proved better than the LSTM for some tasks, reducing the number of parameters and hence being less prone to overfitting.

Which is better?

Several different models have been proposed over the years, like depth gated RNN [62], recurrent highway network [63], and clockwork RNN [64] (which takes a different approach to the long-term dependencies). Although the presented models deal with remembering the past by creating a hidden state, they do not have an explicit memory. Some progress has been made on creating an explicit memory block, such as neural Turing machine [65], and end-to-end memory networks [66].

There are a plenty of LSTMs and GRU variations. Indeed, it seems that every paper that uses a recurrent neural network makes its own adaptation to achieve its goal. Greff *et al.* [67] presented a study of eight most popular LSTM variants and found that they are all about the same. Jozefowicz *et al.* [68] went beyond, using a genetic algorithm to vary the building blocks of RNN, generating more than 10,000 different architectures. They found that some architectures are better than others in certain tasks. They also discovered that initializing the LSTM's forget gate to one closes the gap between the LSTM and GRU.

All in all, machine learning practitioners avoid using RNN due to the gradient problems. LSTM and GRU, the more common recurrent networks, can be used interchangeably, if the LSTM's forget bias is initialize to one. Other recurrent networks are highly experimental and used only in specific contexts.

2.12 On the difficulty of regularizing recurrent nets

Several papers have pointed the problems on training RNNs [59, 69–71]. Recurrent neural networks are more prone to overfitting, have several issues regarding generalization, and have the problem of vanishing and exploding gradients. LSTM, which introduces a way of dealing with vanishing gradient, only alleviates it. Clipping gradient, a simple trick of dealing with exploding gradient, seems to be a rough method.

Most of the regularization methods were developed without considering recurrent connections and long-term dependency. Regularizing an RNN by applying too much ℓ_2 penalty to the weights could work against long-term dependencies [59]. Dropout, one of the most powerful techniques for regularization of deeper models, is not very effective for RNN [70]. Indeed, dropout in recurrent weights generates noise that is multiplied over the time steps, exploding its gradients. The internal covariate shift is also a common problem inside RNN due to its non zero-centered nonlinearities, and direct application of BatchNorm is still an issue [27, 28].

Techniques like the injection of white noise into the training data seem to work well when modeling acoustic data [53, 72]. Applying noise to the weights has also been demonstrated effective under some scenarios [55]. Although weight and gradient noise are analogous when using standard SGD methods, using adaptive learning methods breaks this equivalence, and promising results have been made on applying the latter on recurrent networks [73].

Besides all research that have been done over the years, an efficient method for regularization of recurrent networks is still necessary. In the next five sections, we go through some recent advances on this topic.

2.12.1 Dropout for RNNs

Bayers (2013) *et al.* [74] argued that conventional dropout does not work well with recurrent networks because the recurrence amplifies the noise, which is harmful to the training. Zaremba, Sutskever, and Vinyals (2015) [70] applied the dropout only in non-recurrent connections, preventing noise amplification. They showed promising results in language modeling, speech recognition, machine translation, and image captioning. In the following year, Gal *et al.* [75] have developed a mathematics ground for dropout in RNN models. Previous techniques consisted of different dropout masks at each time step for inputs and outputs alone. Their proposed method, called variational dropout, applies the same mask at each time step for inputs, outputs, and recurrent layers. Their theoretically motivated method outperformed existed methods and improved the state-of-the-art model in language modeling.

2.12.2 Zoneout

Zoneout [76] slightly resembles dropout, but it was specifically designed for recurrent networks. Instead of masking some units' activation as in dropout, the zoneout method randomly replace some units' activation with the activation from the previous time step. Mathematically,

$$\boldsymbol{h}^{(t)} = \boldsymbol{m}^{(t)} \odot \boldsymbol{h}^{(t-1)} + (\mathbf{1} - \boldsymbol{m}^{(t)}) \odot \tanh(\boldsymbol{W}_{xh}^{T}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}^{T}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h), \qquad (2.73)$$

where $\boldsymbol{m}^{(t)}$ is the mask sampled at time step t over a Bernoulli distribution with probability p. Krueger *et al.* [76] argued that zoneout preserves the information flow forward and backward through the time step, unlike the dropout approach.

2.12.3 Batch norm for RNNs

Recurrent neural networks are hard to train and difficult to parallelize. Batch norm (BN) aims to significantly reduce training time and can act as a regularizer [26]. Unfortunately, applying BN to RNNs is not a trivial task. Laurent *et al.* (2015) [27] showed that using BN to hidden-to-hidden weights does not seem to speed up the training. Applying BN to input-to-hidden weights could lead to faster convergence, but it seems not to affect the generalization. Cooijmans *et al.* (2016) [28] noticed that RNNs usually are deeper in the time steps[9], and BN would be most beneficial when applied horizontally — throughout the time steps (hidden-to-hidden weights). In the case of LSTM, they applied BN on the recurrent term $\boldsymbol{W}_h^T\boldsymbol{h}^{(t-1)}$, on the input term $\boldsymbol{W}_x^T\boldsymbol{x}^{(t)}$ and on the cell state when updating the hidden state. They kept the statistics over each BN independently for each time step. Albeit they presented promising results, this strategy requires more computational power when we are dealing with long sequences (*e.g.* speech), precluding some applications; furthermore, if we have longer sequences in test time than in training time, we will not have the statistics for those longer sequences. Also, BN does not work if we adopt a batch size of one.

[9]We usually have much more time steps to unroll than stacked layers.

2.12.4 Layer normalization

To overcome some of the BN issues, Ba *et al.* [77] proposed the layer normalization (LN). Instead of normalizing through the batch, they suggested normalizing all inputs to the neurons in a layer on a single training case. Unlike BN, layer norm performs the same computation at training and test times. They presented empirical results, showing that LN can reduce the training time when compared with BatchNorm for RNN. In the case of vanilla RNN, layer norm re-centers and re-scales its activation by performing

$$\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{W}_{xh}^{T}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}^{T}\boldsymbol{h}^{(t-1)} \\
\boldsymbol{h}^{(t)} &= \tanh\left[\frac{\boldsymbol{\alpha}_{\mathrm{LN}}}{\sigma^{(t)}} \odot (\boldsymbol{a}^{(t)} - \mu^{(t)}) + \boldsymbol{\beta}_{\mathrm{LN}}\right] \\
\mu^{(t)} &= \frac{1}{H}\sum_{h=1}^{H} a_h^{(t)} \\
\sigma^{(t)} &= \sqrt{\frac{1}{H}\sum_{h=1}^{H}(a_h^{(t)} - \mu^{(t)})^2},
\end{aligned} \tag{2.74}$$

where H denotes the number of hidden units in a layer, and $\boldsymbol{\alpha}_{\mathrm{LN}}$ and $\boldsymbol{\beta}_{\mathrm{LN}}$ are defined as the gain and bias learnable parameters, respectively. Ba *et al.* [77] recommended initializing the gain with zero and the bias with a vector of ones. As we can notice, layer norm does not lay any restriction over the batch size.

2.12.5 Multiplicative integration

Another simple and yet very powerful modification was made by Wu *et al.* [78], called multiplicative integration (MI). Instead of describing the activation of recurrent units by a sum, they proposed to use the Hadamard product. For a vanilla RNN

$$\boldsymbol{a}^{(t)} = \boldsymbol{\alpha}_{\mathrm{MI}} \odot (\boldsymbol{W}_{xh}^{T}\boldsymbol{x}^{(t)}) \odot (\boldsymbol{W}_{hh}^{T}\boldsymbol{h}^{(t-1)}) + \boldsymbol{\beta}_{\mathrm{MI}} \odot (\boldsymbol{W}_{hh}^{T}\boldsymbol{h}^{(t-1)}) + \boldsymbol{\gamma}_{\mathrm{MI}} \odot (W_{xh}^{T}\boldsymbol{x}^{(t)}) + \boldsymbol{b}, \tag{2.75}$$

where $\boldsymbol{\alpha}_{\mathrm{MI}}$, $\boldsymbol{\beta}_{\mathrm{MI}}$, and $\boldsymbol{\beta}_{\mathrm{MI}}$ are hyperparameters. The effect of multiplication results in a gating type structure, where $\boldsymbol{W}_{xh}^{T}\boldsymbol{x}^{(t)}$ and $\boldsymbol{W}_{hh}^{T}\boldsymbol{h}^{(t-1)}$ are gates to each other. These changes enjoy better gradient properties due to the gating effect, and MI gives better generalization and it is easier to optimize. They have shown empirical results demonstrating the effectiveness of this method.

2.12.6 General discussion

All these recently proposed regularization methods have demonstrated their effectiveness through empirical results. However, strategies mixing them have not been deeply studied yet.

Chapter 3

All-neural speech recognition

Deep learning is conquering its place as the state-of-the-art in several applications, such as image classification [25], instance segmentation [79], machine translation [49], and speech recognition [80]. In image task areas, deep learning methods are applied directly to raw input images, outperforming several engineering features, such as the use of histogram of oriented gradient (HOG) for image detection [81].

Unfortunately, the same cannot be said of the speech recognition area. Automatic speech recognition has had significant advances with the introduction of the neural network and unsupervised learning; however, it has been applied only as a single component in a complex pipeline. The first step of the pipeline is the input feature extraction: standard techniques include mel-scale filter banks (and an optional transformation into cepstral coefficients) and speaker normalization techniques. Neural networks are then trained to classify individual frames of acoustic data, and their output distributions are fed as emission probabilities for an HMM. The loss function used to train the network is different from the true performance measure (sequence-level transcription accuracy). This is exactly the sort of inconsistency that end-to-end learning want to avoid. Over the years, researchers found that a significant gain in frame accuracy could not translate to transcription accuracy; in fact, it could even degrade the performance. Thus, building state-of-the-art ASR systems remains a complicated, expertise-intensive task (dictionaries, phonetic questions, segmented data, GMM models to obtain initial frame-level labels, multiple stages with different feature processing techniques, an expert to determine the optimal configurations of many hyperparameters, and so on). Why not developing a system applying neural networks as a single pipeline?

3.1 Traditional speech recognizers

An automatic speech recognition system aims at transcribing an utterance in a word sequence $\boldsymbol{W}^*$, as summarized in Fig. 3.1. The raw waveform from a microphone is converted into a sequence of fixed-size acoustic vectors $\boldsymbol{X}$ in a process called feature extraction. Then, the decoder tries to find the word sequence $\boldsymbol{W}$ that is most likely to have generated $\boldsymbol{X}$. This could be done by maximizing the probability of the word sequence given the utterance, $\mathrm{P}(\boldsymbol{W}|\boldsymbol{X})$. Unfortunately, finding $\underset{\boldsymbol{W}}{\operatorname{argmax}}\,\mathrm{P}(\boldsymbol{W}|\boldsymbol{X})$ is a difficult task. One way to discover this posterior probability is to transform the problem using the Bayes rule

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmax}} \frac{\mathrm{P}(\boldsymbol{X}|\boldsymbol{W})\,\mathrm{P}(\boldsymbol{W})}{\mathrm{P}(\boldsymbol{X})}, \tag{3.1}$$

where the likelihood $\mathrm{P}(\boldsymbol{X}|\boldsymbol{W})$ is called the acoustic model, the prior $\mathrm{P}(\boldsymbol{W})$ is the language model, and $\mathrm{P}(\boldsymbol{X})$ is the observation. If one chooses the maximum likelihood criterion to decode the word sequence, $\mathrm{P}(\boldsymbol{X})$ may be assumed constant for all $\boldsymbol{X}$ and can be ignored. In traditional ASR systems, the acoustic model represents the likelihood of phone sequence instead of word sequence. Therefore, we may write

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmax}} \frac{\mathrm{P}(\boldsymbol{X}|\boldsymbol{F})\,\mathrm{P}(\boldsymbol{F}|\boldsymbol{W})\,\mathrm{P}(\boldsymbol{W})}{\mathrm{P}(\boldsymbol{X})}, \tag{3.2}$$

where $\boldsymbol{F}$ is the phone sequence, and $\mathrm{P}(\boldsymbol{F}|\boldsymbol{W})$ is termed pronunciation model.

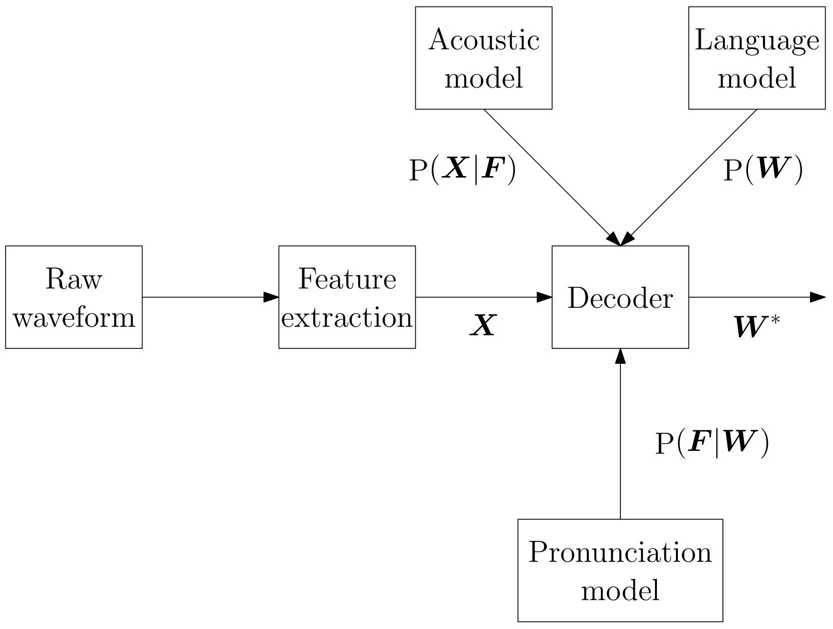


Figure 3.1: Block-diagram for the ASR problem.

In the ASR scheme depicted in Fig. 3.1, the feature extraction is responsible for providing a fixed-size compact representation $\boldsymbol{X}$ of speech waveform. Several algorithms have been used over the years. Linear predictive coding (LPC), perceptual linear predictive (PLP), log filter banks, and mel-frequency cepstral coefficients are the more common. A good feature extractor is one that provides a good match with the distribution assumptions made by the input of an acoustic model. The preprocessing can be (optionally) followed by one or more normalization schemes, such as the mean and variance normalization (to cancel some of the speaker variation) and the vocal tract length normalization (to cancel the difference between male and female speakers) [82, 83].

Usually, a Gaussian mixture model is responsible for emitting the probabilities of a phone given the acoustic vectors to an HMM model, which is responsible for modeling the possible transitions at each time step and allows the occurrence of multiple pronunciations (provided by the pronunciation model $\mathrm{P}(\boldsymbol{L}|\boldsymbol{W})$).

The language model is typically an n-gram model [84] — where the probability of each word depends only on its $(n-1)$ predecessors — trained in a separated text *corpus* with millions (even billions) of texts.

The decoder combines the acoustic scores $\mathrm{P}(\boldsymbol{X}|\boldsymbol{F})$ produced by the HMM with the language model prior $\mathrm{P}(\boldsymbol{W})$ and performs a search through all possible word sequences using a clever algorithm to pruning weak hypothesis thereby keeping the search tractable. When the end of utterance is reached, the most likely word sequence $\boldsymbol{W}^*$ is output. Recent algorithms use a compact trellis of possible hypothesis and perform the decodification with the Viterbi algorithm [9].

Instead of training with the maximum likelihood criterion, discriminative training criteria are more common in state-of-the-art HMM-based models, such as maximum mutual information [85, 86] (MMI), minimum classification error [87] (MCE), and minimum Bayes' risk [88, 89] (MBR), which try to relate directly the problem of minimizing the expected word error rate instead of minimizing the frame-level accuracy.

For over 30 years, the speech recognition area has not seen much improvement. The major breakthrough was given by Geoffrey Hinton *et al.* [6], in 2012, where they proposed to replace the GMM by a deep neural network (DNN), improving the state-of-the-art model performance by over 30%.

Following recent trends in deep learning applications for computer vision, where deep models with raw pixel at the input have outperformed traditional machine learning algorithms with hand-engineering feature extractors [1, 38, 40], the end-to-end area for speech recognition has been rising over the years [53, 57, 90, 91]. End-to-end model is a system where as much of the ASR problem pipeline as possible is replaced by a deep neural network architecture. With a single model, the parameters and features learned are solely tuned by the training algorithm to increase the accuracy rate of the system, and detailed in the following section.

3.2 End-to-end speech recognizers

The first successful shot was made by Graves *et al.* in 2006 [11], proposing the connectionist temporal classification (CTC). The main goal of this method is that CTC was specifically designed for temporal classification tasks, *i.e.*, for sequence labeling problems where the alignment between the inputs and the targets is unknown. It also does not require pre-segmented training data, or external post-processing to extract the label sequence from the network outputs. Since then, the CTC method has been extensively used in end-to-end speech recognition systems, and was adopted by Google as the default algorithm in Google Voice Search [92].

More recently, attention-based recurrent networks have been successfully applied to speech recognition [12, 93]. Those RNNs are based on the encoder-decoder architecture (shown in Fig. 2.17e), often used to deal with variable-length input and output sequences, such as in neural machine translation [94], image caption generation [95], and handwriting synthesis [96]. The encoder is typically an RNN that transforms the input into an intermediate representation. The decoder, also typically an RNN, uses this representation to output the desired sequences. The attention mechanism acts by selecting relevant content from the input at every time step, thus helping the decoder.

Although showing promising results, the encoder-decoder method has not outperformed the CTC method yet. We focus the rest of the chapter on the CTC method, since it is the basis of our end-to-end ASR system.

3.3 Connectionist Temporal Classification

The CTC consists of a softmax output layer [20], with one more units than there are labels. The activations of softmax indicate the probabilities of outputting the corresponding labels at a particular time, given the input sequence and the network weights. Considering K unique labels, the neural network output will emit probabilities of $K + 1$, where the extra label will give a probability of outputting a blank, or no label. The softmax can be computed as:

$$\sigma(\boldsymbol{z}) = \frac{e^{\boldsymbol{z}}}{\sum_k e^{z_k}}, \tag{3.3}$$

where the exponentiation is taken element-wise. The softmax function is often used as the final layer of neural networks trained for classification, alongside with the cross-entropy loss [3]

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_i y_i \log \hat{y}_i, \tag{3.4}$$

where $\boldsymbol{y}$ is the desired distribution (target label), and $\hat{\boldsymbol{y}} = \sigma(\boldsymbol{z})$ is the output of our model. The derivative of the loss with respect to the input of softmax layer $\boldsymbol{z}$ is given by:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial z_j} &= -\sum_i \frac{y_i}{\hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_j}\\ &= \underbrace{-y_i(1-\hat{y}_i)}_{i=j} - \sum_{i\neq j}\frac{y_i}{\hat{y}_i}(-\hat{y}_j\hat{y}_i)\\ &= \hat{y}_i\sum_j y_j - y_i = \hat{y}_i - y_i, \qquad (3.5)\end{aligned}$$

which can be represented in vector form as

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{z}} = \hat{\boldsymbol{y}} - \boldsymbol{y}. \qquad (3.6)$$

Unfortunately, the cross-entropy loss is not suitable to problems where we have a sequence as input (raw audio) and another sequence with different size as output (text transcription). This is where the CTC comes in.

Given an utterance $\boldsymbol{X} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(T)})$, its label sequence (*e.g.* characters, words, phonemes) is denoted as $\boldsymbol{l} = (l_1, \ldots, l_u, \ldots, l_U)$, where the blank label $\varnothing$ will be indexed as 0. Therefore, l_u is an integer ranging from 1 to K. The length of $\boldsymbol{l}$ is constrained to be no greater than the length of the utterance, *i.e.*, $U \leq T$. CTC tries to maximize $\log \mathrm{P}(\boldsymbol{l}|\boldsymbol{X})$[1], the log-likelihood of the label sequence given the inputs, by optimizing the RNN model parameters.

The last layer of the RNN is a softmax layer which has $(K+1)$ elements that correspond to the $(K+1)$ labels (including $\varnothing$). At each frame t, we get an output vector $\hat{\boldsymbol{y}}^{(t)}$ whose k-th element $y_k^{(t)}$ is the posterior probability of label k.

Then, if we assume the output probabilities at each time step to be independent given $\boldsymbol{X}$[2], we get the following conditional distribution:

$$\mathrm{P}(\boldsymbol{p}|\boldsymbol{X}) = \prod_{t=1}^{T} \hat{y}_{p_t}^{(t)}, \qquad (3.7)$$

where $\boldsymbol{p} = (p_1, \ldots, p_T)$ is the CTC path, a sequence of labels at frame level. It differs from $\boldsymbol{l}$ in that the CTC path allows the occurrences of the blank label and repetitions of non-blank labels. The label sequence $\boldsymbol{l}$ can be mapped to its corresponding CTC paths. This is a one-to-multiple mapping because multiple CTC paths can correspond to the same label sequence, e.g., both "A A $\varnothing$ B C $\varnothing$" and "$\varnothing$ A A B $\varnothing$ C C" are mapped to the label sequence "A B C". Considering the set of CTC paths

[1]We will denote $\mathrm{P}(\mathbf{a} = \boldsymbol{a}|\mathbf{b} = \boldsymbol{b})$ as $\mathrm{P}(\boldsymbol{a}|\boldsymbol{b})$ for clarity.

[2]*i.e.*, a Markovian assumption

for $\boldsymbol{l}$ as $\Phi(\boldsymbol{l})$, the likelihood of $\boldsymbol{l}$ can be evaluated as a sum of the probabilities of its CTC paths:

$$\mathrm{P}(\boldsymbol{l}|\boldsymbol{X}) = \sum_{\boldsymbol{p}\in\Phi(\boldsymbol{l})} \mathrm{P}(\boldsymbol{p}|\boldsymbol{X}). \tag{3.8}$$

This is the core of CTC. Performing this mapping of different paths into the same label sequence is what makes it possible for CTC to use unsegmented data, because it allows the network to predict the labels without knowing where they occur. In practice, CTC tends to output labels close to where they occur in the input sequence.

As we can see, summing over all CTC paths is computationally impractical. A solution is to represent the possible CTC paths compactly as a trellis. To allow blanks in CTC paths, we add "0" (the blank label) at the beginning and the end of $\boldsymbol{l}$, and also insert "0" between each two original labels in $\boldsymbol{l}$. The resulting augmented label sequence $\boldsymbol{l}^+ = (l_1^+, \dots, l_{2U+1}^+)$ is input to a forward-backward algorithm [9] for efficient likelihood evaluation. In the forward pass, $\alpha(t,u)$ represents the total probability of all CTC paths that end with label l_u^+ at frame t. As with the case of HMMs, $\alpha(t,u)$ can be recursively computed from the previous states. Similarly, a backward variable $\beta(t,u)$ carries the total probability of all CTC paths that start with label l_u^+ at time t and reach the final frame T, and can be computed recursively from the next states. So, defining the set $\varphi(t,u) = \{\boldsymbol{p_t} = (p_1, \dots, p_t) : \boldsymbol{p_t} \in \Phi(\boldsymbol{l}_{u/2}), p_t = l_u^+\}$, one has

$$\alpha(t,u) = \sum_{\boldsymbol{p}_t\in\varphi(t,u)} \prod_{i=1}^{t} \hat{y}_{p_i}^{(t)}. \tag{3.9}$$

Given the above formulation it is easy to see that the probability of $\boldsymbol{l}$ can be expressed as the sum of the forward variables with and without the final blank at time T:

$$\mathrm{P}(\boldsymbol{l}|\boldsymbol{X}) = \alpha(T, 2U+1) + \alpha(T, 2U). \tag{3.10}$$

Since all paths must start with either a $\varnothing$ or the first symbol l_1 in $\boldsymbol{l}$, we have the following initial conditions:

$$\alpha(1,1) = \hat{y}_{\varnothing}^{(1)} \tag{3.11}$$

$$\alpha(1,2) = \hat{y}_{l_2^+}^{(1)} \tag{3.12}$$

$$\alpha(1,u) = 0,\ \forall u > 2. \tag{3.13}$$

For $t = 2$ we have:

$$\alpha(2,1) = \alpha(1,1)\hat{y}^{(2)}_{l^+_1} \tag{3.14}$$
$$\alpha(2,2) = (\alpha(1,1) + \alpha(1,2))\hat{y}^{(2)}_{l^+_2} \tag{3.15}$$
$$\alpha(2,3) = \alpha(1,2)\hat{y}^{(2)}_{l^+_3} \tag{3.16}$$
$$\alpha(2,4) = \alpha(1,2)\hat{y}^{(2)}_{l^+_4} \tag{3.17}$$
$$\alpha(2,u) = 0,\ \forall u > 4, \tag{3.18}$$

For $t = 3$:

$$\alpha(3,1) = (\alpha(1,1) + \alpha(2,1))\hat{y}^{(3)}_{l^+_1} \tag{3.19}$$
$$\alpha(3,2) = (\alpha(2,1) + \alpha(2,2))\hat{y}^{(3)}_{l^+_2} \tag{3.20}$$
$$\alpha(3,3) = (\alpha(2,2) + \alpha(2,3))\hat{y}^{(3)}_{l^+_3} \tag{3.21}$$
$$\alpha(3,4) = (\alpha(2,2) + \alpha(2,3) + \alpha(2,4))y^{(3)}_{l^+_4} \tag{3.22}$$
$$\alpha(3,5) = \alpha(2,4)\hat{y}^{(3)}_{l^+_5} \tag{3.23}$$
$$\alpha(3,6) = \alpha(2,4)\hat{y}^{(3)}_{l^+_6} \tag{3.24}$$
$$\alpha(3,u) = 0\ \forall u > 6. \tag{3.25}$$

If we go on with the trellis (shown in Fig. 3.2), we can find the following recursion:

$$\alpha(t,u) = \hat{y}^{(t)}_{l^+_u} \sum_{i=f(u)}^{u} \alpha(t-1, i), \tag{3.26}$$

where

$$f(u) = \begin{cases} u-1 & \text{if } l^+_u = \varnothing \text{ or } l^+_{u-2} = l^+_u \\ u-2 & \text{otherwise,} \end{cases} \tag{3.27}$$

with the boundary condition

$$\alpha(t,0) = 0\ \forall t. \tag{3.28}$$

We can see that

$$\alpha(t,u) = 0\ \forall u < 2U + 1 - 2(T-t) - 1, \tag{3.29}$$

because these variables correspond to states for which there is not enough time steps left to complete the sequence.

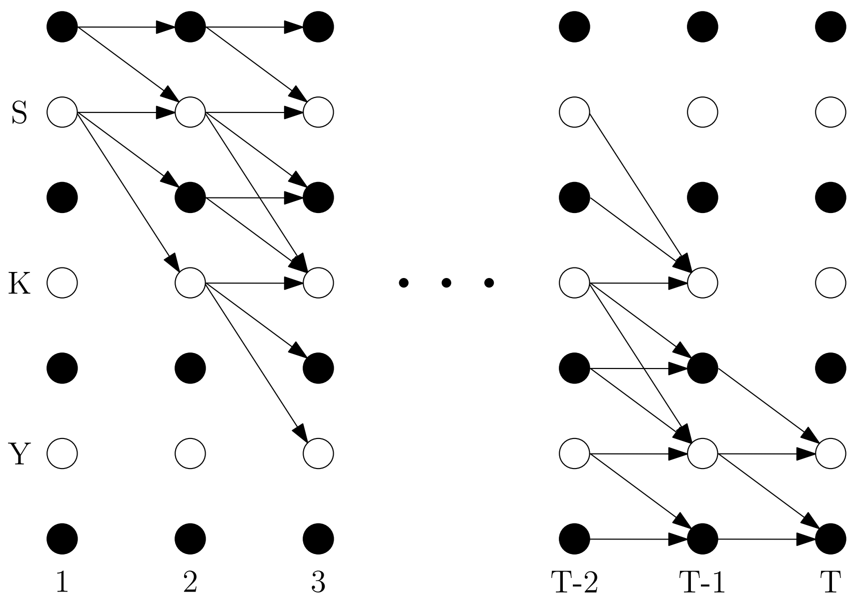


Figure 3.2: CTC trellis of word SKY. Black circles represent blank labels. The arrows represent the paths that will possibly output the correct sentence.

Doing the same thing to $\beta(t,u)$ and defining the set $\varrho(t,u) = \{\boldsymbol{p}_{T-t} : (\boldsymbol{p}_{T-t} \cup \boldsymbol{p}_t) \in \Phi(\boldsymbol{l}), \forall \boldsymbol{p}_t \in \varphi(t,u)\}$, we have the following initial conditions:

$$\beta(T, 2U+1) = \beta(T, 2U+1-1) = 1 \tag{3.30}$$

$$\beta(T, u) = 0, \, \forall u < 2U+1-1. \tag{3.31}$$

Calculating $\beta(t,u)$, we have

$$\beta(t,u) = \sum_{i=u}^{g(u)} \beta(t+1, i) \hat{y}_{l_i^+}^{(t+1)}, \tag{3.32}$$

where

$$g(u) = \begin{cases} u+1 & \text{if } l_u^+ = \varnothing \text{ or } l_{u+2}^+ = l_u^+ \\ u+2 & \text{otherwise.} \end{cases} \tag{3.33}$$

We can see that

$$\beta(t,u) = 0, \, \forall u > 2t, \tag{3.34}$$

with the boundary conditions

$$\beta(t,2U+1+1) = 0\,\forall t. \tag{3.35}$$

3.3.1 Loss function

The CTC loss function $\mathcal{L}(\mathbb{S})$ is defined as the colog probability of correctly labelling all the training examples in some training set $\mathbb{S}$:

$$\mathcal{L}(\mathbb{S}) = -\log \prod_{(\boldsymbol{X},\boldsymbol{l})\in\mathbb{S}} \mathrm{P}(\boldsymbol{l}|\boldsymbol{X}) = -\sum_{(\boldsymbol{X},\boldsymbol{l})\in\mathbb{S}} \log \mathrm{P}(\boldsymbol{l}|\boldsymbol{X}). \tag{3.36}$$

The likelihood of sequence $\boldsymbol{l}$ can then be computed as

$$\mathrm{P}(\boldsymbol{l}|\boldsymbol{X}) = \sum_{u=1}^{2U+1} \alpha(t,u)\beta(t,u), \tag{3.37}$$

where t can be any frame $1 \le t \le T$. Finally, for any time t, we can compute the loss

$$\mathcal{L}(\mathbb{S}) = -\sum_{(\boldsymbol{X},\boldsymbol{l})\in\mathbb{S}} \log \sum_{u=1}^{2U+1} \alpha(t,u)\beta(t,u). \tag{3.38}$$

3.3.2 Loss gradient

The objective function $\log \Pr(\boldsymbol{l}|\boldsymbol{X})$ now is differentiated w.r.t. the RNN outputs $\hat{\boldsymbol{y}}^{(t)}$. Defining an operation on the augmented label sequence $\Gamma(\boldsymbol{l}^+, k) = \{u : l_u^+ = k\}$ that returns the elements of $\boldsymbol{l}^+$ which have the value k, the derivative of the objective function $\log \Pr(\boldsymbol{l}|\boldsymbol{X})$ with respect to $\hat{y}_k^{(t)}$ can be derived as:

$$\frac{\partial \log \mathrm{P}(\boldsymbol{l}|\boldsymbol{X})}{\partial \hat{y}_k^{(t)}} = \frac{1}{\mathrm{P}(\boldsymbol{l}|\boldsymbol{X})} \frac{1}{\hat{y}_k^{(t)}} \sum_{u\in\Gamma(\boldsymbol{l}^+,k)} \alpha(t,u)\beta(t,u). \tag{3.39}$$

Finally, we can backpropagate the gradient through the input of softmax layer $\boldsymbol{z}^{(t)}$ such that

$$\frac{\partial \mathcal{L}(\boldsymbol{X},\boldsymbol{l})}{\partial z_k^{(t)}} = \hat{y}_k^{(t)} - \frac{1}{\mathrm{P}(\boldsymbol{l}|\boldsymbol{X})} \sum_{u\in\Gamma(\boldsymbol{l}^+,k)} \alpha(t,u)\beta(t,u), \tag{3.40}$$

which is similar to Eq. (3.6). Indeed, we can interpret the CTC loss as a soft version of cross-entropy loss [20] where all the paths that form sequence label $\boldsymbol{l}$ are considered.

3.4 Feature extraction

Feeding the recurrent neural network with raw audio does not seem to work well. Raw audio data is noisy, each sample by itself is meaningless, and a recurrent neural network does not have the capacity of correlating them and extracting useful information to predict the desired sequences. Indeed, some recent work has already attempted this, with promising results reported [97, 98]. However, learning from raw data may arguably increase the amount of data and model complexity needed to achieve almost the same performance as learning from hand-designed features.

Usually, a preprocessing stage called feature extraction is performed. Despite not being a neural-network block, this pre-processing stage is usually performed by a single algorithm far less complex than those employed with HMM systems. For this reason, this kind of feature-driven SR system is often called (strictly speaking, misnamed) "all-neural".

The main goal of feature extraction is to identify those components of the signal that are suitable for our objective (recognizing speech) and discard all other information that can be harmful (*e.g.* background noise).

3.4.1 MFCC

Mel-frequency cepstral coefficients (MFCCs) are a feature widely used in speech and speaker recognition systems. Introduced by Davis and Mermelstein in the 1980's [99], the MFCCs are required in almost every real application in ASR.

The first stage in MFCC is to boost the amount of energy at high frequencies. It is well known that high frequencies contain relevant information that could improve the accuracy of speech recognition systems. One example is the spectrum of vowels, where there is more energy at lower frequencies than at higher frequencies, but high-frequency information is needed to differentiate similar vowels. This pre-emphasis is applied by a first-order filter

$$\tilde{s}_n = s_n - \kappa s_{n-1}, \tag{3.41}$$

where $\boldsymbol{s} \in \mathbb{R}^N$ is the digitized audio signal, and $0.9 \leq \kappa < 1.0$ is the pre-emphasis factor.

Speech is a non-stationary signal, meaning that its statistical properties are not constant over time. Instead of analyzing the entire signal, we want to extract features from a small window of speech that could characterize a particular phone or character and for which we can make the assumption of stationarity. We extract this stationary portion of the signal by running an (overlapping) window $\boldsymbol{w}^{\text{in}} \in \mathbb{R}^L$

across the speech signal

$$\tilde{S}_{m,l} = s_l w^{\text{in}}_{l-mS}, \tag{3.42}$$

where $l = \{1, \ldots, L\}$, S is the hop between successive windows, and $m = \left\lceil \frac{N-(l-S)}{S} \right\rceil$ is the index of the frame. The more common window used in MFCC extraction is the Hamming window

$$w^{\text{in}}_n = 0.54 - 0.46 \cos\left(\frac{2\pi n}{N-1}\right). \tag{3.43}$$

The following step is to calculate the power spectrum of each windowed signal

$$S_{m,k} = \left| \sum_l \tilde{S}_{m,l} e^{\frac{-j2\pi kl}{L}} \right|^2, \tag{3.44}$$

where k is the index of the discrete frequency. This is motivated by the human cochlea, which vibrates at different spots depending on the frequency of the incoming sounds. Depending on the spatial location along the cochlea that vibrates, different nerves are activated to inform the brain that certain frequencies are present. The power spectrum performs a similar job, identifying which frequencies are present in the frame.

The cochlea does not have a fine resolution to distinguish two closely spaced frequencies, and this effect is more pronounced as frequency increases. For this reason, we use the mel-scale filter bank, which divides the spectrum in non-uniform bins (to emulate the variable auditory resolution) and sum them up, getting the idea of how much energy exists in different frequency regions. As shown in Fig. 3.3, the first mel-scale filters are very narrow and give an indication of how much energy exists at low frequencies. As frequencies get higher, filters get wider. The mel scale can be computed as

$$\text{mel} = 2595 \log_{10}\left(1 + \frac{f}{700}\right), \tag{3.45}$$

where f is the frequency in Hz. Once we have the filtered energies, we take their logarithm. This is also motivated by human hearing, whose response to sound intensity is also logarithmic (exhibiting higher resolution at lower intensities). Moreover, using a log function makes the feature estimates less sensitive to variations in input, such as power variations due to the speaker's movements.

The next stage is to compute the discrete cosine transform (DCT) of those log filter-bank energies. As the filter banks are all overlapping, the filter-bank energies are quite correlated with each other. The DCT decorrelates those energies, improving the performance of the model to predict the sequences. For the relevant

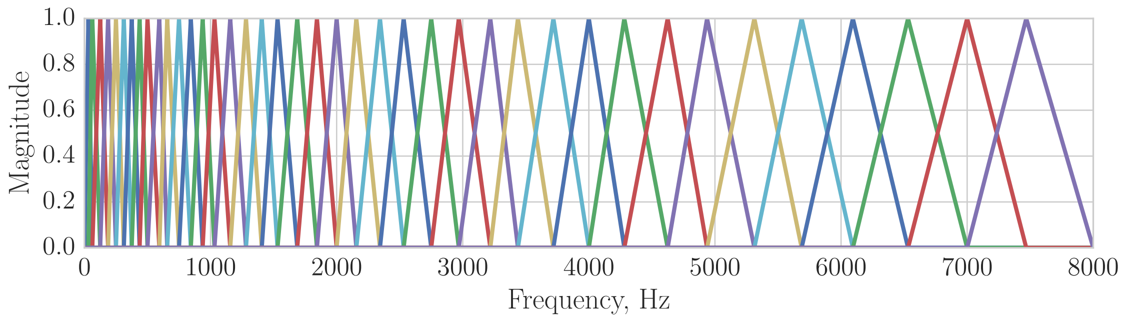


Figure 3.3: 40 filter banks on a mel-scale.

MFCC extraction, we just take the first 12 cepstral[3] values (excluding the 0th), because higher information represent quick changes in the filter-bank energies, which tend to degrade the ASR performance. One may apply sinusoidal liftering[4] to de-emphasize higher MFCCs, which has been claimed to improve the ASR systems' performance in noisy signals [100].

Finally, to balance the coefficients and improve the signal-to-noise ratio, we can subtract the mean of each coefficient calculated over all frames, often termed as cepstral mean normalization (CMN). Also, one might include the first and second derivatives of the coefficients (called delta and double delta features), which has been claimed to help the recognizer. All stages of the MFCCs calculation are shown in Fig. 3.4.

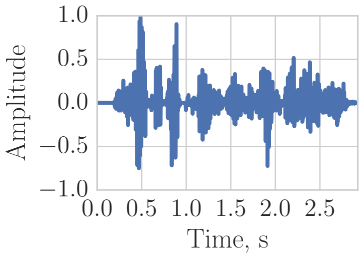
All stages needed to compute filter banks were motivated by the internal correlation of speech and human perception of such signals. The extra steps necessary to calculate the MFCCs, however, are justifiable in part by the limitation of classical ASR systems. For example, the main DCT role is to decorrelate the filter-bank energies. MFCCs became very popular when GMM HMM-based models were very popular, and they coevolved to be the standard way of doing ASR. However, researchers involved in speech recognition powered by deep learning have argued [6] whether MFCCs are still the correct choice, given that deep models are more robust to highly correlated inputs.

3.4.2 Convolutional networks as features extractors

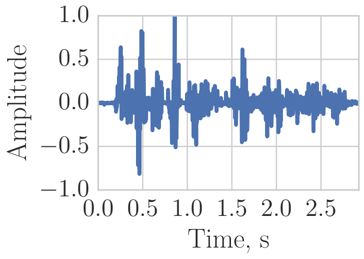
As we described in the last chapter, convolutional neural networks are good as feature extractors. Recently, researchers have achieved promising results [97, 98, 101, 102] when applying CNNs to the raw audio signal. The convolutional neural network applied to raw audio differ from that shown in Sec. 2.10 because it needs to be applied to one-dimensional data. In a simple way, we can just use the 2D

[3]Cepstral relates to spectral in the MFCC domain.

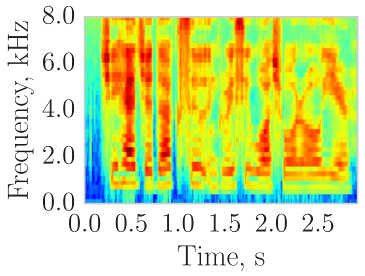
[4]Liftering relates to filtering in the cepstral domain.



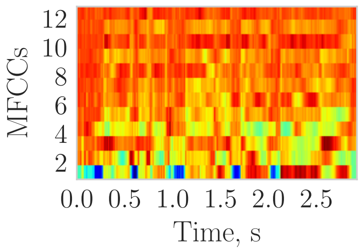
(a) Raw signal



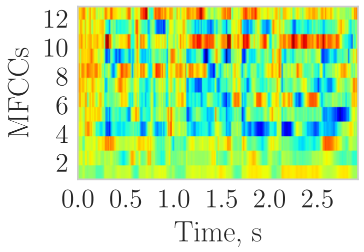
(b) Pre-emphasis signal



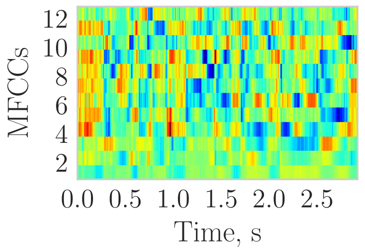
(c) filter bank calculated over mel-scale



(d) DCT



(e) Sinusoidal lifter



(f) CMN

Figure 3.4: MFCCs calculation steps.

convolutional layer by setting the height of inputs and filters to 1, the width as the number of time steps, and the channel size as the number of features. Doing these modifications, all CNN framework described to spatial data can also be applied to the raw audio.

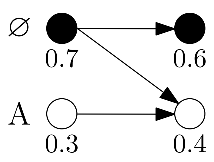
Furthermore, CNNs have also been applied over the preprocessed signal [91, 103, 104]. Their unique architecture characterized by local connectivity, weight sharing, and pooling has exhibited less sensitivity to small shifts in speech features along the frequency axis, which is important to deal with different speakers and environment variations.

3.5 Decoding the sequence

During the inference, our model with CTC method will return at each time step the probability of each label, including the blank label. One simple way to decode the CTC's output into a legible sequence is based on the premise that the most probable path corresponds to the most probable labeling

$$\hat{\boldsymbol{l}} = \Phi^{-1}(\operatorname*{argmax}_{\boldsymbol{p}} \mathrm{P}(\boldsymbol{p}|\boldsymbol{X})), \tag{3.46}$$

often known as greedy decoding. However this strategy can lead to errors, particularly if a label is weakly predicted for several consecutive time steps, as depicted in Fig. 3.5. More accurate decoding can be achieved with a beam search algorithm.



$$P(\hat{\boldsymbol{l}} = \varnothing) = P(\varnothing\varnothing) = 0.7 \times 0.6 = 0.42$$

$$\begin{aligned} P(\hat{\boldsymbol{l}} = \mathrm{A}) &= P(\mathrm{AA}) + P(\mathrm{A}\varnothing) + P(\varnothing\mathrm{A}) \\ &= 0.3 \times 0.4 + 0.3 \times 0.6 + 0.7 \times 0.4 \\ &= 0.58 \end{aligned}$$

Figure 3.5: Example where the greedy decoding fails to find the most probable labelling.

The beam search decoder [90] maintains a small number B of partial hypotheses, where a partial hypothesis is some partial labelling. At each time step we extend each partial hypothesis in the beam with every possible new label. This greatly increases the number of hypotheses; so, we discard all but the B most likely hypotheses according to their probabilities. The decoder ends by returning the most probable labelling at time step T from the B most likely hypotheses. The pseudocode in Alg. 1 describes a simple beam search procedure for a CTC network. For each time step t and for each labelling sequence $\tilde{\boldsymbol{l}}$ in our hypothesis set $\mathbb{H}$, we consider concatenating a new label l to $\tilde{\boldsymbol{l}}$, denoted as $\tilde{\boldsymbol{l}}^{+} \leftarrow \tilde{\boldsymbol{l}} + l$. Blanks and repeated labels with no separating blank are handled separately. We initialize our hypothesis set with one sequence $\tilde{\boldsymbol{l}} = \varnothing$. If $P_{\mathrm{b}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)})$ and $P_{\mathrm{nb}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)})$ mean, respectively, the probabilities of sequence $\tilde{\boldsymbol{l}}$ ending and not ending with blank label given the input $\boldsymbol{X}^{:(t)} = (\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)})$ up to time t, then $P(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)}) = P_{\mathrm{b}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)}) + P_{\mathrm{nb}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)})$. The last label of sequence $\tilde{\boldsymbol{l}}$ is denoted as $\tilde{\boldsymbol{l}}_{-1}$.

3.5.1 Improving performance: employing a language model

None of those decoding methods, however, have a prior knowledge of the language idiosyncrasies (*e.g.* how to spell the words). Including a language model into the decoding has shown to boost the accuracy of speech recognition systems, turning it into an essential tool. One simple way to construct a language model is building a dictionary. In [56], the authors have proposed an easy way of changing the beam search decoding to constrain the search to a dictionary of possible words.

Another simple way of creating a language model is through statistics. Our objective is to compute the probability of a label l (*e.g.* phoneme, character, or word) given some history $\boldsymbol{h}_{\mathrm{IS}}$ from a *corpus* (*e.g.* texts from the Web, books' transcription), or $P(l|\boldsymbol{h}_{\mathrm{IS}})$. One way to estimate $P(l|\boldsymbol{h}_{\mathrm{IS}})$ is counting the relative frequency:

$$P(l|\boldsymbol{h}_{\mathrm{IS}}) \approx \frac{C(\boldsymbol{h}_{\mathrm{IS}} + l)}{C(\boldsymbol{h}_{\mathrm{IS}})}, \tag{3.47}$$

where $C(\cdot)$ is the operator for counting and $\boldsymbol{h}_{\mathrm{IS}} + l$ is the concatenation of sequence $\boldsymbol{h}_{\mathrm{IS}}$ with label l. The idea of calculating the probability of a label given the entire

Algorithm 1 Beam Search Decoding

```
Input CTC likelihoods $y_l^{(t)} = \mathrm{P}(l|\boldsymbol{x}^{(t)})$, $\forall t \in \{1, \dots, T\}$, $l \in \mathbb{L}$, where $\mathbb{L}$ is the set
    of valid labels (including the blank label, $|\mathbb{L}| = K + 1$).
Parameters beam width $B$
Initialize $\mathbb{H} \leftarrow \{\varnothing\}$, $\mathrm{P_b}(\varnothing|\boldsymbol{X}^{:(0)}) \leftarrow 1$, $\mathrm{P_{nb}}(\varnothing|\boldsymbol{X}^{:(0)}) \leftarrow 0$
 1: for $t = 1, \dots, T$ do
 2:     $\hat{\mathbb{H}} \leftarrow \{\}$
 3:     for $\tilde{\boldsymbol{l}} \in \mathbb{H}$ do
 4:         $\mathrm{P_b}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)}) \leftarrow \mathrm{P}(\varnothing|\boldsymbol{x}^{(t)})\,\mathrm{P}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t-1)})$        ▷ Handle blanks
 5:         $\mathrm{P_{nb}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)}) \leftarrow \mathrm{P}(\tilde{\boldsymbol{l}}_{-1}|\boldsymbol{x}^{(t)})\,\mathrm{P_{nb}}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t-1)})$        ▷ Handle repeat character collapsing
 6:         Add $\tilde{\boldsymbol{l}}$ to $\hat{\mathbb{H}}$
 7:         for $l \in \mathbb{L} \setminus \{\varnothing\}$ do
 8:             $\tilde{\boldsymbol{l}}^{+} \leftarrow \tilde{\boldsymbol{l}} + l$
 9:             if $l \neq \tilde{\boldsymbol{l}}_{-1}$ then
10:                 $\mathrm{P_{nb}}(\tilde{\boldsymbol{l}}^{+}|\boldsymbol{X}^{:(t)}) \leftarrow \mathrm{P}(l|\boldsymbol{x}^{(t)})\,\mathrm{P}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t-1)})$
11:             else
12:                 $\mathrm{P_{nb}}(\tilde{\boldsymbol{l}}^{+}|\boldsymbol{X}^{:(t)}) \leftarrow \mathrm{P}(l|\boldsymbol{x}^{(t)})\,\mathrm{P_b}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t-1)})$        ▷ Repeat label have $\varnothing$ between
13:             end if
14:             Add $\tilde{\boldsymbol{l}}^{+}$ to $\hat{\mathbb{H}}$
15:         end for
16:     end for
17:     $\mathbb{H} \leftarrow B$ most probable $\tilde{\boldsymbol{l}}$ by $\mathrm{P}(\tilde{\boldsymbol{l}}|\boldsymbol{X}^{:(t)})$ in $\hat{\mathbb{H}}$
18: end for
Return $\mathrm{argmax}_{\tilde{\boldsymbol{l}} \in \mathbb{H}}\,\mathrm{P}(\tilde{\boldsymbol{l}}|\boldsymbol{X})$
```

history is the base of n-gram models [84]. Instead of computing this probability given the entire history, which would be impractical, we can approximate the history by just the last few labels. For example, a 2-gram model gives the probability that a label occurs given the last label, while a 3-gram model gives the probability that a label occurs given the last couple of labels. Another way of performing the same estimate is through an RNN model [51, 105, 106], where it tries to predict the label given the previous ones.

Applying those language models to the decoder does not seem straightforward. One solution was given by Graves *et al.* [55], where they rescore the softmax output by the probability of the character occurring given the past context. Another way of introducing a character language model into the decoding is rescoring the beam search, as demonstrated in [90].

3.6 Related work

We have already described how neural networks work in Chapter 2, how we can adapt them to handle sequences with recurrent networks in Sec. 2.11, and how to initialize the network parameters and perform the training. Note that our goal is to construct an all-neural speech recognition system, often termed end-to-end speech recognizer. For this reason, we have handled variable length sequences at the input (*e.g.* MFCCs) and variable length sequences at the output (*e.g.* text transcriptions, phonemes) by introducing the CTC method. Also, we have shown how to parse the output of the softmax layer in a legible sequence using decoders. Finally, we have pointed how we can improve the decoder by employing a language model. Thus, at this point, we already have the tools to construct an all-neuron speech recognizer, but we can profit from other researcher's previous experience. Therefore, we will go through a short discussion of end-to-end ASR CTC-based models published in the past years.

3.6.1 Graves' models

Graves *et al.* [11] have proposed the first successful end-to-end solution to the speech recognition task. Their model, depicted in Tab. 3.1, consisted in one bidirectional LSTM (BLSTM) with peephole connections, with 100 hidden units. The input layer was size 26 — 12 MFCCs calculated over a window of 10 ms and hop of 5 ms from 26 filter banks, plus the log energy and delta coefficients. Training was carried out with SGD+Nesterov with a learning rate $\eta = 10^{-4}$ and a momentum $\mu = 0.9$. During the training, Gaussian noise with standard deviation of 0.6 was added to the inputs. They achieved the label error rate (LER)[5] of $30.51 \pm 0.19\%$ over five runs using the TIMIT phoneme recognition dataset [107].

In 2013 [55], Graves *et al.* proposed a novel method combining the CTC-like model with a separate recurrent net that predicts each phoneme given the past ones, training both acoustic and language models together, called RNN transducer [108]. Their best acoustic model, depicted in Tab. 3.2, was different from Tab. 3.1 not just by the jointly training but for stacking more layers. They have achieved a label error rate of 17.7% on the TIMIT, the best score at that time. The regularization was done by applying early stopping and weight noise (the addition of Gaussian noise to the network weights during the training [109]). Weight noise was added once per training sequence, rather than at every time step. The audio data was preprocessed by log filter banks with 40 coefficients (plus energy) distributed on a mel-scale, together with their first and second temporal derivatives. Training was also carried

[5]Label error rate (LER) is the summed edit distance [84] between the output sequences and the target sequences, normalized by the total length of the target sequences.

Table 3.1: Graves 2006 model [11]. The input has size $B \times T \times D$, where B is the batch-size, T is the time step, and D is the number of features. The TimeDistributedDense layer, with N hidden units and without an activation function, applies the same fully connected (dense) layer to each time step. Finally, the CTC loss function includes the softmax calculation.

	Operation	Units	Nonlinearity
Network	Input $T \times 26$		
	Gaussian noise	$\sigma = 0.6$	
	BLSTM	100	*peepholes connections*
	TimeDistributedDense	62	
	Preprocessing	12 MFCCs + log energy + delta coefficients	
	Loss	CTC	
	Decoder	Prefix search decoding	
	Optimizer	SGD+Nesterov, with $\eta = 10^{-4}$ and $\mu = 0.9$	
	Regularization		
	Batch size	1	
	Epochs		
	Learning rate schedule		
	Weight Initialization	Uniform distribution ranging from -0.1 to 0.1	

out with SGD+Nesterov with a learning rate $\eta = 10^{-4}$ and a momentum $\mu = 0.9$. Beam search decoding was used, with a beam width of 100. The advantage of deep networks was obvious with the error rate of CTC dropping from 23.9% to 18.4% as the number of layers increased from one to five, without the transducer [55].

Table 3.2: Graves2013 model. The label 5× indicates that the BLSTM layer is repeated 5 times.

	Operation	Units	Nonlinearity
Network	Input $T \times 123$		
5×	BLSTM	250	
	TimeDistributedDense	62	
	Preprocessing	40 log filter banks + log energy + 1st and 2nd derivatives	
	Loss	CTC + RNN Transducer	
	Decoder	Beam search, with beam width = 100	
	Optimizer	SGD+Nesterov, with $\eta = 10^{-4}$ and $\mu = 0.9$	
	Regularization	Weight noise	
	Batch size	1	
	Epochs		
	Learning rate schedule		
	Weight Initialization	Uniform distribution ranging from -0.1 to 0.1	

3.6.2 Maas' model

Maas *et al.* [90] was one of the first successful papers proposing to convert speech into characters instead of phonemes using CTC. They achieved competitive results on the challenging Switchboard telephone conversation transcription task [110]. They have presented a speech recognition system that uses only a neural network for the acoustic model, a character-level language model (CLM), and the beam search procedure. By operating over characters, they eliminated the need for a lexicon[6] and enabled the transcription of new words, fragments, and disfluencies. Also, they proposed using the clipped ReLU activation

$$\sigma(x) = \min(\max(x{,}0), \text{threshold}), \tag{3.48}$$

where this ReLU function is clipped to a maximum delimited by a threshold to prevent overflow.

Their input consists of MFCCs coefficients with a context window of 10 frames at each side. Their network has five layers and is described in Tab. 3.3. One intriguing characteristic is that the model not only relies on RNN layers. Instead, they have used more fully connected layers than RNNs. Their output label set consists of 33 characters, including the special blank character, "-", and apostrophe, as used in contractions. They have trained their model using SGD + Nesterov with a learning rate $\eta = 10^{-5}$ and momentum $\mu = 0.95$. After each epoch, they divided the learning rate by 1.3 and trained the model for 10 epochs using the Switchboard dataset (over 300 hours of speech), available at LDC under catalog number LDC97S62. The model was evaluated over the HUB5'00 (full) dataset [111]. To learn how to spell words, they used a corpus with over 31 billion words gathered from the web to train an n-gram model. Also, they modified the beam search decoder to accept information from the CLM. Their filtered results are shown in Tab. 3.4. It is worth noticing that without a language model they have achieved 27.7% of label error rate.

3.6.3 EESEN

EESEN [57] is distinct from the others by proposing a new way of decoding the sequences using a weighted finite-state transducer (WFST) [84], which enables the efficient incorporation of lexicons and language models into CTC decoding. They have shown WERs comparable to those of standard hybrid DNN systems (using HMM model). Their topology is summarized in Tab. 3.5. Training was carried out with a batch size of 10 samples, sorted by their sequence lengths. The learning rate η started with value 4×10^{-5} and remained constant until the drop of LER

[6]Lexicon could be a simple dictionary converting sequences of phonemes to words.

Table 3.3: Maas' model. The batch size and the number of features are "?" (unknown), because the author did not specify them properly. "merge by sum" means that the forward and backward states of the BRNN were merged by sum instead of concatenation.

	Operation	Units	Nonlinearity
Network	Input $T\times?$		
$2\times$	TimeDistributedDense	1824	Clipped ReLU, theshold=20
	BRNN	1824	Clipped ReLU, theshold=20 *merge by sum*
$2\times$	TimeDistributedDense	1824	Clipped ReLU, theshold=20
	TimeDistributedDense	33	
	Preprocessing	MFCCs + context window of ±10 frames	
	Loss	CTC	
	Decoder	Beam search, with beam width = 100 and CLM	
	Optimizer	SGD+Nesterov, with $\eta = 10^{-5}$ and $\mu = 0.95$	
	Regularization		
	Batch size		
	Epochs	10	
	Learning rate schedule	Divide by 1.3 after each epoch	
	Weight Initialization		

Table 3.4: Results from Maas *et al.* [90]. Label error rate and word error rate (WER) results for the model without a language model, with 7-gram character language model (CLM), and CLM built with an RNN of 3 layers, respectively.

Model	LER	WER
Maas w/o LM	27.7%	47.1%
Maas + 7-gram	24.7%	35.9%
Maas + RNN-3	24.7%	30.8%

on the validation set within two consecutive epochs fell below 0.5%. Then, the learning rate is reduced by a factor of 0.5 at each of the subsequent epochs. The whole training process terminates when the LER fails to decrease by 0.1% between two consecutive epochs. The experiments were conducted on the Wall Street Journal [112, 113] (WSJ) *corpus*. Inputs of the model were 40-dimensional filter bank features together with their first and second-order derivatives. These features were normalized via mean subtraction and variance normalization by speaker. For the phoneme recognition, a set of 72 labels were used, whereas for the character-based systems, a set of 59 labels were employed, including letters, digits, punctuation marks, and so on. The best results are presented in Tab. 3.6. Notice that without a proper decoder, the WER rises quickly to 26.92%.

Table 3.5: EESEN model. "phn" stands for the phoneme-based system, and "char" stands for the character-based system.

	Operation	Units	Nonlinearity
Network	Input $T \times 120$		
$4\times$	BLSTM	320	
	TimeDistributedDense	phn: 72 char: 59	
	Preprocessing	40 log filter banks + 1st and 2nd derivatives Mean and variance normalization by speaker	
	Loss	CTC	
	Decoder	WFST-based	
	Optimizer	SGD+Nesterov, with $\eta = 4 \times 10^{-5}$	
	Regularization		
	Batch size	10, sorted by the sequence length	
	Epochs	Until LER fails to decrease by 0.1% between 2 epochs	
	Learning rate schedule	Decayed by 0.5 at each epoch after special condition	
	Weight Initialization	Uniform distribution ranging from -0.1 to 0.1	

Table 3.6: EESEN best results.

Model	LM	WER
EESEN-PHN	lexicon	26.92%
EESEN-PHN	trigram	8.5%
EESEN-CHAR	trigram	7.34%

3.6.4 Deep Speech 1 and 2 by Baidu Research

Baidu research has distinguished from the others by employing model and data parallelism [1, 114], combined with its own massive dataset with huge data augmentation by a complex addition of synthetic noise. Their first model, called Deep Speech 1 [115], is described in Tab. 3.7, which is inspired a lot by the Maas' model. Regularization was done by applying dropout only in the non-recurrent connection, with one different mask for each time step, and by applying a special type of jitter on the input. Training was performed using SGD+Nesterov with a momentum μ of 0.99. Inputs were preprocessed by computing the spectrogram of 80 linearly spaced log filter banks and an energy term with a context window of ± 9 frames. The filter banks were computed over windows of 20 ms with a hop of 10 ms.

In [115], the authors have stated that the errors made by an RNN tend to be a phonetically plausible rendering of English words. The model was integrated with an n-gram language model trained on a dataset up to 220 million phrases, supporting a vocabulary of 495,000 words. Aiming to minimize the running time of recurrent layers, they have shortened the recurrent layers by taking strides of size 2 in the original input, halving the unrolled RNN.

Results are shown in Tab. 3.8. Training was spawned over the SwitchBoard dataset (SWB) + Fisher [116] (FSH) and over the combination of its own dataset,

Table 3.7: Deep Speech 1 model. The Deep Speech SWB+FSH model is an ensemble of 4 models with 2304 neurons in each layer.

	Operation	Units	Nonlinearity
Network	Input $T/2 \times 1539$		
3×	Dense block		
	BRNN	2048	Clipped ReLU, threshold= 20
			merge by sum
	Dropout	*dropout probability between* 5% *and* 10%	
	Dense block		
	TimeDistributedDense	29	
Dense block	TimeDistributedDense	2048	Clipped ReLU, threshold= 20
	Dropout	*dropout probability between* 5% *and* 10%	
	Preprocessing	80 linearly spaced log filter banks	
		+ log energy + context window ±9	
	Loss	CTC	
	Decoder	Customized with language model	
	Optimizer	SGD+Nesterov, with $\mu = 0.99$	
	Regularization		
	Batch size		
	Epochs		
	Learning rate schedule		
	Weight Initialization		

Fisher, SwitchBoard, and WSJ, totalizing over 7000 hours of speech. The former was evaluated over the HUB5'00 and the latter over a constructed noisy speech dataset. The noisy experiment has shown that their system outperformed several commercial speech systems.

Table 3.8: Deep Speech 1 model. WER over different datasets.

Train set	Test set	WER
SWB	HUB5'00	25.9%
SWB + FSH	HUB5'00	16.0%
All	Noisy	11.85%

Deep speech 2 [91] proposed a unique model that could be used to recognize either English or Mandarin Chinese speech with minor modifications. One of the key approaches of Deep Speech 2 was the efficient use of hyper power computer techniques, including an extremely efficient GPU implementation of the CTC loss function, resulting in a 7× speedup over their previous model.

The model, detailed in Tab. 3.9, was quite different from the previous system. Instead of dense layers, convolutional layers were adopted. Batch norm was employed to accelerate training. For the recurrent connection, batch norm was applied only to the non-recurrent weights, as denoted in [27]. Also, a novel algorithm called SortaGrad [91] was proposed. In the first training epoch, the iteration through the training set is done in increase order of the length of the longest utterance in the mini-batch. After the first epoch, training is normally performed. They argued that long sequences are more likely to cause the internal state of the RNNs to explode at an early stage in training, and the SortaGrad helps to avoid this phenomenon. Inputs were preprocessed by computing its spectrograms, and the output for the English-based system is a bi-grapheme or bigram labels. For the Chinese-based model, they have adapted the output to handle more than 6000 characters, which include the Roman alphabet. Their reported results are shown in Tab. 3.10. All models were trained for 20 epochs on either the full English Dataset (11940 hours of speech) or the full Mandarin dataset (9400 hours of speech).

Training was carried out with SGD+Nesterov with a momentum μ of 0.99, along with mini batches of 512 utterances. Gradient clipping was set to 400. The learning rate η was chosen from $[10^{-4}, 6\times10^{-4}]$ to yield the fastest convergence and annealed by a factor of 1.2 at the end of each epoch.

Table 3.9: Deep Speech 2 model. The best result were given by a 11-layer architecture.

	Operation	Units	Nonlinearity
Network	Input $T/2 \times 1539$		
3×	Conv block		
7×	BRNN	2048	Clipped ReLU, threshold= 20
			merge by sum
			Recurrent batch norm
	TimeDistributedDense	?	
Conv block	Conv2D		Clipped ReLU, threshold= 20
	Batch normalization		
	Preprocessing	80 linearly spaced log filter banks	
		+ log energy + context window ±9	
	Loss	CTC	
	Decoder	Customized with language model	
	Optimizer	SGD+Nesterov	
		$\eta = [10^{-4}, 6 \times 10^{-4}]$, and $\mu = 0.99$	
	Regularization		
	Batch size	512	
	Epochs	20 (SortaGrad)	
	Learning rate schedule	Annealed by a factor of 1.2 after each epoch	
	Weight Initialization		

3.7 Proposed model

All architectures listed above follow the same recipe: starting with fully connected layers (or convolutional layers), followed by recurrent layers, and then by further fully connected layers.

Table 3.10: Deep Speech 2 WER over different datasets. Baidu Test is their internal English test set with 3,300 examples. The test set contains a wide variety of speech including accents, spontaneous and conversational speech. WSJ is the Wall Street Journal corpus test set available in the LDC catalog. CHiME eval real dataset has 1320 utterances from the WSJ test set read in various noisy environments. The internal test set for Chinese Mandarin speech is its internal *corpus* consisting of 1882 examples.

Language	Test set	DS1	DS2	Human
English	Baidu Test	24.01%	13.59%	-
	WSJ eval'93	6.94%	4.98%	8.08%
	CHiME eval real	67.94%	21.79%	11.84%
Chinese Mandarin	Internal	-	7.93%	-

Both Baidu's models need to parallelize the model and the data, being necessary more than one GPU available, and a lot of expertise and optimized implementations to get everything working. Moreover, their models require a huge dataset, and it is unfeasible for us.

Unfortunately, there is a lack of detailed explanation (or any information at all) about the regularization methods to achieve better generalization in some models. We try to investigate the different methods to regularize the network presented in Sec. 2.12.

Overall, our topology is quite simple, as given in Tab. 3.11. We avoid using dense layers, because they add many parameters, potentializing the overfitting problem. Using a lot of dense layers only scales well for an enormous amount of data. The input will be preprocessed by extracting the MFCCs (as described in Sec. 3.4.1) with their first and second derivatives and applying cepstral mean and variance normalization to each utterance. We also investigate the use of context window.

Then, the inputs are fed to a BLSTM layer, with forget gate initialized with one. As explained in Sec. 2.11.2, bidirectional recurrent networks are better in modeling the co-articulation effect on speech and show some improvements in accuracy rate on ASR systems [56] over the unidirectional ones. Variational dropout will be utilized as a regularizer, and we investigate the use of layer normalization, and multiplicative integration, to improve generalization and speed up the training. Also, we investigate using zoneout instead of variational dropout. Weight decay is applied to all weights of the network. One or more layers of BLSTM is employed. In the end, a dense layer is appended to the output of the last BLSTM at each time step. Our model, at each time step, emits the probability of a character given the input. Finally, the CTC (with a softmax) computes the loss of our network. At training time, the greedy decoder is used to calculate the label error rate (LER) at every epoch. At test time, beam search decoder with a beam width of 100 is used, without any language model. Unfortunately, we do not test the impact of a language model in this dissertation.

Training will be carried out with Adam, maintaining their default parameters $\epsilon = 10^{-8}$, $\alpha_v = 0.9$ and $\alpha_m = 0.99$, and a search over the hyperparameters, such as the learning rate η, number of layers, and the number of hidden units, will be performed.

Table 3.11: The proposed model. The variables batch size B, size of context window C, number of layers N, number of hidden units H, dropout probability P, and number of labels L will be investigated in the next chapter.

	Operation	Units	Nonlinearity
Network	Input $T \times (1 + 2 * C)$		
$N\times$	BLSTM	H	
	TimeDistributedDense	L	
	Preprocessing	12 MFCCs	
		+ log energy	
		+ delta and double delta	
	Loss	CTC	
	Decoder	Beam search, with beam width = 100	
	Optimizer	Adam	
	Regularization	Weight decay	
		Variational dropout	
	Batch size	B	
	Epochs	30-100	
	Learning rate schedule	None	
	Weight Initialization	Recurrent: orthogonal [117]	
		Nonrecurrent: Xavier	
		Forget bias = 1	

Chapter 4

Experiments on all-neural speech recognition

In this Chapter, we go through a discussion of several topologies, different regularization techniques, and hyperparameters choices. Firstly, we validate our implementation by replicating the first Graves' model [11]. After that, we apply our knowledge to build an end-to-end Brazilian Portuguese speech recognition system.

4.1 English and Portuguese datasets

Two datasets were used to perform the evaluations. The TIMIT dataset was employed to validate our implementation details, duplicating the Graves' model and results. The second dataset, the main target of this dissertation, is an ensemble of four different datasets.

4.1.1 TIMIT

TIMIT [107] is a speech dataset that was developed by Texas Instruments and MIT with DARPA's (Defense Advanced Research Projects Agency) financial support by the end of 1980, and now is maintained by the linguistic data consortium (LDC) under catalog number LDC93S1. This dataset has many applications, such as the study of acoustic and phonetic properties and the evaluation/training of automatic speech recognition systems (ASR).

There are broadband recordings of 630 speakers of 8 major dialects of American English, each reading ten phonetically rich sentences. Each utterance is separated into 3 broad categories: SA (dialect sentence), SX (compact sentence), and SI (diverse sentence). The distribution of the dataset is detailed in Tab. 4.1

The SA sentences were meant to show the dialectal variants among the speakers and were read by all 630 speakers. Therefore, for an automatic speaker-independent

Table 4.1: Distribution over region and genre of the TIMIT dataset. Dialect regions: (1) New England, (2) Northern, (3) North Midland, (4) South Midland, (5) Southern, (6) New York City, (7) Western, (8) Army Brat (moved around).

Region	Men	Women	Total
1	31 (63%)	18 (27%)	49 (8%)
2	71 (70%)	31 (30%)	102 (16%)
3	79 (67%)	23 (23%)	102 (16%)
4	69 (69%)	31 (31%)	100 (16%)
5	62 (63%)	36 (37%)	98 (16%)
6	30 (65%)	16 (35%)	46 (7%)
7	74 (74%)	26 (26%)	100 (16%)
8	22 (67%)	11 (33%)	33 (5%)
Total	438 (70%)	192 (30%)	630 (100%)

recognition system, these sentences must be ignored.

The phonetically-compact (SX) sentences were designed to provide a good coverage of pairs of phones, with extra occurrences of phonetic contexts thought to be either difficult or of particular interest. Each speaker read 5 of these sentences and each text was spoken by 7 different speakers.

Finally, the phonetically-diverse (SI) sentences were selected from existing text sourcesto add diversity in sentences types. Each speaker reads 3 of these sentences, with each sentence being read only by 1 speaker. All audio files were recorded in a controlled environment.

Each utterance in the TIMIT dataset has its own time-aligned orthographic, phonetic, and word transcriptions as well as 16-bit, 16 kHz speech waveform in the National Institute of Standards and Technology (NIST) format. Also, the dataset is separated by two major sets: test and train. The test set has 168 speakers and 1344 utterances available (recalling that SA sentences were not meant to be used in ASR systems). This test set is also called the complete test set. Using the complete test set has a drawback: the intersection of SX sentences by different speakers. Facing that, the researchers would rather evaluate the ASR system in the core test set.

The core test set has 24 speakers, 2 men and 1 woman of each dialect region, where each one reads 5 unique SX sentences plus its 3 SI sentences, given 192 utterances, as shown in Tab. 4.2. Usually, the rest of the sentences presented in the complete test set are used as the validation set.

It is worth mentioning that the TIMIT dataset presents high-quality records, with high signal-to-noise ratio, and controlled environment. Also, the TIMIT corpus transcriptions have been hand verified, containing balanced phonetic sentences and great dialect coverage. In total, there are 61 different annotated phonemes; however, it is usually mapped to a 39-phone subset as proposed in [118].

Table 4.2: The core test set distribution. The column "Man" and "Woman" shows the unique speaker identification.

Region	Man	Woman
1	DAB0, WBT0	ELC0
2	TAS1, WEW0	PAS0
3	JMP0, LNT0	PKT0
4	LLL0, TLS0	JLM0
5	BPM0, KLT0	NLP0
6	CMJ0, JDH0	MGD0
7	GRT0, NJM0	DHC0
8	JLN0, PAM0	MLD0

4.1.2 Brazilian Portuguese dataset

We aim to build an end-to-end Portuguese speech recognition system using state-of-the-art algorithms powered by deep learning. Therefore, we need a large dataset, and it is difficult to find one because they are either not freely accessible or expensive to acquire.

The Brazilian Portuguese speech dataset (BRSD) for long vocabulary continuous speech recognition has been built from four different datasets that we have available. Three of them are freely distributed. The last one is distributed by the Linguistic Data Consortium under catalog number LDC2006S16. These datasets are summarized in Tab. 4.3.

Table 4.3: Basic summary of each dataset employed. "CE": Controlled environment. "WRD/PHN": word or phonetic-level transcription. *: Not all utterances contain both types of transcription.

Dataset	Distribution	Speakers	Utterances	WRD/PHN	CE?
CSLU: Spoltech Brazilian Portuguese	Paid	477	8,080	Both*	No
Sid	Free	72	5,777	WRD	No
VoxForge	Free	+111	4,090	WRD	No
LapsBM1.4	Free	35	700	WRD	No

The center for spoken language understanding (CSLU): Spoltech Brazilian Portuguese dataset version 1.0 [119] includes recordings from several regions in Brazil. The *corpus* contains 477 speakers, totalizing 8080 utterances, consisting both of reading speech (for phonetic coverage) and response to questions (for spontaneous speech). A total of 2,540 utterances have been transcribed at word level and without alignment, and 5,479 utterances have been transcribed at phoneme level, with time alignments. All audio samples have been recorded at 44.1 kHz, and the acoustic environment was not controlled. As pointed in [120], some audio records do not have their corresponding transcriptions, and many of these records contain both transcriptions with errors, such as misspelling or typos. Also, they have used 189 phonetic symbols, many of them with few occurrences. For comparison, the TIMIT dataset has only 61 symbols.

The Sid dataset contains 72 speakers (20 are women), ranging from 17 to 59 years old. Each speaker has information about place of birth, age, genre, education, and occupation. All audios have been recorded at 22.05 kHz in a non-controlled environment. A total of 5,777 utterances has been transcribed at word level without time alignment. The sentences vary from spoken digits, single words, complex sequences, spelling of name and local of birth to phonetic covering, and semantically unpredictable sentences. Reading the transcribed utterances, we have noticed a recurrent error in the same chunk of sentences, probably due to automation of the process (*e.g.* parsing each transcription to the text file), forcing us to disregard these sentences.

The Voxforge [121] dataset is the most heterogeneous *corpus*. The idea of VoxForge is to distribute transcribed speech audio under GPL license, facilitating the development of acoustic models. Everyone can record specific utterances and send to them. Their Portuguese Brazilian language section contains at least 111 speakers[1] not all having information about genre or age. The audio files have been recorded at different sample rates ranging from 16 kHz to 44.1 kHz, and many records are in low-quality, presenting low signal-to-noise ratio (SNR). A total of 4,130 utterances were transcribed at word level.

Finally, LapsBM1.4 [122] is a dataset used by the Fala Brasil group of Federal University of Pará to evaluate large vocabulary continuous speech recognition (LVCSR) system in Brazilian Portuguese. It contains 35 speakers (10 women), each one with 20 unique utterances, totaling 700 utterances. Audio has been recorded at 22.05 kHz without environment control.

We need to define three distinct sets: train, validation, and test. It seems clear that the LapsBM should be used as the test set since it was created to evaluate LVCSR systems. Sid, VoxForge and the CSLU datasets will integrate the train set. Randomly choosing a part of the train set to form the validation set is not wise. We do not have control on which utterance belongs to a speaker from the VoxForge dataset, and we do not want to contaminate our validation set with speakers from the train set. Also, choosing one "sub-dataset" from the training set would bias our result. Furthermore, separating some speakers from the LapsBM to validate our results seems more natural, since each speaker has spoken unique utterances. We randomly split 21 speakers (7 women) from the LapsBM for the validation set and maintain the others (14 speakers) for the test set.

We have preprocessed the dataset to clean up wrong transcriptions, short records that could cause an error on the CTC (remembering that we need that our target

[1]Many utterances are sent anonymously.

sequences should be smaller than our input sequence), and many other defects as described above. Also, we are only interested in word-level transcription. Hence, we have discarded many CSLU utterances that did not have word-level transcriptions. All audio files were resampled to 16 kHz. The recordings' length is concentrated around 3 seconds but could span up to 25 seconds, as shown in Fig. 4.1a. The distribution of spoken utterances among speakers is shown in Fig. 4.1b. The highest peak is due to the anonymous contributions of VoxForge dataset. In Figs. 4.1c and 4.1d, we show the relation between the total number of utterances and the number of unique utterances in each sub-dataset as well the mean duration of those recordings. The summarization of our dataset is shown in Tab. 4.4. In Tab. 4.5, we show the evaluation of three major commercial systems with public API on our test set. The Google API demonstrates the best perfomance on both metrics (LER and WER). The difference between the best and worst LER is 4.52%, while this difference for WER is of about 13% — indicating that using a proper language model can make a huge difference in real ASR systems.

Gathering four different datasets, with various environments conditions and distinct hardware to acquire the signal, makes this dataset an ideal challenge for a deep learning approach, and far more stringent than the TIMIT dataset.

Table 4.4: Train, valid and test split of BRSD. In the second column "M/F" indicates the number of Males/Females. "LL" means the label length and "TD" indicates the total duration of all records.

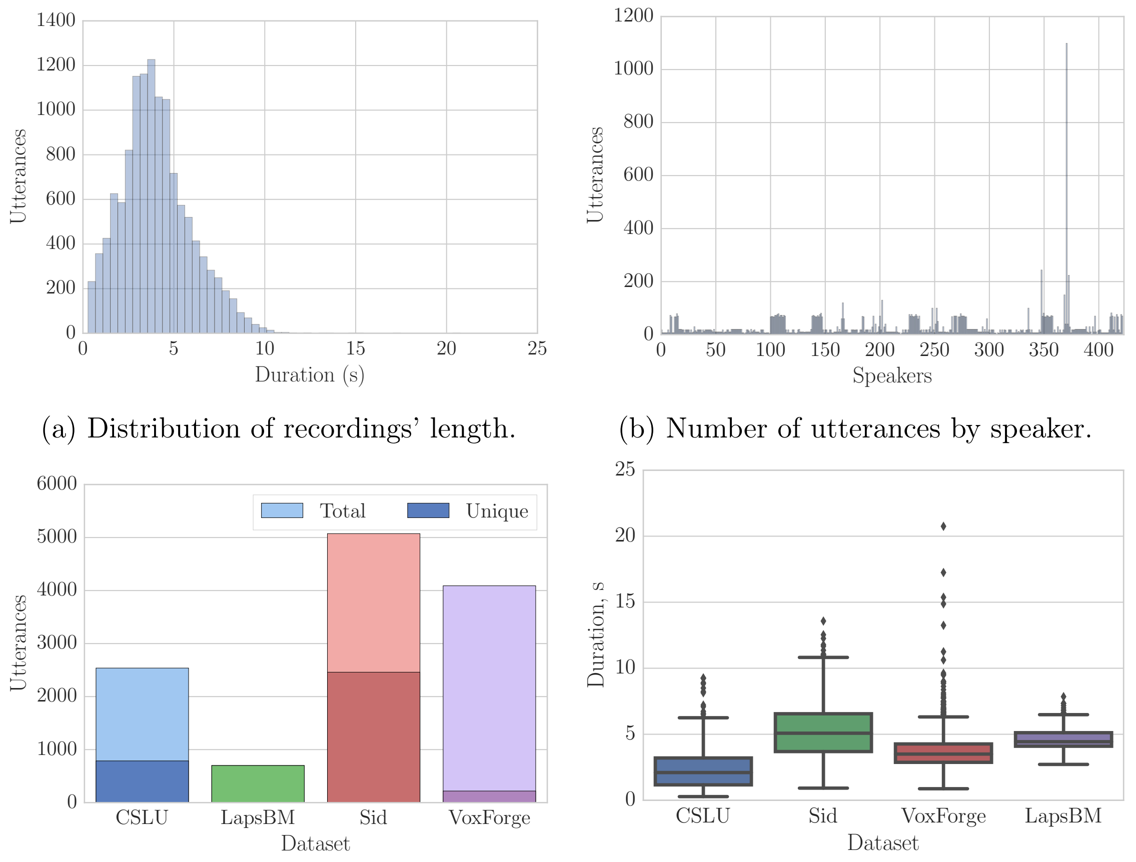
Dataset	Speakers (M/F)	Utterances (unique)	LL (min/max)	TD (hours)
Train	390 (150/29)	11,702 (3,437)	2/149	13.01
Valid	21 (14/7)	420 (420)	36/95	0.55
Test	14 (11/3)	280 (280)	39/87	0.35

Table 4.5: Results for the 4 commercial systems evaluated on our test set. Generated in 02/15/16.

System	LER	WER
Google API	10.25%	27.83%
IBM Watson	11.38%	35.61%
Microsoft Bing Speech	14.77%	40.84%

4.2 Hardware and software

All simulations were performed in a computer with a video card GTX 1080 8GB, 64GB DDR4 2133MHz of RAM, Intel$^{\text{TM}}$ Core i7 6850-K 3.6 GHz processor, and using Ubuntu 16.04 as the operational system.



(a) Distribution of recordings' length.

(b) Number of utterances by speaker.

(c) Utterances per dataset.

(d) Mean duration of recordings in each dataset.

Figure 4.1: Some statistics of BRSD.

Regarding software, there are a plenty of solutions and specific frameworks for deep learning. We have searched for several toolkits that have been developed by many researchers in the Deep Learning area, among which Torch [123], Theano [124], PyLearn2 [125], Caffe [126], CNTK [127], and Tensorflow [128] are the top ones.

Each tool has its advantages and disadvantages[2], and at the end, we chose one that had great coverage of convolutional and recurrent layers, had a highly optimized code for GPU, and was not developed to be used with pipelines or configuration files (like CNTK[3]). Also, we already had some familiarities with Python. We chose Tensorflow as the default toolkit, powered by Keras [129] — a nice front-end engine for Tensorflow that encapsulates many codes.

At the time of this dissertation, Tensorflow did not have CTC GPU-based implementation yet. Therefore, we have used the freely available CTC GPU-based code [130] made by Baidu Research for Deep Speech 2 model. Also, adopting Ten-

[2]In `https://github.com/zer0n/deepframeworks` one finds very nice comparison between different deep learning toolkits.

[3]Nowadays, CNTK toolkit has Python bindings, but when we began this dissertation they were not available.

sorflow and Keras to perform speech recognition was not straightforward as we thought. Since then, we have already made 2 contributions to the Tensorflow code, and 3 bug fixes to the Keras code. We have made our code freely available[4] under MIT License for the sake of reproducibility.

4.3 Case study: Graves' model

Firstly, as we described at the beginning of the chapter, we will validate our implementation by replicating the Graves' model [11].

Remember that the audio data was preprocessed using a frame signal of 10 ms, with 5 ms of overlap, using 12 MFCCs from 26 log filter-bank channels. The log-energy and the first derivative were also added, giving a vector of 26 coefficients per frame.

Instead of using the complete test set as was proposed by the paper, here we will use the core test set, because the results will be more reliable and unbiased, and 400 utterances of the complete test set will be employed as validation data. As a target, they used 61 phonemes (+ blank label) that came from the TIMIT dataset transcriptions. Their topology, described in Sec. 3.6.1, is shown in Tab. 3.1.

As in the paper, all weights were initialized with uniform random noise ranging from -0.1 to 0.1 and the forget bias was kept at zero. The SGD+Nesterov was used with a learning rate of 10^{-4} and a momentum of 0.9 in a batch size of one sample.

Using a batch size of 1 is time-consuming. We have run their model only in CPU, and the forward and backpropagation through the entire training set took about 21 minutes. Considering 200 epochs, each simulation would take about 70 hours, almost 3 days. This is unacceptable because the dataset is tiny compared to BRSD and we can spend several weeks fine-tuning our hyperparameters. We can, however, run the simulations with a bigger batch size, thus enjoying the optimized matrix operations.

4.3.1 Increasing the speedup with bigger batch size

Unfortunately, we had no time to run the simulations several times. Thus, the results are for only one simulation for each batch size. As far as we have noticed, there are no larger deviations through several runs. We ran the same model as cited above, but with a variable batch size of $\{1, 2, 8, 16, 32, 64, 256\}$ and the results are shown in Fig. 4.2a.

First of all, this plot is not fair. Despite the increasing batch size, we have a pitfall: we do fewer gradient updates in each epoch but at the same time, each

[4]Our code is available at `https://github.com/igormq/asr-study`.

epoch is faster (due to the use of larger matrices and therefore better use of the computational time of the CPU). Calculating the time spent in each epoch and calculating the median we can see a very nice speed up, as given in Tab. 4.6. The normalized version of Fig. 4.2a is shown in Fig. 4.2b. As one can see, using a larger batch size we get a better gradient estimation, so the cost evolves more smoothly than using a batch size of 1. However, as we increase our batch size we do less weight updates per epoch. On the other hand, using a batch size of length 1, our training could be too slow and noisy to converge to some minimum. Our best batch size was 32.

Table 4.6: Speed up achieved increasing the batch size.

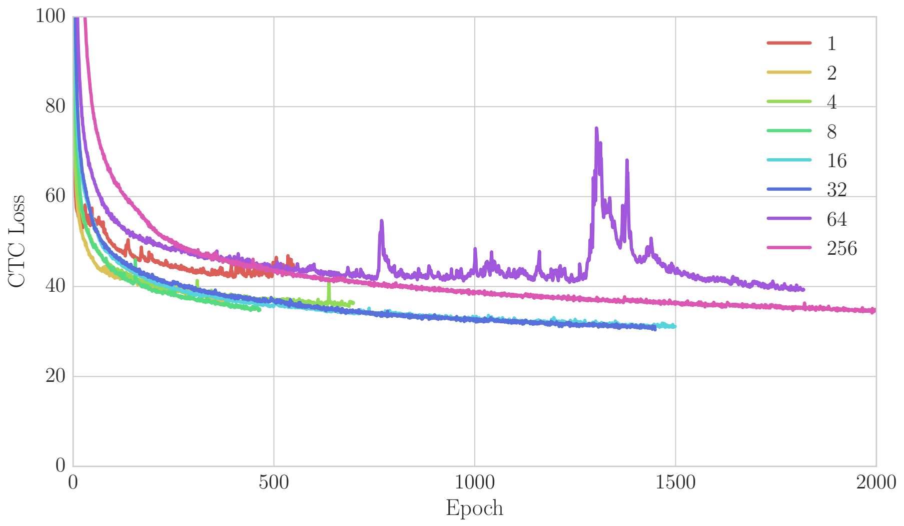
Batch Size	Speedup	Median (minutes)
1	1.00×	21.93
2	0.99×	22.25
8	1.54×	14.23
16	2.01×	10.91
32	2.36×	9.30
64	2.71×	8.09
256	2.91×	7.52

In Fig. 4.2b, the bias in the CTC loss between the valid set and the train set gets worse, showing that the network was not able to generalize well, indicating that the use of random Gaussian noise at the input of network was not a good regularizer. Getting the best model, that occurred at epoch 706, testing it against the test set, and applying the beam search decoder with a width of 100, we have got an LER of 29.64%, which is slightly better than the results presented in the original paper (of 30.51% ± 0.19%). However, remember that our training dataset is a little bigger, our test set is more difficult, and we have decoded using the beam search.

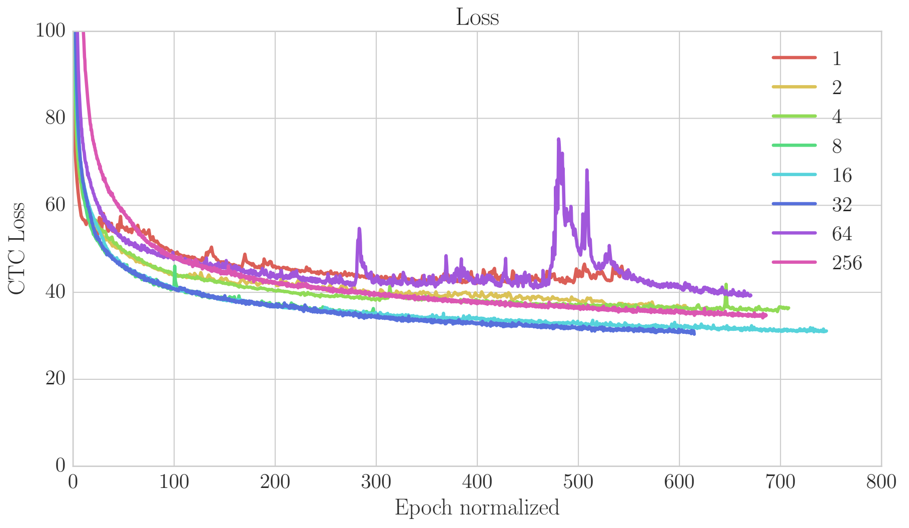
4.4 Applying the knowledge: building an end-to-end ASR that understands Brazilian Portuguese

Now that we have all the tools and a working implementation, we can start developing our model. We start from the previous model, and proceed by doing gradual modifications.

Our input is slightly bigger than the described above, as mentioned in Sec. 3.7. The audio data was preprocessed with a frame window of 25 ms, hop of 10 ms, using 12 MFCCs from 40 log filter banks in mel-scale. Also, the log energy was added as



(a) CTC loss per epoch for different batch sizes.



(b) CTC loss per normalized epoch for different batch sizes

Figure 4.2: Evolution of CTC loss for various batch sizes.

well as the first and second derivatives (delta and double delta coefficients), yielding a vector of size 39[5]

The training was carried out with Adam (it is faster than SGD+Nesterov), and a learning rate η of 10^{-3} was chosen after a careful search. All simulations were performed in 30 epochs[6] with a batch size of 32, which takes an average of one and a half day each simulation in our GPU. The recurrent weights of LSTM were initialized with orthogonal matrices sampled from a normal distribution [117] — the eigenvalues of the Jacobian matrices will be 1, and it helps to alleviate the vanishing gradient problem over long time steps. The non-recurrent weights were initialized from a uniform distribution following the Xavier initialization method. Also, the

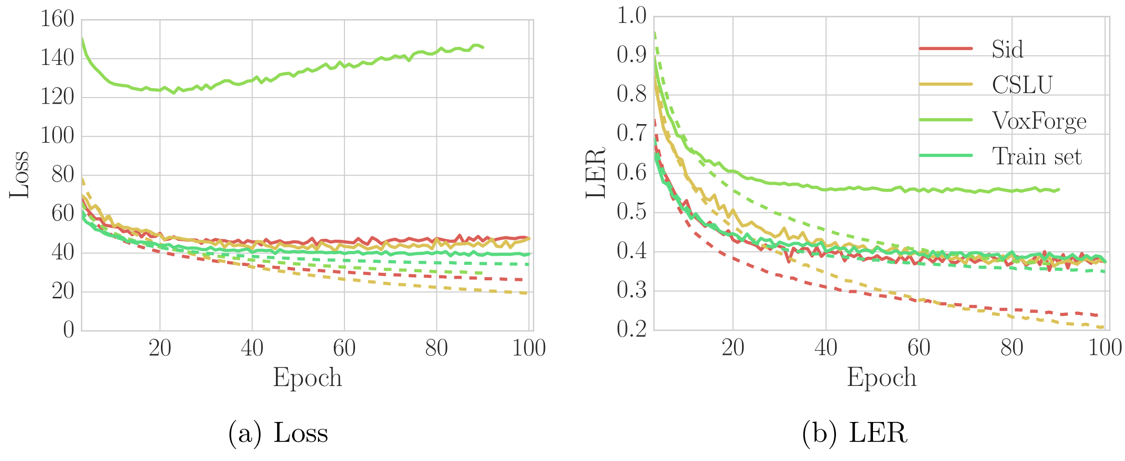
[5]This choice of preprocessing differs from [11] in order to adopt a more common approach that we have found through several papers.

[6]We have set this limit due to the time constraint.

forget bias was set to 1. We added the same regularization method as in Graves' model — white Gaussian noise with standard deviation of 0.6 — to the inputs of our model.

Our model outputs the probability of character emission instead of phonemes. Thus, our label set is $\{a, \ldots, z, \text{space}, \varnothing\}$, where "space" is used to delimit the word boundaries. We have mapped the target sequences in our dataset to rely only on this label set — punctuation, hyphens, and stress marks were removed/mapped to their corresponded label (*e.g.* á - a, ç - c).

Running the same topology used by Graves *et al.*, we evaluate each subset of the training set against our validation set, as shown in Fig. 4.3. The solid lines represent the results in the validation set, while the dashed lines are the results in the training (sub)set. As we can see in Fig. 4.3b, each subset has a large bias — high difference between training and test values — indicating that it does not generalize well. The entire dataset, however, presented the best validation values (loss and LER) and the lowest bias; moreover, the training loss is not evolving towards zero, indicating that we should increase our model capacity. We have tried to increase the regularization by increasing the standard deviation, but it did not improve the results. It is worth mentioning that while it seems that the training with VoxForge is overfitting in Fig. 4.3a, the LER is still decaying (Fig. 4.3b). It is due to the fact that the CTC loss is just a proxy to the real non-differentiable evaluation, the LER. Hence, it is pretty common finding curves like that, but the inverse (LER raising while loss is decaying) must not happen. For the early stopping algorithm, we use the LER value and not the CTC loss.



(a) Loss (b) LER

Figure 4.3: Results for training with the subset instead of the full training set. The validation set was kept the same. Solid lines show the results for the validation set while dashed lines show the results during training.

4.4.1 Stacking more layers

We can increase the model capacity in four ways: stacking more recurrent layers; using more dense layers; adding convolutional layers; or increasing the number of hidden units. Dense layers add many parameters, and they are difficult to regularize. Even if convolutional layers have shown promising results, we did not investigate their application. In Fig. 4.4, we show the results for a different number of recurrent layers with 256 hidden units each. Clearly, stacking more layers is advantageous: LER falls from 44.97% to 33.92% as the number of layers increases from 1 to 5. We almost did not see any improvements adding the 6th layer, and for a 7-layer model we can observe a slight deterioration. This effect is probably due to the vanishing gradient problem getting worse as we stack more layers.

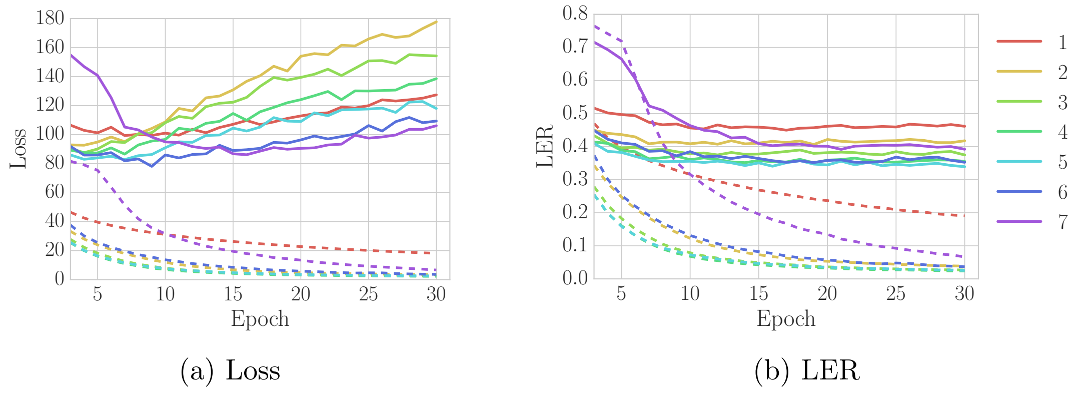
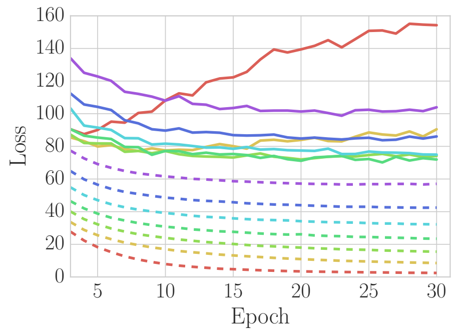


Figure 4.4: Different models ranging the number of layers from 1 to 7.

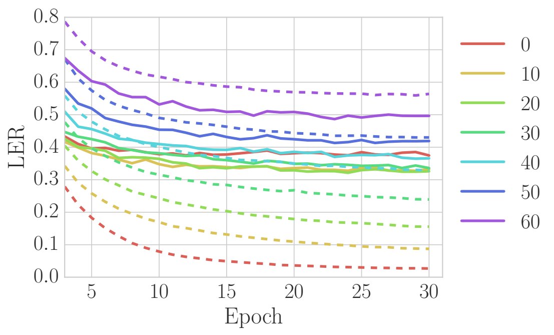
4.4.2 Regularization

We have investigated two methods of regularization: weight decay and dropout. As we have already seen, applying dropout to RNN is not straightforward. Indeed, we gave some directions in Sec. 2.12.1, which led us to the variational dropout. In Fig. 4.5a and 4.5b, we applied variational dropout with the same dropout probability to the recurrent and non-recurrent weights and kept the weight decay in zero, and in Fig. 4.5c and 4.5d we used weight decay as regularizer and no dropout. As we can see, both do the same thing, preventing overfitting and giving a better generalization. Dropout, however, seems to give better generalization than weight decay (32.15% of LER against 35.51%). Cross-validating them with our best previous model — 5 BLSTM layers with 256 hidden units — we found that a weight decay of 10^{-4} and dropout $p = 0.2$ gave the best result, achieving an LER of 29.50% in the validation set.

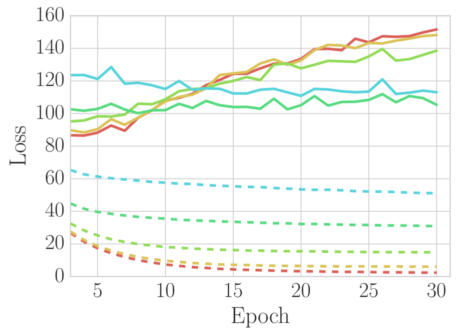
Also, we have instigated using zoneout (Sec. 2.12.2) instead of dropout. Adapted to LSTM, zoneout is applied separately to the cell state (Eq. (2.65)) and the hidden state (Eq. (2.66)), each one having its own “zoneout” probability. We applied the same zoneout probability for both equations, though. The results are shown in Fig. 4.6 and zoneout does not seem to make any effect, even with high values (50%). The authors that proposed zoneout [76] carefully chose different zoneout probabilities for the cell state and hidden state; not doing that seems to be the cause of our failure. Also, they have only tested models with one layer. All in all, further investigation with careful choices of both probabilities are needed as well as the investigation of zoneout applied to models with several recurrent layers.



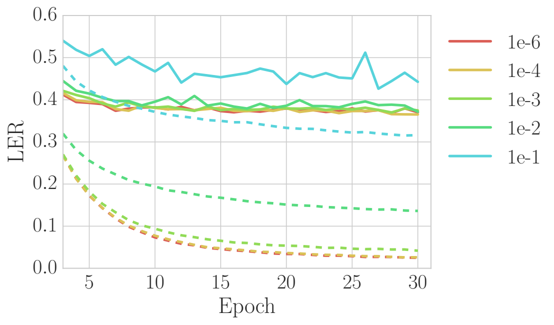
(a) Loss



(b) LER

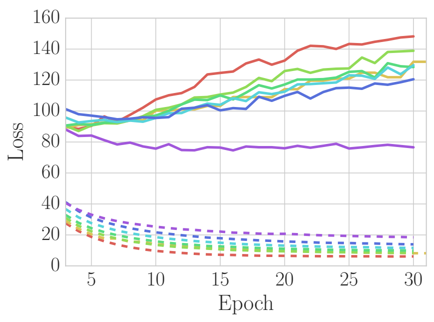


(c) Loss

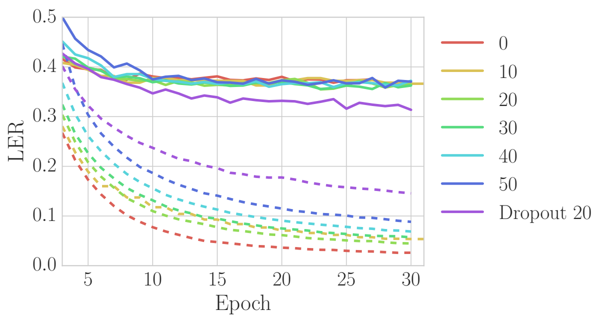


(d) LER

Figure 4.5: Different values of dropout (top) and weight decay (bottom). The trained model has 3 BLSTM layers with 256 hidden units each.



(a) Loss



(b) LER

Figure 4.6: Different values for the zoneout probability. Both cell state and hidden state zoneout probability are kept the same through the layers.

4.4.3 Multiplicative integration and layer normalization

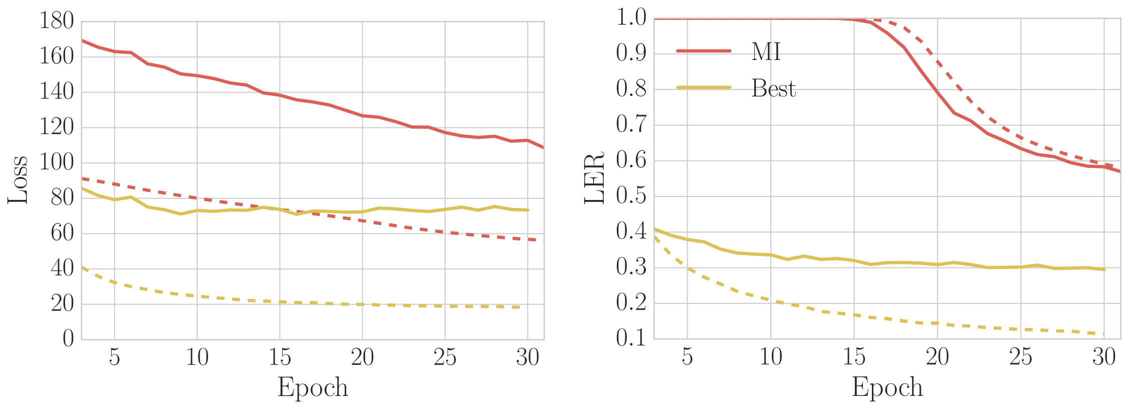
Multiplicative integration (MI) and layer normalization (LN) are techniques developed to speed up the training, as described in Secs. 2.12.5 and 2.12.4.

Multiplicative integration inserts three new hyperparameters per recurrent layer ($\boldsymbol{\alpha}_{\text{MI}}$, $\boldsymbol{\beta}_{\text{MI}}$, $\boldsymbol{\gamma}_{\text{MI}}$). In Fig. 4.7, we have applied MI with its hyperparameters initialized with a vector of ones on our best topology — 5 BLSTM with 256 hidden units each, weight decay of 10^{-4}, and variational dropout of 20%. Again, Wu *et al.* citeWu:2016vm have carefully tuned the hyperparameters for each task they have presented, although they have tested MI with several layers. In their speech recognition problem, using the EESEN model (Sec. 3.6.3), they obtained better results with MI, but they did not specify the hyperparameters they used.

Finally, we also tried employing LN in our model. Adapting to LSTM, Ba *et al.* [77] proposed the following adaptions:

$$\begin{aligned} \boldsymbol{a} &= \text{LN}(\boldsymbol{W}_x^T\boldsymbol{x}^{(t)}; \boldsymbol{\alpha}_{\text{LN}_1}, \boldsymbol{\beta}_{\text{LN}_1}) + \text{LN}(\boldsymbol{W}_h^T\boldsymbol{h}^{(t-1)}; \boldsymbol{\alpha}_{\text{LN}_2}, \boldsymbol{\beta}_{\text{LN}_2}) \\ \boldsymbol{h}^{(t)} &= \boldsymbol{o} \odot \tanh(\text{LN}(\boldsymbol{c}^{(t)}; \boldsymbol{\alpha}_{\text{LN}_3}, \boldsymbol{\beta}_{\text{LN}_3}) \end{aligned} \tag{4.1}$$

where $\text{LN}(\cdot, \boldsymbol{\alpha}_{\text{LN}}, \boldsymbol{\beta}_{\text{LN}})$ is the layer normalization operation. We have tied all gains and bias initializations, *i.e.* the same initialization for bias/gains in all layers. Our best model did not converge with layer norm, and proper investigation is needed.

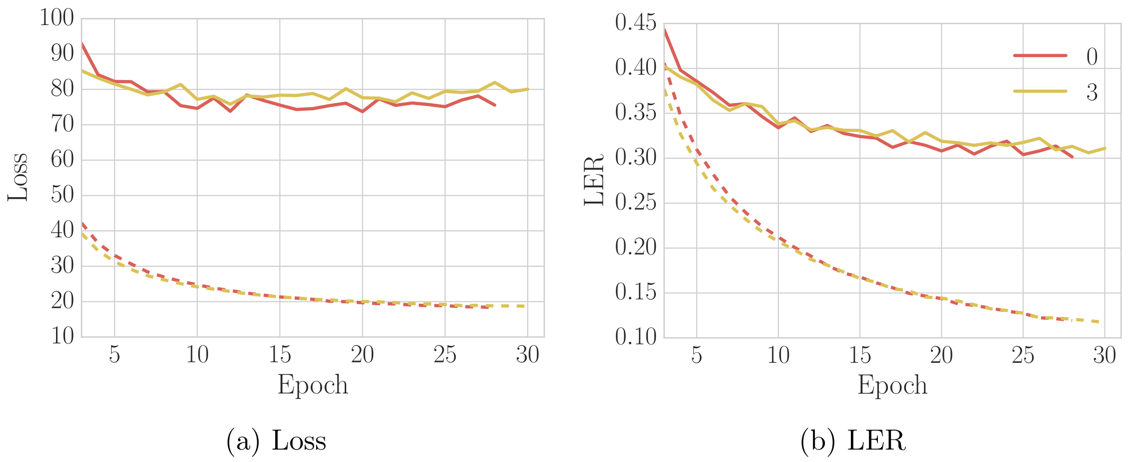


(a) Loss (b) LER

Figure 4.7: The best model with and without MI.

4.4.4 Broader context

Both Maas [90], and Deep Speech models [91, 115] used as input their preprocessed signal with a context window (±10 in Maas and ±9 in Deep Speech), facilitating the recurrent network to gather information about short past and future contents. In Fig. 4.8, we have also applied to our best model a larger input using a context of ±3. We did not investigate using broader contexts due to the limits of GPU memory. As we can see, we had only a minor improvement; moreover, we have increased the memory consumption and the number of parameters (leading to a higher training time). There is limited evidence for a conclusion; however, both models used vanilla RNN[7], which has the problem of learning long-term dependencies, and using a broader context could temper it. Nevertheless, our model employs LSTM layers, which by themselves are more capable of learning long-term dependencies, indicating that using a broader context may not be helpful in our case, as shown in Fig. 4.8.



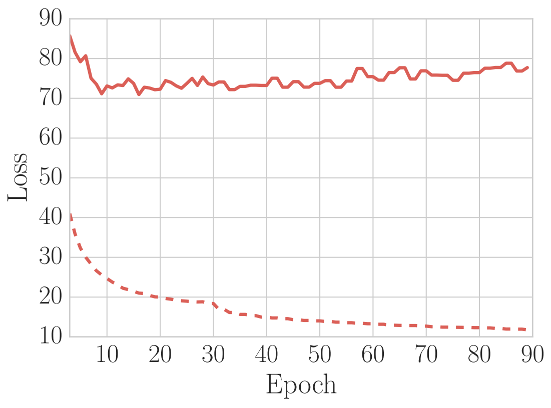
(a) Loss (b) LER

Figure 4.8: Comparison of our best model without a context and with a context of ±3 time steps.

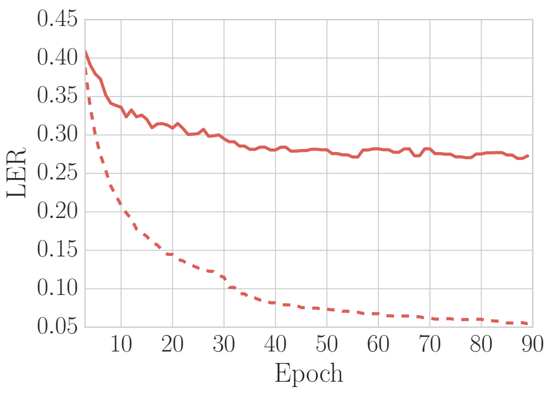
4.4.5 Final model

The best model after several design choices employs 5 layers of BLSTM with 256 units, no context window, and as regularizers variational dropout with dropout probability of 20% and weight decay of 10^{-4} applied to every recurrent layer. We have trained this model for more epochs, as shown in Fig. 4.9. Our best validation result occurs in epoch 58, with an LER of 26.95%. In our test set, we have decoded the sentence using the beam search with a beam width of 100. We obtained an LER of 25.13%, which is close to the result that Maas [90] has obtained without a language model (Sec. 3.6.2), but almost 11% worse when compared to commercial systems (Tab. 4.5), which have several pipelines, including a lexicon and language models.

[7]Actually, they have changed the tanh activation to clipped ReLU.



(a) Loss



(b) LER

Figure 4.9: Best model training.

4.4.6 Analyzing the transcriptions

We have chosen some transcriptions that are worth to analyze. We have found out that some errors that occur to our transcription are phoneticaly reasonable, as pointed in [91]. In Tab. 4.7, our network changed “flexa” (Lucia’s mid name) with “flecha” (arrow), which have the same sound. This misspelling might occur because “flecha” has a higher number of occurrences in the training set.

Another interesting fact is demonstrated in Tab. 4.8. Our model changed one letter in the word “e**x**plicacoes” (explanation), substituting “x” by “s”. This substitution could be explained due to region dialects. People from Rio de Janeiro, for example, pronounce the “x” in “explicacoes” like “sh” as in /leash/, while people from others states tend to pronounce as a sibilant “s” as in /juice/. Indeed, listening to the dataset recordings, we have found that the majority of the speakers are not from Rio de Janeiro, which explains the behavior of our network.

Finally, we also have found that our network made some “mistakes” by transcribing some speakers’ peculiarities, depicted in Tab. 4.9. While the ground truth is “tem se” the network outputs “ten ci”, which is a reasonable mistake, since both constructions sound equal. It seems that the network transcribed the sentence in the way the speaker has spoken. All in all, such misspellings could be easily corrected using a proper language model[8].

Table 4.7: Comparison between the ground truth sequence and the sequence transcribed by our best model. The network misspelling that has the same phonetic sound is highlighted.

Truth	esta instalado na casa do avo de lucia fle**x**a de lima
Network	esta estalado na casa do arode duscia fle**ch**a dima

[8]With the exception of Lucia’s mid name, of course.

Table 4.8: Comparison between the ground truth sequence and the sequence transcribed by our best model. The network misspelling that could be explained by the difference in region dialects is highlighted.

Truth	ele podia dar e**x**plicacoes praticas para sua preferencia por faroestes
Network	ele putiadar e**s**plicacoes cratifos para soubre ferencia por faraeste

Table 4.9: Comparison between the ground truth sequence and the sequence transcribed by our best model. The network misspelling that could be explained by the way that the speaker speaks is highlighted.

Truth	**tem se** uma receita mensal de trezentos e quarenta mil dolares
Network	**ten ci** uma receira mensalbe trezentos e quarenta mil bolarte

Chapter 5

Conclusions and future works

A lot of information was given through the chapters. We have started writing about neurons, layers, and how they connected to each other to perform some real calculation. We have described the universal approximation theorem and pointed that a feedforward network with a single hidden layer and enough number of neurons can approximate any continuous function. Stacking more than two layers, however, have demonstrated to learn better representations than the shallow ones when the input has a high number of features (*e.g.* speech, pixels of an image) [21]. Deep learning is only a new fancy name for neural networks with more than a couple of layers, but it has not been a reality until the development of parallel computation using GPUs and the Hinton's idea.

Training deep models can be a lot easier with properly initialization of the weights, and for that, we have developed some mathematical grounds for the Xavier [24] and He [19] initialization. Batch norm, a technique to alleviate the internal covariate shift, is a must for any machine learning practitioner. Due to the non-linearity of the model, a gradient-based method was required to training the network to perform the desired calculation. For that, we have associated neural networks to computational graphs, and we have shown how to perform the back-propagation on them. Then, simple methods like stochastic gradient descent and its flavors (Momentum [31], RMSProp [33], and Adam [34]) should be used to minimize the loss by adjusting the weights.

The vanilla neural network can not model well sequences, and for that, we have described the recurrent networks — powerful models that can handle sequences at input and sequences at the output. Recurrent neural networks, however, do not perform well for long sequences due to the vanishing and exploding gradient problems. For the exploding gradient, gradient clipping is a simple but very effective tool. Long Short-Term Memory networks were introduced to deal with the vanishing gradient, using several gates to control the gradient flow.

Recurrent neural networks do not generalize well, are more prone to overfitting, and regularizing recurrent networks have not been proved a trivial task. Indeed, quite an effort has been made through the years to find an ultimate method. We have discussed new approaches like variational dropout [75] and zoneout [76] for that. Another common problem of recurrent nets is that their training is quite slow, and to cut down the training time we have shown recent methods like batch recurrent norm [28], layer norm [77], and multiplicative integration [78].

The central theme of this dissertation is the construction of an all-neural speech recognition system for the Brazilian Portuguese language. Unfortunately, using the recurrent network for that task is only possible if we have a dataset with frame-alignment transcriptions, which is costly and time demanding. To address that problem, we have shown Graves' work [53] — the connectionist temporal classification method. CTC works by computing all possible paths of a sequence of labels and intuitively can be interpreted as the soft cross-entropy loss function by adding all possible paths instead of one path.

Decoding the sequence after training a CTC-based model is not trivial. Indeed, using a naïve decoding could lead to errors. We have shown that a more powerful decoding could be achieved with the beam search decoder [90]. The decoder by itself does not have any information about the peculiarities of the language (*e.g.* how to spell words), and we could improve our recognizer by adding a language model.

Subsequently, we have discussed several CTC-based models that arose in the last years. Graves *et al.* [11], the first successful end-to-end solution showed us the advantage of deep models — dropping the LER from 23.9% to 18.4% as the number of layers increases from one to five. Maas *et al.* [90], one of the first successful shot to convert speech directly to characters, introduced a different topology by mainly focusing on dense connections with minimum recurrent layers, and by using clipped ReLU as activation for all layers (including the recurrent). It is worth mentioning that they achieved a result of 27.7% without a proper language model. EESEN, which it is considered the (feasible) state-of-the-art end-to-end CTC-based methods, proposed decoding the sequences using a weighted finite-state transducer (WFST) [57], which is a clever way to incorporate the lexicons and language models into CTC decoding. They have shown comparable WERs with standard hybrid DNN system. Finally, the last couple of models studied, Deep Speech 1 and 2 made by Baidu Research, are huge by their nature: models with hundreds of millions of parameters trained over a dataset of thousand of hours. They have shown similar results to humans.

Afterwards, we have validated our implementation by replicating the first Graves' model. We have demonstrated that we could achieve a speedup of 2.36 by increasing the batch size from 1 to 32. Then, we have achieved a slightly lower LER than Graves due to our bigger training set and the better decoder, showing that our implementation is working. Furthermore, based on the case study model, we have built our model by doing gradual modifications. Firstly, we increased the number of hidden layers from 100 to 256. Secondly, we removed the white Gaussian noise at the input, since it has not shown any improvement. Then, we tested bigger models, showing that a 5-layer BLSTM gave the best results. Next, we cross-validated different values of weight decay and variational dropout, and found out that the best configuration is 10^{-4} for weight decay and a variational dropout of 20%. Also, we demonstrated that using a broader context gave us no improvement because the LSTM network already handles long-term dependencies quite well. We have tried applying zoneout, multiplicative integration and layer norm to our model, without any success yet.

Given enough training time to our best model, we achieved an LER of 26.95% in the validation set. In the test set, we have decoded our sequences by applying the beam search decoder with a beam width of 100 and no language model. Thus, we have achieved an LER of 25.13%, which is comparable to the results of Maas [90]; however, we have a long way of improvements to achieve results similar to those of commercial systems (Tab. 4.5).

5.1 Future works

There is a lot of work to be done. Firstly, we have not succeeded in applying zoneout, LN, and MI to our model, and further investigation (with a broader range of hyperparameters) must be carried out to validate these methods. Using a language model is an essential tool for any reliable ASR system, and yet we have applied none to our system. Further investigation of different language models (*e.g.* RNN-based, WFST) is necessary. Also, we have not stressed our topology by adding convolutional layers, or employing other kinds of RNN structures (*e.g.* GRU) or different kinds of input (*e.g.* raw audio, and filter banks without MFCCs computation).

Increasing our training data is also required. Besides being bigger than TIMIT, it is far away from more used datasets, like the SwitchBoard (over 300 hours of speech). Furthermore, we did not investigate adding punctuation, accents, and hyphens to our softmax layer, which could disambiguate some Portuguese Brazilian words, and seems more logical to be used.

Moreover, encoder-decoder based models with attention mechanism is a raising area for end-to-end solutions in speech recognition, and future work must be done with these models.

Bibliography

[1] KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E. "ImageNet classification with deep convolutional neural networks". In: *Advances in Neural Information Processing Systems*, pp. 1097–1105, Lake Tahoe, USA, December 2012.

[2] LECUN, Y., BOTTOU, L., BENGIO, Y., et al. "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, v. 86, n. 11, pp. 2278–2324, November 1998.

[3] GOODFELLOW, I., BENGIO, Y., COURVILLE, A. *Deep learning.* Adaptive computation and machine learning series. Cambridge, England, MIT Press, 2016.

[4] OUYANG, W., WANG, X. "Joint deep learning for pedestrian detection". In: *IEEE International Conference on Computer Vision*, pp. 2056–2063, Sydney, Australia, December 2013.

[5] CHEN, C., SEFF, A., KORNHAUSER, A., et al. "DeepDriving: Learning affordance for direct perception in autonomous driving". In: *IEEE International Conference on Computer Vision*, pp. 2722–2730, Santiago, Chile, December 2015.

[6] HINTON, G. E., DENG, L., YU, D., et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups", *IEEE Signal Processing Magazine*, v. 29, n. 6, pp. 82–97, November 2012.

[7] SUK, H.-I., LEE, S.-W., SHEN, D., et al. "Hierarchical feature representation and multimodal fusion with deep learning for AD/MCI diagnosis", *Neuro Image*, v. 101, pp. 569–582, November 2014.

[8] DAVIS, K., BIDDULPH, R., BALASHEK, S. "Automatic recognition of spoken digits", *The Journal of the Acoustical Society of America*, v. 24, n. 6, pp. 637–642, November 1952.

[9] RABINER, L. R. "A tutorial on hidden markov models and selected applications in speech recognition", *Proceedings of the IEEE*, v. 77, n. 2, pp. 257–286, February 1989.

[10] DAHL, G. E., YU, D., DENG, L., et al. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition", *IEEE Transactions on Audio, Speech, and Language Processing*, v. 20, n. 1, pp. 30–42, January 2012.

[11] GRAVES, A., FERNÁNDEZ, S., GOMEZ, F. J., et al. "Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks". In: *International Conference on Machine Learning*, pp. 369–376, Pittsburgh, USA, June 2006.

[12] CHOROWSKI, J., BAHDANAU, D., SERDYUK, D., et al. "Attention-based models for speech recognition". In: *Advances in Neural Information Processing Systems*, pp. 577–585, Montreal, Canada, December 2015.

[13] CYBENKO, G. "Approximation by superpositions of a sigmoidal function", *Mathematics of Control, Signals, and Systems*, v. 2, n. 4, pp. 303–314, December 1989.

[14] HORNIK, K. "Approximation capabilities of multilayer feedforward networks", *Neural Networks*, v. 4, n. 2, pp. 251–257, January 1991.

[15] HAYKIN, S. *Neural networks and learning machines*. Pearson Education, 2009.

[16] LECUN, Y., KANTER, I., SOLLA, S. A. "Second order properties of error surfaces". In: *Advances in Neural Information Processing Systems*, pp. 918–924, Denver, USA, November 1990.

[17] NAIR, V., HINTON, G. E. "Rectified linear units improve restricted Boltzmann machines". In: *International Conference on Machine Learning*, v. 30, pp. 807–814, Haifa, Israel, June 2010.

[18] MAAS, A. L., HANNUN, A., NG, A. Y.-T. "Rectifier nonlinearities improve neural network acoustic models". In: *International Conference on Machine Learning*, v. 30, p. 1–6, Atlanta, USA, June 2013.

[19] HE, K., ZHANG, X., REN, S., et al. "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification". In: *IEEE International Conference on Computer Vision*, pp. 1026–1034, Santiago, Chile, December 2015.

[20] BISHOP, C. M. *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York, USA, Springer Verlag, 2006.

[21] BENGIO, Y. "Learning deep architectures for AI", *Foundations and Trends in Machine Learning*, v. 2, n. 1, pp. 1–127, January 2009.

[22] HINTON, G. E., OSINDERO, S., TEH, Y.-W. "A fast learning algorithm for deep belief nets", *Neural Computation*, v. 18, n. 7, pp. 1527–1554, July 2006.

[23] FISCHER, A., IGEL, C. "Training restricted Boltzmann machines: An introduction", *Pattern Recognition*, v. 47, n. 1, pp. 25–39, January 2014.

[24] GLOROT, X., BENGIO, Y. "Understanding the difficulty of training deep feedforward neural networks". In: *International Conference on Artificial Intelligence and Statistics*, v. 9, pp. 249–256, Sardinia, Italy, May 2010.

[25] HE, K., ZHANG, X., REN, S., et al. "Deep residual learning for image recognition". In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, Las Vegas, USA, June 2016.

[26] IOFFE, S., SZEGEDY, C. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *International Conference on Machine Learning*, v. 37, pp. 1–9, Lille, France, July 2015.

[27] LAURENT, C., PEREYRA, G., BRAKEL, P., et al. "Batch normalized recurrent neural networks". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2657–2661, Shanghai, China, March 2016.

[28] COOIJMANS, T., BALLAS, N., LAURENT, C., et al. "Recurrent batch normalization". February 2016. eprint arXiv:1502.03167v3.

[29] SRIVASTAVA, N., HINTON, G. E., KRIZHEVSKY, A., et al. "Dropout: A simple way to prevent neural networks from overfitting", *Journal of Machine Learning Research*, v. 15, pp. 1929–1958, June 2014.

[30] HINTON, G. E., SRIVASTAVA, N., KRIZHEVSKY, A., et al. "Improving neural networks by preventing co-adaptation of feature detectors". July 2012. eprint arXiv:1207.0580v1.

[31] QIAN, N. "On the momentum term in gradient descent learning algorithms", *Neural Networks*, v. 12, n. 1, pp. 145–151, January 1999.

[32] SUTSKEVER, I., MARTENS, J., DAHL, G. E., et al. "On the importance of initialization and momentum in deep learning". In: *International Conference on Machine Learning*, v. 28, pp. 1139–1147, Atlanta, USA, June 2013.

[33] HINTON, G. E., TIELEMAN, T. "Neural networks for machine learning". `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf`. Lecture 6a. Slide 29. Accessed: 2017-02-24.

[34] KINGMA, D. P., BA, J. "Adam: A method for stochastic optimization". In: *international conference for learning representations*, pp. 1–15, San Diego, USA, May 2015.

[35] LECUN, Y., BOSER, B. E., DENKER, J. S., et al. "Backpropagation applied to handwritten zip code recognition", *Neural Computation*, v. 1, n. 4, pp. 541–551, Winter 1989.

[36] RAZAVIAN, A. S., AZIZPOUR, H., SULLIVAN, J., et al. "CNN features off-the-shelf: An astounding baseline for recognition". In: *ieee conference on computer vision and pattern recognition workshops*, pp. 512–519, Columbus, USA, June 2014.

[37] YOSINSKI, J., CLUNE, J., BENGIO, Y., et al. "How transferable are features in deep neural networks?" In: *Advances in Neural Information Processing Systems*, pp. 3320–3328, Montreal, Canada, December 2014.

[38] DAI, J., LI, Y., HE, K., et al. "R-FCN: Object detection via region-based fully convolutional networks". In: *Advances in Neural Information Processing Systems*, pp. 379–387, Long Beach, USA, December 2016.

[39] FARABET, C., COUPRIE, C., NAJMAN, L., et al. "Learning hierarchical features for scene labeling", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 35, n. 8, pp. 1915 – 1929, August 2013.

[40] HARIHARAN, B., ARBELÁEZ, P. A., GIRSHICK, R. B., et al. "Simultaneous detection and segmentation". In: *European Conference on Computer Vision*, pp. 297–312, Zurich, Switzerland, September 2014.

[41] HARIHARAN, B., ARBELÁEZ, P. A., GIRSHICK, R. B., et al. "Hypercolumns for object segmentation and fine-grained localization". In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 447–456, Boston, USA, June 2015.

[42] DAI, J., HE, K., SUN, J. "Instance-aware semantic segmentation via multi-task network cascades". In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 3150–3158, Las Vegas, USA, June 2016.

[43] TAIGMAN, Y., YANG, M., RANZATO, M., et al. "Deepface: Closing the gap to human-level performance in face verification". In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1701–1708, Columbus, USA, June 2014.

[44] SUN, Y., LIANG, D., WANG, X., et al. "DeepID3: Face recognition with very deep neural networks". 2015. eprint arXiv:1502.00873v1.

[45] FARFADE, S. S., SABERIAN, M. J., LI, L.-J. "Multi-view face detection using deep convolutional neural networks". In: *ACM International Conference on Multimedia Retrieval*, pp. 643–650, Shanghai, China, June 2015.

[46] LIN, M., CHEN, Q., YAN, S. "Network in network". March 2013. eprint arXiv:1312.4400v3.

[47] SIMONYAN, K., ZISSERMAN, A. "Very deep convolutional networks for large-scale image recognition". April 2014. eprint arXiv:1409.1556v6.

[48] CHO, K., VAN MERRIENBOER, B., GÜLÇEHRE, Ç., et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *Conference on Empirical Methods in Natural Language Processing*, pp. 1724–1734, Doha, Qatar, October 2014.

[49] WU, Y., SCHUSTER, M., CHEN, Z., et al. "Google's neural machine translation system: Bridging the gap between human and machine translation". October 2016. eprint arXiv:1609.08144v2.

[50] FIRAT, O., CHO, K., BENGIO, Y. "Multi-way, multilingual neural machine translation with a shared attention mechanism". In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 866–875, San Diego, USA, June 2016.

[51] MIKOLOV, T., KARAFIÁT, M., BURGET, L., et al. "Recurrent neural network based language model". In: *Annual Conference of the International Speech Communication Association*, pp. 1045–1048, Makuhari, Japan, September 2010.

[52] KIM, Y., JERNITE, Y., SONTAG, D., et al. “Character-aware neural language models”. In: *AAAI Conference on Artificial Intelligence*, pp. 2741–2749, Phoenix, USA, February 2016.

[53] GRAVES, A. *Supervised Sequence Labelling with Recurrent Neural Networks*. Studies in Computational Intelligence. Heidelberg, Germany, Springer Verlag, 2012.

[54] SCHUSTER, M., PALIWAL, K. K. “Bidirectional recurrent neural networks”, *IEEE Transactions on Signal Processing*, v. 45, n. 11, pp. 2673–2681, November 1997.

[55] GRAVES, A., MOHAMED, A.-R., HINTON, G. E. “Speech recognition with deep recurrent neural networks”. In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 6645–6649, Vancouver, Canada, May 2013.

[56] GRAVES, A., JAITLY, N. “Towards end-to-end speech recognition with recurrent neural networks”. In: *International Conference on Machine Learning*, v. 32, pp. 1–9, Beijing, China, June 2014.

[57] MIAO, Y., GOWAYYED, M., METZE, F. “EESEN: end-to-end speech recognition using deep RNN models and WFST-based decoding”. In: *IEEE Workshop on Automatic Speech Recognition and Understanding*, pp. 167–174, Scottsdale, USA, December 2015.

[58] GRAVES, A., SCHMIDHUBER, J. “Offline handwriting recognition with multidimensional recurrent neural networks”. In: *Advances in Neural Information Processing Systems*, pp. 545–552, Whistler, Canada, December 2008.

[59] PASCANU, R., MIKOLOV, T., BENGIO, Y. “On the difficulty of training recurrent neural networks”. In: *International Conference on Machine Learning*, v. 28, pp. 1310–1318, Atlanta, USA, June 2013.

[60] HOCHREITER, S., SCHMIDHUBER, J. “Long short-term memory”, *Neural Computation*, v. 9, n. 8, pp. 1735–1780, November 1997.

[61] GERS, F. A., SCHMIDHUBER, J. “Recurrent nets that time and count”. In: *IEEE-INNS-ENNS International Joint Conference on Neural Networks*, v. 3, pp. 189–194, Como, Italy, July 2000.

[62] YAO, K., COHN, T., VYLOMOVA, K., et al. “Depth-gated LSTM”. August 2015. eprint arXiv:1508.03790v4.

[63] ZILLY, J. G., SRIVASTAVA, R. K., KOUTNÍK, J., et al. "Recurrent highway networks". March 2016. eprint arXiv:1607.03474v4.

[64] KOUTNÍK, J., GREFF, K., GOMEZ, F. J., et al. "A clockwork RNN". In: *International Conference on Machine Learning*, v. 32, p. 1–9, Beijing, China, June 2014.

[65] GRAVES, A., WAYNE, G., DANIHELKA, I. "Neural Turing machines". December 2014. eprint arXiv:1410.5401v2.

[66] SUKHBAATAR, S., SZLAM, A., WESTON, J., et al. "End-to-end memory networks". In: *Advances in Neural Information Processing Systems*, pp. 2440–2448, Montreal, Canada, December 2015.

[67] GREFF, K., SRIVASTAVA, R. K., KOUTNÍK, J., et al. "LSTM: A search space odyssey", *IEEE Transactions on Neural Networks and Learning Systems*, v. PP, n. 99, pp. 1–11, July 2015.

[68] JÓZEFOWICZ, R., ZAREMBA, W., SUTSKEVER, I. "An empirical exploration of recurrent network architectures". In: *International Conference on Machine Learning*, v. 37, p. 1–9, Lille, France, July 2015.

[69] BENGIO, Y., SIMARD, P. Y., FRASCONI, P. "Learning long-term dependencies with gradient descent is difficult", *IEEE Transactions on Neural Networks*, v. 5, n. 2, pp. 157–166, March 1994.

[70] ZAREMBA, W., SUTSKEVER, I., VINYALS, O. "Recurrent neural network regularization". February 2014. eprint arXiv:1409.2329v5.

[71] OGNAWALA, S., BAYER, J. "Regularizing recurrent networks: On injected noise and norm-based methods". 2014. eprint arXiv:1410.5684v1.

[72] MAAS, A. L., LE, Q. V., O'NEIL, T. M., et al. "Recurrent neural networks for noise reduction in robust ASR". In: *Annual Conference of the International Speech Communication Association*, pp. 22–25, Portland, USA, September 2012.

[73] NEELAKANTAN, A., VILNIS, L., LE, Q. V., et al. "Adding gradient noise improves learning for very deep networks". 2015. eprint arXiv:1511.06807v1.

[74] BAYER, J., OSENDORFER, C., CHEN, N., et al. "On fast dropout and its applicability to recurrent networks". 2013. eprint arXiv:1311.0701v7.

[75] GAL, Y., GHAHRAMANI, Z. "A theoretically grounded application of dropout in recurrent neural networks", pp. 1019–1027, December 2016.

[76] KRUEGER, D., MAHARAJ, T., KRAMÁR, J., et al. "Zoneout: Regularizing RNNs by randomly preserving hidden activations". January 2016. eprint arXiv:1606.01305v3.

[77] BA, L. J., KIROS, R., HINTON, G. E. "Layer normalization". 2016. eprint arXiv:1607.06450v1.

[78] WU, Y., ZHANG, S., ZHANG, Y., et al. "On multiplicative integration with recurrent neural networks", pp. 2856–2864, December 2016.

[79] LI, Y., QI, H., DAI, J., et al. "Fully convolutional instance-aware semantic segmentation". November 2016. eprint arXiv:1611.07709v1.

[80] XIONG, W., DROPPO, J., HUANG, X., et al. "Achieving human parity in conversational speech recognition". February 2016. eprint arXiv:1610.05256v2.

[81] DALAL, N., TRIGGS, B. "Histograms of oriented gradients for human detection". In: *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 886–893, San Diego, USA, June 2005.

[82] LEE, L., ROSE, R. C. "Speaker normalization using efficient frequency warping procedures". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*, v. 1, pp. 353–356, Atlanta, USA, May 1996.

[83] WELLING, L., KANTHAK, S., NEY, H. "Improved methods for vocal tract normalization". In: *IEEE International Conference on Acoustics, Speech, and Signal Processing*, v. 2, pp. 761–764, Phoenix, USA, March 1999.

[84] JURAFSKY, D., MARTIN, J. H. *Speech and language processing*. Prentice Hall Series in Artificial Intelligence. 2nd ed. New Jersey, USA, Prentice Hall, 2014.

[85] BAHL, L., BROWN, P., DE SOUZA, P., et al. "Maximum mutual information estimation of hidden Markov model parameters for speech recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, v. 11, pp. 49–52, Tokyo, Japan, April 1986.

[86] POVEY, D., KANEVSKY, D., KINGSBURY, B., et al. "Boosted MMI for model and feature-space discriminative training". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4057–4060, Las Vegas, USA, March 2008.

[87] JUANG, B.-H., HOU, W., LEE, C.-H. "Automatic recognition of spoken digits", *IEEE Transactions on Speech and Audio Processing*, v. 5, n. 3, pp. 257–265, May 1997.

[88] KAISER, J., HORVAT, B., KACIC, Z. "A novel loss function for the overall risk criterion based discriminative training of HMM models". In: *International Conference on Spoken Language Processing*, pp. 887–890, Beijing, China, October 2000.

[89] GIBSON, M., HAIN, T. "Hypothesis spaces for minimum Bayes risk training in large vocabulary speech recognition". In: *International Conference on Spoken Language Processing*, v. 6, pp. 2406–2409, Pittsburgh, USA, September 2006.

[90] MAAS, A. L., XIE, Z., JURAFSKY, D., et al. "Lexicon-free conversational speech recognition with neural networks". In: *Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT)*, pp. 345–354, Denver, USA, May 2015.

[91] AMODEI, D., ANUBHAI, R., BATTENBERG, E., et al. "Deep speech 2: end-to-end speech recognition in english and mandarin". In: *International Conference on Machine Learning*, v. 48, pp. 1–10, New York, USA, June 2016.

[92] "Google voice search: Faster and more accurate". `https://research.googleblog.com/2015/09/google-voice-search-faster-and-more.html`. Accessed: 2017-03-02.

[93] BAHDANAU, D., CHOROWSKI, J., SERDYUK, D., et al. "End-to-end attention-based large vocabulary speech recognition". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4945–4949, Shanghai, China, March 2016.

[94] BAHDANAU, D., CHO, K., BENGIO, Y. "Neural machine translation by jointly learning to align and translate". In: *International Conference on Learning Representations*, pp. 1–15, San Diego, USA, April 2014.

[95] XU, K., BA, J., KIROS, R., et al. "Show, attend and tell: Neural image caption generation with visual attention". In: *International Conference on Machine Learning*, v. 37, pp. 1–10, Lille, France, July 2015.

[96] GRAVES, A. "Generating sequences with recurrent neural networks". June 2013. eprint arXiv:1308.0850v5.

[97] PALAZ, D., MAGIMAI-DOSS, M., COLLOBERT, R. "Analysis of CNN-based speech recognition system using raw speech as input". In: *Annual Conference of the International Speech Communication Association*, pp. 11–15, Dresden, Germany, September 2013.

[98] GHAHREMANI, P., MANOHAR, V., POVEY, D., et al. "Acoustic modelling from the signal domain using CNNs". In: *Annual Conference of the International Speech Communication Association*, pp. 3434–3438, San Francisco, USA, September 2016.

[99] DAVIS, S., MERMELSTEIN, P. "Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, v. 28, n. 4, pp. 357–366, August 1980.

[100] JUANG, B.-H., RABINER, L. R., WILPON, J. G. "On the use of bandpass liftering in speech recognition", *IEEE Transactions on Acoustics, Speech, and Signal Processing*, v. 35, n. 7, pp. 947–954, July 1987.

[101] PALAZ, D., COLLOBERT, R., MAGIMAI-DOSS, M. "Estimating phoneme class conditional probabilities from raw speech signal using convolutional neural networks". In: *Annual Conference of the International Speech Communication Association*, pp. 1766–1770, Lyon, France, August 2013.

[102] HOSHEN, Y., WEISS, R. J., WILSON, K. W. "Speech acoustic modeling from raw multichannel waveforms". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 4624–4628, Brisbane, Australia, April 2015.

[103] ABDEL-HAMID, O., MOHAMED, A.-R., JIANG, H., et al. "Convolutional neural networks for speech recognition", *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, v. 22, n. 10, pp. 1533–1545, October 2014.

[104] CHAN, W., LANE, I. "Deep convolutional neural networks for acoustic modeling in low resource languages". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 2056–2060, Brisbane, Australia, April 2015.

[105] MIKOLOV, T., KOMBRINK, S., BURGET, L., et al. "Extensions of recurrent neural network language model". In: *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 5528–5531, Prague, Czech Republic, May 2011.

[106] JÓZEFOWICZ, R., VINYALS, O., SCHUSTER, M., et al. "Exploring the limits of language modeling". February 2016. eprint arXiv:1602.02410v2.

[107] GAROFOLO, J. S., LAMEL, L. F., FISHER, W. M., et al. "Timit acoustic-phonetic continuous speech corpus LDC93S1". Philadelphia, 1993. Linguistic Data Consortium.

[108] GRAVES, A. "Sequence transduction with recurrent neural networks". November 2012. eprint arXiv:1211.3711v1.

[109] JIM, K.-C., GILES, C. L., HORNE, B. G. "An analysis of noise in recurrent neural networks: Convergence and generalization", *IEEE Transactions on Neural Networks*, v. 7, n. 1–6, pp. 1424–1438, November 1996.

[110] GODFREY, J., HOLLIMAN, E. "Switchboard-1 release 2 LDC97S62". Philadelphia, 1993. Linguistic Data Consortium.

[111] NIST MULTIMODAL INFORMATION GROUP. "1997 hub5 english evaluation LDC2002S23". Philadelphia, 2002. Linguistic Data Consortium.

[112] GAROFOLO, J., GRAFF, D., PAUL, D., et al. "CSR-I (WSJ0) Sennheiser LDC93S6B". Philadelphia, 1993. Linguistic Data Consortium.

[113] "CSR-II (WSJ1) Sennheiser LDC94S13B". Philadelphia, 1994. Linguistic Data Consortium.

[114] COATES, A., HUVAL, B., WANG, T., et al. "Deep learning with cots HPC systems". In: *International Conference on Machine Learning*, v. 30, pp. 1–9, Atlanta, USA, June 2013.

[115] HANNUN, A. Y., CASE, C., CASPER, J., et al. "Deep speech: Scaling up end-to-end speech recognition". December 2014. eprint arXiv:1412.5567v2.

[116] CIERI, C., MILLER, D., WALKER, K. "The fisher corpus: A resource for the next generations of speech-to-text". In: *International Conference on Language Resources and Evaluation*, pp. 69–71, Lisbon, Portugal, May 2004.

[117] SAXE, A. M., MCCLELLAND, J. L., GANGULI, S. "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks". In: *International Conference on Learning Representations*, pp. 1–22, San Diego, USA, April 2014.

[118] LEE, K.-F., HON, H.-W. “Speaker-independent phone recognition using hidden markov models”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, v. 37, n. 11, pp. 1641–1648, November 1989.

[119] SCHRAMM, M., FREITAS, L. F., ZANUZ, A., et al. “CSLU: Spoltech Brazilian Portuguese version 1.0 LDC2006S16”. Philadelphia, 2006. Linguistic Data Consortium.

[120] NETO, N., SILVA, P., KLAUTAU, A., et al. “Spoltech and OGI-22 baseline systems for speech recognition in Brazilian Portuguese”. In: *International Conference on Computational Processing of Portuguese Language*, v. 5190, pp. 256–259, Aveiro, Portugal, September 2008.

[121] “Voxforge”. `https://http://www.voxforge.org`. Accessed: 2017-03-06.

[122] “Falabrasil - UFPA”. `http://www.laps.ufpa.br/falabrasil/`. Accessed: 2017-03-06.

[123] COLLOBERT, R., KAVUKCUOGLU, K., FARABET, C. “Torch7: A matlab-like environment for machine learning”. 2015.

[124] AL-RFOU, R., ALAIN, G., ALMAHAIRI, A., et al. “Theano: A Python framework for fast computation of mathematical expressions”. May 2016. eprint arXiv:1605.02688v1.

[125] GOODFELLOW, I. J., WARDE-FARLEY, D., LAMBLIN, P., et al. “PyLearn2: A machine learning research library”. August 2016. eprint arXiv:1308.4214v1.

[126] JIA, Y., SHELHAMER, E., DONAHUE, J., et al. “Caffe: Convolutional architecture for fast feature embedding”. June 2014. eprint arXiv:1408.5093v1.

[127] AGARWAL, A., AKCHURIN, E., BASOGLU, C., et al. *An introduction to computational networks and the computational network toolkit*. Technical Report 112, Microsoft, 2014.

[128] ABADI, M., AGARWAL, A., BARHAM, P., et al. “Tensorflow: large-scale machine learning on heterogeneous systems”. 2015. Disponível em: <`http://tensorflow.org/`>. Software available from tensorflow.org.

[129] CHOLLET, F. “Keras”. `https://github.com/fchollet/keras`, 2015.

[130] “Warp-CTC”. `https://github.com/baidu-research/warp-ctc`. Accessed: 2017-03-06.