Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Thi Huyen Cao

German Word Level Lip Reading with Deep Learning

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Thi Huyen Cao

German Word Level Lip Reading with Deep Learning

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. -Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 04. Juni 2019

Thi Huyen Cao

Thema der Arbeit
Deutsches Lippenlesen auf Wortebene mit Deep Learning

Stichworte
Deep Learning, Lippenlesen, C3D, LSTM, LRCN

Kurzzusammenfassung
Lippenlesen, auch visuelle Spracherkennung genannt, ist eine der herausfordendesten Aufgabe in der Kreuzung zwischen Computervision und Verarbeitung der natürlichen Sprachen. Fast alle Arbeiten in diesem Bereich konzentrieren sich auf die englische Sprache. Diese Abschlussarbeit untersucht, implementiert und evaluiert verschiedene neuronale Netze für die deutsche Sprache. Dafür wird der Datensatz DLIP erschaffen.

Thi Huyen Cao

Title of the paper
German Word Level Lip Reading with Deep Learning

Keywords
Deep Learning, Lip Reading, C3D, LSTM, LRCN

Abstract
Lip Reading, also known as Visual Speech Recognition is a challenging task in the intersection between Computer Vision and Natural Language Processing. Most works in this field focus on the English language. This thesis analyses, implements and evaluates different neural networks on the German language. DLIP is created and served as the dataset.

Contents

List of Figures

List of Tables

Acronyms

AI Artificial Intelligence. 1, 2, 5

BPTT Backpropagation Through Time. 19, 20

CNN Convolutional Neural Network. 16, 17, 18, 21, 28

CV Computer Vision. 6, 7

DL Deep Learning. 1, 2, 3, 5, 7, 13, 24, 28, 31, 33, 59, 60

LR Lip Reading. 1, 3, 4, 5, 34, 35, 59, 60

LRCN Long-term Recurrent Convolutional Network. 21, 34, 59

LSTM Long Short Term Memory. 20, 21, 34

ML Machine Learning. 1, 2, 5, 8, 24, 31

MLP Multilayer Perceptron. iv, vi, 13, 18

NLP Natural Language Processing. 28, 60

NN Neural Network. 2, 6, 13, 14, 16, 17, 18, 21, 27, 31

RNN Recurrent Neural Network. 18, 19, 20, 21

TBPTT Truncated Backpropagation Through Time. 20

Chapter 1

Introduction

This chapter aims to give a brief overview of the history of Artificial Intelligence (AI), the state of art of Deep Learning (DL), Lip Reading (LR) as well as the author's motivation of working on this topic. Furthermore, the scope of the thesis and its aims are defined concretely.

1.1 Artificial Intelligence and Deep Learning

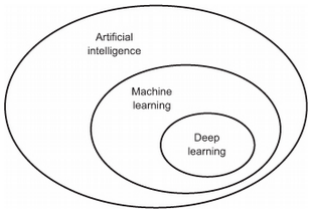


Figure 1.1: Relationship among Artificial Intelligence, Machine Learning and Deep Learning [15]

AI is a part of computer science, which has grown exponentially and gained a lot of attention from researcher, developer and company in the last several decades. It was born in 1956 with the target of creating a non-human intelligence to think, act and perform tasks like a human or even better. Among different approaches to this field, four common ones (think humanly, act humanly, think rationally, act rationally) are described detailed in [46]. In the early days, it focused on creating programs with significant explicit rules to manipulate knowledge (symbolic AI), which got remarkable success in playing chess, solving logic tasks etc. This approach hit its limit in the 1980s when it comes to complex tasks with less internal logic such as image recognition, voice recognition and so on. It is soon replaced by a new approach, known as Machine Learning (ML).

Unlike classical programming, ML "teaches" computer the knowledge (muster, pattern etc.) by giving it samples.

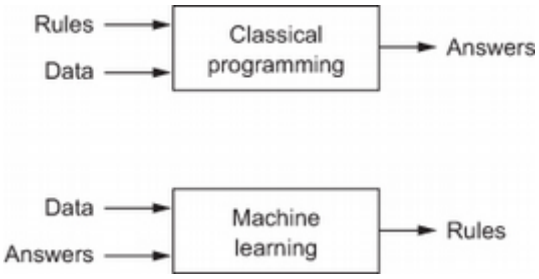


Figure 1.2: Comparison between Machine Learning and classical programming [15]

Three common categories of ML are: supervised learning, unsupervised learning and reinforcement learning. DL belongs to supervised learning. The base knowledge of DL was set early in the 1980s. Since AI experienced "its winter" in a couple of years from 1998 because of difficulties with the implementing of algorithms, finding data, as well as efficient algorithms, DL actually achieved huge success in many challenging tasks in the last over one decade including image recognition, speech recognition, action recognition, text translation, sequence classification, sequence generation, cancer detection, face detection, along with others. Three main reasons for this significant change are pointed out by François Chollet [16]

- *Hardware*: The fast development of the chip industry, especially for ML/DL applications enables the implementing of algorithms and increases the speed of the training process. The last decade is also labeled as "the era of GPU (graphical processing units)". Companies like NVIDIA and AMD continue improving the performance of GPU. Another trend is developing FPGAs [1] and ASICs [2]. A well-known example is Google TPU (Tensor processing unit), which was released in Google IO 2016.

- *Datasets and benchmarks*: The rise of the internet allows collecting data easier than ever. Google releases Google Datasets for searching for data. Kaggle provides a lot of hands-on datasets for competitions and does a very good job on motivating researchers and engineers to publish their data as a contribution to the deep learning community. A lot of ground truth tools (LabelMe, VitBat, VATIC etc.) and platforms (Amazon Mechanical Turk) help create labels for data fastly and efficiently.

- *Algorithmic advances*: A deeper understanding of gradient propagation, along with important algorithmic improvements on activation functions, weights initialization and optimizers around 2009 allows training in deeper neural networks. In the last few years, many proposed methods such as batch normalization, dropout, regularization etc. have proven their huge effects on fighting against underfitting and overfitting, which are two biggest problems of DL. Neural Network (NN) with complex architecture and high depth is consequently able to be trained.

[1] Field-Programmable Gate Arrays, an integrated circuit designed to be programmed by developer, a detailed comparison between GPU and FPGA can be found [https://www.aldec.com/en/company/blog/167–fpgas-vs-gpus-for-machine-learning-applications-which-one-is-better]

[2] Application-specific Integrated Circuits, produced to be customized for special calculations rather than general-purpose computing

Besides, the development of frameworks also plays an essential role in enlarging the DL community, not only making it easier for beginner to access and understand DL but also simpler for developers when working on complex tasks.

Deeper and more complex networks are showed to be able to solve challenges with very high complexity. Having high confidence about the scope and potential of DL nowadays, the author deeply believes that DL is the right technology to be applied for Lip Reading (LR).

1.2 Lip Reading

LR, also known as visual speech recognition is about understanding speech by interpreting only visual signal including lip, face, mouth and tongue movements whereas lips contain most information. LR is a challenging task and not fully solved yet. Human performs this task at average very bad. [8] shows that it is affected by different factors such as context, environment, relationship between communicators, personal knowledge, intelligence, training and so on. The closer communicators know each other, the better they can do LR. As a result, even the reliability of a professional human lip-reader is debatable. Nevertheless, LR experts or LR training courses are in general very expensive. Therefore, it is important to create automatic LR (machine LR). [6] emphasizes the difficulties of machine LR because it requires extracting spatiotemporal features from the video (since both position and motion are important).

LR would make a huge difference for hard-of-hearing and deaf people by helping them understand the dialog daily, be more confident in their communication abilities, open an opportunity to access the resources which are not supported for them such as news of any kind without subtitle. Also, mute people could profit from LR. LR facilitates these targeted consumers to better understand the content and decipher conversations. The author believes that LR has a high potential to handle the main job of sign languages in the future. Other applications of LR are:

- Communication in a noisy environment such as in a bar (bartender-customers), disco, conference etc. where it is hard or unable to understand acoustically.
- Automatic subtitle for video as an alternative to audio recognition, as well as a contribution to audio-visual speech recognition AVSR.
- Improve the quality of video with poor sound quality such as video conference with bad internet connection.
- Investigation: In many cases of police investigation, it is needed to reconstruct the communication of suspected people while traffic and surveillance cameras are used to having no sound included.
- Creating sound/subtitle for silent videos of historical documentation, theater plays, communication between players in football matches and so on.

Understanding the benefit as well as the challenge of LR, many researchers have contributed to improve the performance of automatic LR both word and sentence-level in the last few years. Table 1.1 gives a summary of existing datasets and their best-recorded performances [17] [6].

Name	**Env.**	**Output**	**I/C**	**#class**	**#subject**	**Best perf.**
AVICAR	In-car	Digits	C	10	100	37.9%
AVLetter	Lab	Alphabet	I	26	10	64.6%
CUAVE	Lab	Digits	I	10	36	83.0 %
GRID	Lab	Words	C	8.5*	34	86.4%
GRID	Lab	**Sentences**	C	-	34	**95.2%** [6]
OuluVS1	Lab	Phrases	I	10	20	91.4%
OuluVS2	Lab	Phrases	I	10	52	94.1%
OuluVS2	Lab	Digits	C	10	52	92.8 %
LRW	TV	Words	C	333/500	1000+	65.4% / 61.1%
MIRACL	Lab	Words	I	15	10	59 % [25]
URDU	Lab	Words	I	10	10	62 % [21]
URDU	Lab	Digits	I	10	10	72 % [21]
LRS2 [1]	TV	Sentences	C	-	-	- **
LRS3 [1]	TED/TEDx	Sentences	C	-	-	- **

Table 1.1: Table of existing datasets. **I** for **I**solated (one word, letter or digit per recording); **C** for **C**ontinuous recording. The reported performance is on speaker-independent experiments. (* For GRID, there are 51 classes in total, but the first word in a phrase is restricted to 4, the second word 4, etc. 8.5 is the average number of possible classes at each position in the phrase. ** For LRS2 and LRS3, different statistics about utterances, sentence length and word instances are given. Their performance is meassured based of Word Error Rate WER. More detail in [1])

1.3 Scope of the thesis

Unfortunately, almost datasets and models are built for the English language. To the best of author's knowledge, there is no existed work on the German language. The German language contains 30 phonemes whereas only 11 kinetically (in terms of movements) distinguishable. [34] [33] gives some examples of words and characters which have a high similarity of lip movements, which makes German LR very challenging.

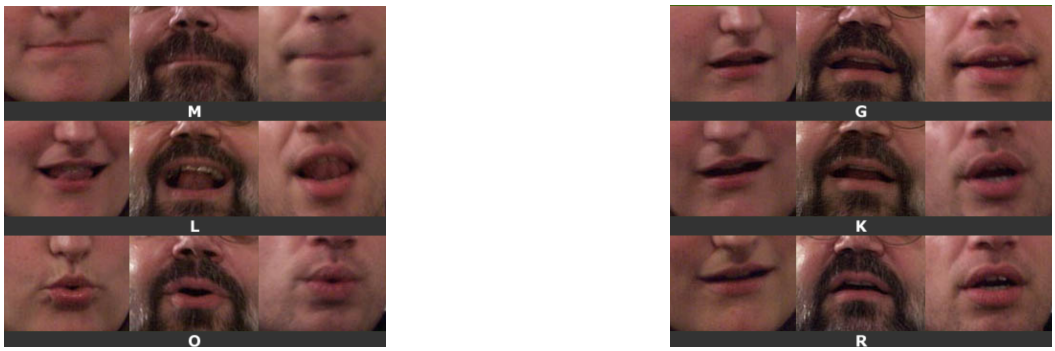
[p]	[b]
[k]	[g]
[t]	[d]
[f], [v]	[ph]
[v]	[w]
[m]	[n]

(a) phonemes couples

Reifen	Greifen
Freunde	Freude
backen	packen
Juni	Juli
Kampf	Krampf
gejagt	gesagt
Dreißig	fleißig, weiß ich, weiß nicht
Staat	Stadt, statt
Beet	Bett
Achtzig	hat sich, macht sich

(b) examples

Table 1.2: Similar couple of phonemes and examples



(a) easy to read lip movements (b) difficult to read lip movement

Figure 1.3: Example of lip movements

Therefore, the author is desired about giving a work in this thesis to find out how robust is German machine LR. Since LR is a very complex task because of the variety of object, speaking speed, speaking habits, mouth shape, face shape, the combination of words in a sentence, along with others, it is limited in this thesis at world-level LR. It means word classification among a number of words spoken one by one. As there is no existed dataset, DLIP is created and serves as the base resource for training and validation (detail in chapter 5).

1.4 Objectives and structure

The objectives of this thesis are:

- To give a summary of the state of art of DL and LR.
- To find out the accuracy of German Word Level LR with DL.
- To find out the performance of different model architectures for LR.
- Giving detail error analysis to find out which words happen to be recognized easily and which laboriously.
- Doing research to find out the possibility of sentence-level LR and real-time LR system.

The thesis is divided into 6 chapters. Chapter 1 is aimed to give an introduction into the topics AI, DL, LR and defines the scope of the thesis and its objectives. In chapter 2 general concepts used for LR will be explained including detection, tracking, different neural network architectures, hyperparameter tuning techniques for overfitting and underfitting, different metrics for validation and error analysis. Followed by chapter 3 are the tools and chapter 4 is the general workflow of ML/DL projects as well as LR task which is used for the implementation in chapter 5. Chapter 5 will present step by step how the data is prepared, processed and validated. Lastly, the result of different models will be taken into comparison. A confusion matrix and an error analysis table will be provided to give a detailed view of model accuracy. The final chapter gives a summary of all the work done, future work and the author's opinion on the further developing of LR.

Chapter 2

Concepts

This chapter explains the general concepts which will be used later for the implementation. It includes the introduction of different NN architectures and their applied fields, the training techniques and their effect and how to evaluate the model.

2.1 Face Detection and Tracking

2.1.1 Image Classification, Localization and Detection

Image Classification is a common task in Computer Vision (CV). According to [40] it refers to the task of assigning an input image to one of the defined output class. Simply it answers the question whether the output class appears in the input image. There is normally one object in the input image.

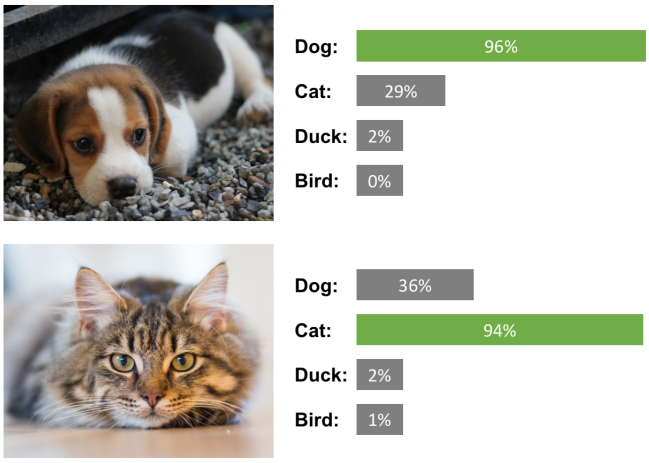


Figure 2.1: Example of Image Classification. [47]

Object Localization is the process of localizing the position of the object in an image. Normally a bounding box around the object is marked as its location. **Object Detection** wraps Image Classification and Object Localization together, provides both information about the object appearance, its label and its location if exists. Another precise way to locate the object is called **Image Segmentation**, where it partitions an image into multiple segments. Every pixel is labeled. The group of pixels with same label normally characterizes a specific object. Detection and Segmentation can be applied to image with multiple objects. Figure 2.2 visualizes the differences concretely.

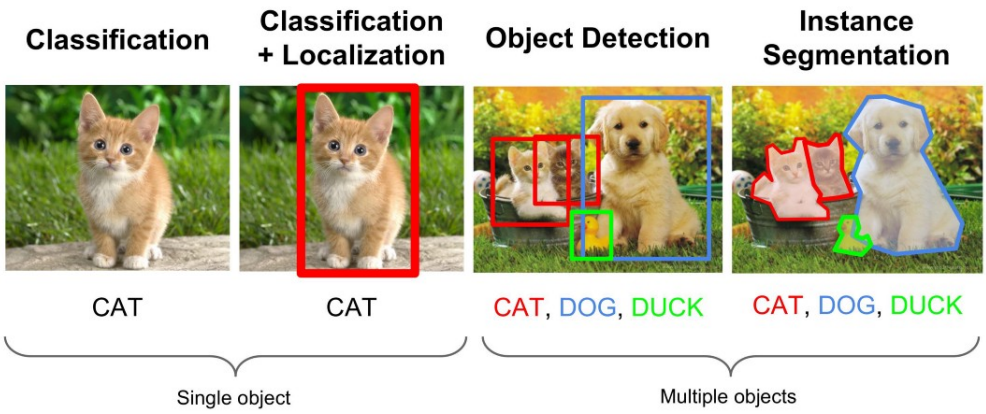


Figure 2.2: Comparison among Image Classification, Object Detection and Instance Segmentation [42]

While Object Detection seems like an easy task for human, it does not to machine. Figuring out the relationship among pixels is very challenging. Object Detection was and still is a fundamental problem of CV. During the years, a lot of datasets, competitions have been created such as ImageNet, COCO, PASCAL VOC etc. Many algorithms have been proposed and have proven their effectiveness such as Fast R-CNN, Faster R-CNN, YOLO and so on. A detailed comparison among algorithms and performance review on different datasets, as well as validation methods such as Intersection over Union IoU, mean Average Precision mAP is presented in [42] [11] [47].

Among different Object Detection, Face Detection appears to be one of the field with high potential application in real life and serves as the first essential step for Face Recognition. Dr. Robert Frischholz believes that when faces can be located exactly in any scene, the recognition step afterwards is not so complicated anymore [22]. Well-known examples of Face Recognition are Facebook Face Recognition for images developed by DeepFace team, FaceID by Apple, automatic Face Recognition for employees in Baidu's headquarter in China. The applied field in those examples can be easily categorized as entertainment, convenience and security. According to [22], the following approaches have been used for Face Detection:

1. Using typical skin color to find face segments. However, it is difficult to cover all kind of skin and control external effect such as brightness, contrast etc.

2. Finding faces by motion as the fact is that face is always moving. The problem happens when there are other moving objects around. [1]

3. Model-based Face Detection: using Edge-Orientation Matching [2], Hausdorff Distance [3]

4. Using Cascade of "weak classifiers"

5. Histogram of Oriented Gradients (HOGs) and DL

[1]These 2 first approaches are very poor when it comes to unconstrained scenes

[2]Fröba, Küblbeck: Audio- and Video-Based Biometric Person Authentication, 3rd International Conference, AVBPA 2001, Halmstad, Sweden, June 2001. Proceedings, Springer. ISBN 3-540-42216-1.

[3]Jesorsky, Kirchberg, Frischholz: Audio- and Video-Based Biometric Person Authentication, 3rd International Conference, AVBPA 2001, Halmstad, Sweden, June 2001. Proceedings, Springer. ISBN 3-540-42216-1.

Among these, 4 is the most commonly used nowadays, also known as Haar Feature-based Cascade Classifiers or Haar Cascade, which was introduced by Paul Viola and Michael Jones in 2001 [52] [51]

2.1.2 Face Detection with Haar Cascade

Haar Cascade is a ML object detection algorithm in images and videos, which is mainly used for detecting face and body parts. The algorithm is trained with a lot of positive and negative images, for example images with and without a face. The training consists of 3 stages

1. Extracting Features
2. Adaboost Training
3. Cascading Classifiers

1. Extracting Features
In comparison to pixel-based approach, [51] emphasizes that the features can act to encode ad-hoc domain knowledge, which is difficult to learn using a finite quantity of training data. In addition, feature-based systems operate much faster than pixel-based systems. Therefore, haar features are collected and calculated.

A haar feature considers adjacent rectangular regions at a specific location in a detection windows, summarize the pixel intensities in each region and calculate the differences between these sums. Concretely in Figure 2.3 Value = $\sum$(pixels within white rectangles) - $\sum$(pixels within black rectangles)

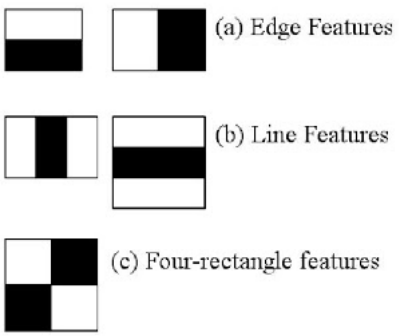


Figure 2.3: Haar features [41]

Haar features are then calculated through each image, with all possible size and location. For instance, an image of resolution 24x24 can result over 180 000 features. Addressing this exploding computing problem, [51] proposed **integral image**, which speeds up the calculation remarkably. The **integral image** at location x,y contains the sum of all pixels above and to the left of x,y:

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y')$$
$$s(x,y) = s(x, y-1) + i(x,y)$$
$$ii(x,y) = ii(x-1, y) + s(x,y)$$

where s(x,y) is the cumulative row sum, s(x, -1) = 0 and ii(x,y) is the integral image at location x,y, ii(-1,y) = 0

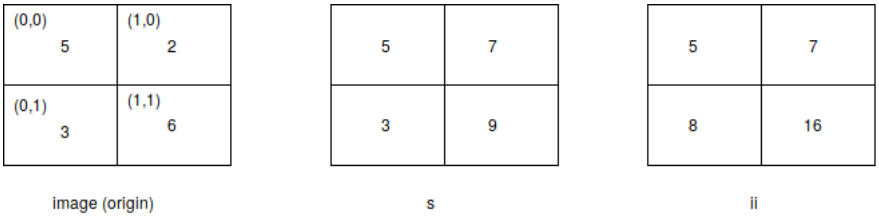
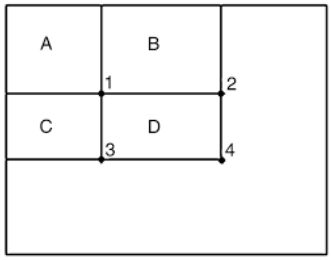
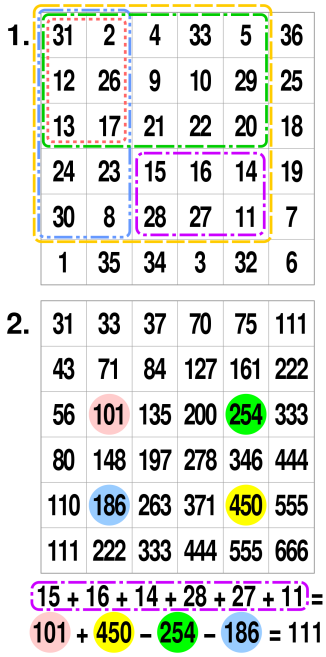


Figure 2.4: Example of integral image



(a) The sum of the pixels within rectangle D can be computed with four array references. The value of integral image at location 1 is the sum of pixels in rectangle A. The value at location 2 is A+B, at location 3 is A+C and at location 4 is A+B+C+D. The sum within D can be computed as 4+1-(2+3)



(b) 1.original image, 2. integral image [49]

Figure 2.5: Example of calculating haar features

The sum of pixels in some rectangle which is a subset of the original image can be done within a constant time, means complexity O(1) as shown in Figure 2.5

2. Adaboost Training

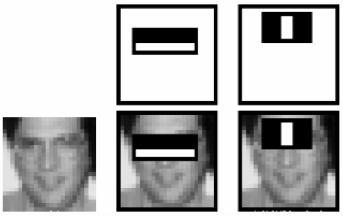


Figure 2.6: Relevant features for Face Detection [41]

Figure 2.6 shows that not all features are relevant to detecting face. The first feature for example focuses on eyes region and the second on nose region. Applying those features

in other regions such as cheek does not really give any result. Adaboost is then used to figure out the most relevant features. This small set of features will be able to detect a face efficiently. Adaboost is an efficient boosting algorithm which combines "weak classifiers" [4] to a "strong classifier".

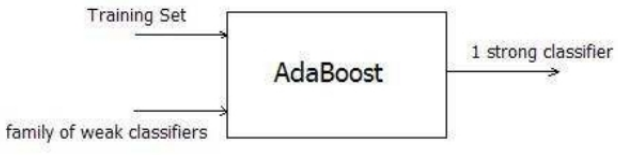


Figure 2.7: General Scheme of Adaboost [39]

A "weak classifier" in this case can be considered as a feature evaluation, followed with an optimal threshold so that the number of misclassified samples using only this feature is minimum [22]. The process is as follows: each data sample is assigned a weight. A number of iterations are executed until the required accuracy is arrived or the number of relevant features are found. In each iteration, a "weak classifier" (feature) is chosen with respect to the distribution of data with current weights, which provides the best separation between positive and negative images. The weights of data are then updated. The weights of misclassified samples are increased. The final classifier is then constructed as a linear combination of all chosen "weak classifier". In other words, it is a weighted sum of them, where the weight is given respectively to the accuracy of each "weak classifier" alone. The better the "weak classifier" alone performs, the more information it will contribute to the final classifier and as a result is assigned a bigger weight.

$$F(x) = \alpha_1 f_1(x) + \alpha_2 f_2(x) + \alpha_3 f_3(x) + ...$$

A more detailed view on how to find optimal threshold as well as how to choose the weak clasifier in each iteration can be found in [39] [51]. [51] states that a classifier formed by 200 features can already achieve an accuracy of 95%.

3. Cascading Classifiers

Another issue when it comes to Face Detection is that a face is usually just a small part of an image. Applying an even small set of features on every part of an image is time-wasting. Viola and Jones also introduced the concept of **Cascade of Classifier** in their paper [51] to increase the detection performance as well as reduce rapidly the computation. The principle is to reject the non-face regions as soon as possible. **Cascade of Classifier** consists of multiple stages. The early stages use a simple classifier, which is built from a very less number of features. Processing through those stages costs very less computing power and could effectively remove a lot of negative sub-windows. Those stages have a very high accuracy rate, roughly 100 % but also produces a high rate of false positives [5], about 40% [39], which is solved in the later stages. Only sub-windows, which pass the early stages can be processed to the next stage. The later stages use a more complex classifier, with more features and as a result provides a smaller rate of false positive while consuming more computing. The sub-window which passes all stages contains a face. Figure 2.8 visualizes the process schematically.

[4] Calling "weak" because this kind of classifier does not need to get a high accuracy in general. It only requires to be better than guessing, means more than 50%

[5] a negative sample is mistakenly classified as positive

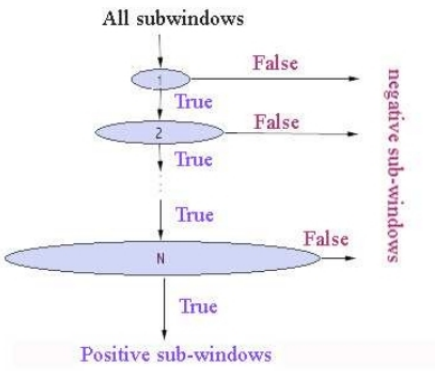


Figure 2.8: Schematic description of Cascade of Classifier [39]

According to [51], their final cascade contains 38 stages with over 6000 features. The number of features in the first stages are 1, 10, 25, 25, 50. Haar Cascade is embedded directly in OpenCV and can be used easily in some lines of code.

```
# Creating face detector with pre-trained Haar Cascade
face_detector = cv2.CascadeClassifier
        ('haarcascade_frontalface_default.xml')
# Detecting a list of face
faces = face_detector.detectMultiScale(
        image,
        scaleFactor=1.1,
        minNeighbors=10,
        minSize=(30,30),
        flags = cv2.CASCADE_SCALE_IMAGE)
```

Haar Cascade could also work well with a sequence of frames, where the face is detected in each frame separately. Thus, in many cases the face shape (eyes, mouth etc.) or orientation tends to change over time, makes it very hard for detection to work consistently. In those cases the information through time is very important. The position of faces in the last frames plays an essential role in localizing them in the current frame. Tracking exactly takes advantage of that information.

2.1.3 Tracking

In comparison to Detection, [36] shows that Tracking works much more efficient on videos.

1. Tracking is faster than Detection. By carrying the information in the last frame, Tracker is able to search for the face in the surrounding of the last position, reducing a lot of computing while Detection always starts from scratch.

2. Tracking can help when Detection fails. As mentioned above, Detection likely fails when the eyes, mouth shape change or a part of face is covered. A good Tracker, on the other hand, can handle some level of occlusion.

3. Tracking preserves identity. Unlike Detection, Tracking does not only output the appearance of a face in a frame, but also its identity, which is held during the whole videos.

There are different tracking algorithms, each has advantages and disadvantages. Table 2.1 shows some of them [36] [43].

Description	Pros	Cons
BOOSTING is based on AdaBoost. This classifier needs to be trained at runtime by positive and negative examples. The init bounding box serves as the first positive example. Processing a new frame, the classifier searches in the neighborhood of last location. The sub-window with maximum score is the new location, the old location is added as another positive example	None (a decade old)	not reliable failure report
MIL Multiple Instance Learning [7] is similar to BOOSTING, except it generates a bag of positive examples from the neighbor of the current location instead of picking only the current location. Multiple Instance Learning MIL is then used to choose the new location.	better than BOOSTING	not reliable failure report, handle occlusion poorly.
KCF Kernelized Correlation Filters are based on the idea of BOOSTING and MIL. It uses the fact that positive samples in the bag have a large overlapping regions.	better, faster, and reports failure better than BOOSTING and MIL	handle occlusion poorly.
TLD Tracking, learning and detection contains 3 components: a tracker, a detector and a learner. From the author's paper, " The tracker follows the object from frame to frame. The detector localizes all appearances that have been observed so far and corrects the tracker if necessary. The learning estimates detector's errors and updates it to avoid these errors in the future."	works well under occlusion, over scale changes	high rate of false positive
MedianFlow tracks in both forward and backward direction in time, measures the discrepancies between these two trajectories	excellent failure report, works well when motion is predictable and no occlusion	fails under large of jump in motion
GOTURN is based on Convolutional Neural Network CNN	robust to changes of viewpoint, light and deformations	handle occlusion poorly
MOSSE Minimum Ouput Sum of Squared Error [10] uses adaptive correlation filter, which produces stable correlation filters when initialized using a single frame	robust to variations in lighting, scale, pose and non-rigid deformations, occlusion and works at higher frame rate 669 fps	-
CSRT: Discriminative Correlation Filter (with Channel and Spatial Reliability) [35]	high accuracy	operates at lower frame rate 25fps

Table 2.1: Different Tracking algorithms

2.2 Artificial Neural Networks

The "Deep" in DL refers to the depth of the learning models, the artificial neural networks NN. It is believed that NN is inspired by biological neural system, which controls how the brain works. NN contains different layers, which hold some information, that help map the input to the output. These layers are also called data representation. Each is built from a number of neurons, which help transfer the information through the layers. Whether the neurons should be activated to the next layer depends on a system of parameters. The learning process is described detailed in Figure 2.9

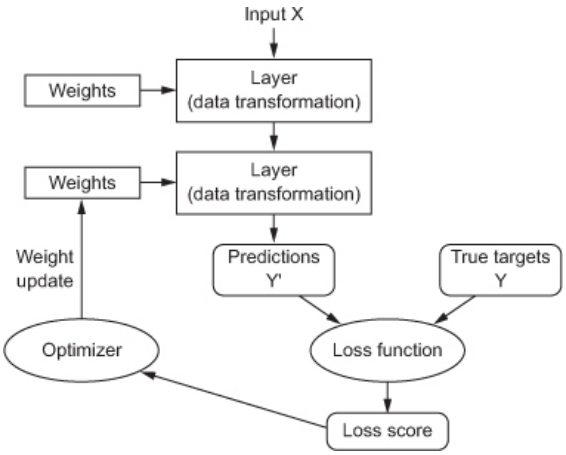


Figure 2.9: Learning process of neural networks [16]

The aim here is to find the set of parameters (weights and bias for each layers, configuration for cost function, optimizer function etc.), which performs the best mapping between input and output. This process is iterative and can shortly summarized in 3 main parts:

1. Training forward (Forwardpropagation)
2. Calculating costs
3. Training backward (Backpropagation)

For the sake of clarity, those parts are explained along with one of the very first NN architecture: Feedforward NN. Calling feedforward because this architecture does not contain any backward connection between layers, the data flows simply only in the forward direction.

2.2.1 MLP

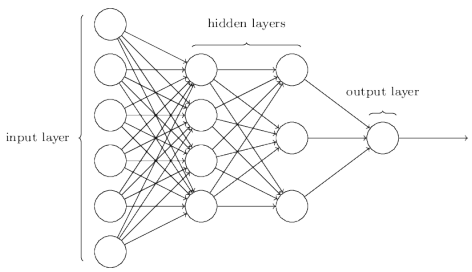


Figure 2.10: A 3-layer MLP

Multilayer Perceptron (MLP) is a class of Feedforward NN, which contains at least an input layer, a hidden layer and an output layer. Each layer is fully connected with the next layer. See Figure 2.10

1. Training forward
Each neuron consists of weights and bias. $W^{[l]}$ is then the weight matrix to connect all neurons in layer l-1 to layer l e.g. $W^{[1]}$ is the weight matrix for connection between input layer and layer 1. The value of next layer is calculated as follow:

$$Z^{[l]} = W^{[l]} * A^{[l-1]} + b^{[l]}$$
$$A^{[l]} = a^{[l]}(Z^{[l]})$$

with $A^{[l-1]}$ is the activation function value of layer l-1 and $a^{[l]}$ is the activation function used in layer l. In general, the weights of each neuron is initialized randomly to break the symmetry. Deep NN used to suffer from exploding and vanishing gradients problem, which makes it very hard to train. [24] issued the difficulty and proposed a different way to initialize weights called Xavier or Glorot Weight Initialization, which works better with very deep NN using sigmoid activation function. [26] also carried out another method, as known as He Weight Initialization, which helps with convergence of very deep models, which use activation functions ReLU/PReLU while Xavier Initialization doesn't seem to help. Both initializes weights with a random values, depending on the number of neurons in the last layer n by calibrating the variance [6] of weights var(w) = $\frac{1.}{n}$ (xavier) and var(w) = $\frac{2.}{n}$ (he) and bias to 0. An simple implementation in python is as followed:

```
# W_l: weights of layer l, n is the number of neurons in layer l-1
# shape = (number of neurons in layer l, number of neuron in layer l-1)
W_l = np.random.randn(shape)* np.sqrt (1. / n) # (xavier)
W_l = np.random.randn(shape)* np.sqrt (2. / n) # (he)
```

Besides ReLU and sigmoid, there are many other activation functions such as leaky ReLU, softmax, tanh etc. where softmax is mostly used as output activation function for multiple class classification. An activation function has in general 3 requirements.

1. it should be non-linear: Using linear activation function makes the flows of data through all layers just a linear transformation. Normally linear transformation is not able to solve tasks with high complexity.

2. it should be continuously differentiable. The backward training uses backpropagation which requires the derivative of those functions to know in which direction the parameters should be updated.

3. it should have a limited number range: Fixing the layer output in a range makes the training process more stable.

2. Calculating costs
Once the step forward is done, a prediction is made at the output layer. It is then compared to the correct label. The difference between the training result and ground truth is calculated by a loss function, which serves as a measure of how incorrect the model still is. The choice of loss functions nevertheless depends on different factors such as

[6] variance σ^2 measures the spread or variability of a distribution of (random) variables

presence of outliers, data distribution, time for computing gradient and so on. For instance L1 Loss or Mean Absolute Error MAE=$\frac{1}{m}\sum|\hat{y}-y|$ (with m: number of samples, $\hat{y}$: output prediction, y: label) is more robust to outliers than L2 Loss or Mean Square Error MSE=$\frac{1}{m}\sum(\hat{y}-y)^2$ because in case of outlier the difference between prediction and label is quite large and square of it is even bigger. If outliers represent anomaly that need to be detected, then MSE is preferred since it is more sensitive to outliers. Another more commonly used loss function is Cross Entropy Loss L $= -\frac{1}{m}\sum_{c=1}^{C} y_{ci}.log(\hat{y}_{ci})$ for C classes classification, m samples in total. In case of 2 classes, binary cross entropy can be calculated as L $= -\sum_{c=1}^{2} y_{ci}.log(\hat{y}_{ci}) = -\frac{1}{m}\sum_{i=1}^{m}(y_1.log(\hat{y}_1) + y_2.log(\hat{y}_2)) = -\frac{1}{m}\sum_{i=1}^{m}(y.log(\hat{y}) + (1-y).log(1-\hat{y}))$ since $y_1 + y_2 = 1$

3. Training backward

In other words, the task is to update parameters in a direction where the loss decreases. Finding the minimum of loss function can be done by solving the equation L'(parameters) = 0. Since it is unrealistic in practice because of a big set of parameters, it is done by calculating the gradient of loss function with regard to model's parameters, which means calculating the partial derivation of loss function with respect to each parameter. Then the parameters will be moved a little toward the minimum by a learning rate μ, which determines the size of the step to take, either after processing the whole data (batch gradient descent), a data unit (stochastic gradient descent) or a mini-batch of data (mini-batch gradient descent).

$$\theta := \theta - \mu * \frac{\partial L}{\partial \theta}$$

For example: updating weight matrix $W^{[3]}$:l2 $\rightarrow$ l3(output layer) and $W^{[2]}$: l1 $\rightarrow$ l2 of a model with 3 layers.

$$\begin{aligned}
dW^{[3]} &= \frac{\partial \text{Ł}}{\partial W^{[3]}} \\
&= \frac{\partial \text{Ł}}{\partial \hat{y}}.\frac{\partial \hat{y}}{\partial Z^{[3]}}.\frac{\partial Z^{[3]}}{\partial W^{[3]}}(\text{chain rule}) \\
&= \frac{\partial \text{Ł}}{\partial \hat{y}}.a'^{[3]}(Z^{[3]}).A^{[2]} \\
dW^{[2]} &= \frac{\partial \text{Ł}}{\partial W^{[2]}} \\
&= \frac{\partial \text{Ł}}{\partial \hat{y}}.\frac{\partial \hat{y}}{\partial Z^{[3]}}.\frac{\partial Z^{[3]}}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial Z^{[2]}}.\frac{\partial Z^{[2]}}{\partial W^{[2]}} \\
&= \frac{\partial \text{Ł}}{\partial \hat{y}}.a'^{[3]}(Z^{[3]}).W^{[3]}.a'^{[2]}(Z^{[2]}).A^{[1]} \\
W^{[3]} &= W^{[3]} - \mu.dW^{[3]} \\
W^{[2]} &= W^{[2]} - \mu.dW^{[2]}
\end{aligned}$$

A good choice of learning rate is very important. While a too high learning rate could lead to missing minimum, a too low could cause very slow convergence and in worse case getting stuck at local minimum instead of finding global minimum. Navigating to global minimum in the shortest time is therefore very challenging. In order to speed up convergence, different gradient descent optimization algorithms are introduced including

Momentum, RMSProp, Adam, Adagrad etc. A detailed description of those optimizers is given in [45]. Another way is to reduce the learning rate over time (either decrease gradually after each epoch or big drop after a specific number of epochs), also known as learning rate decay. Dr Jason Brownlee asserts it is proven in practice to accelerate training and improve performance when using with stochastic gradient descent [12]. This architecture with only fully connected layers does not seem to perform well in complex tasks because of the following reasons:

- Exploding of parameter: since every neurons are connected to the next layer, there are a huge of parameters which need to be trained. For example: processing an image with resolution of 256x256 RGB will result the input layer of 256x256x3 = 196608 neurons. Using the first hidden layer with for example 5000 neurons to avoid data bottleneck [7] ends up with 983,040,000 weights.

- Missing information: different from image recognition, where it is important to extract not only pixel by pixel but also a region of pixels,for natural language processing tasks such as translation it is essential to remember some information in the past.

- Fix size of input: while reformatting images seems to be fine, finding an efficient way to preprocess texts or videos with variable length is very difficult.

In the following some other common used topologies of NN with better performance are introduced.

2.2.2 CNN and C3D

Convolution is original a mathematical operation (*), which expresses the overlap of a function shifting over another. It is used in image processing as filter such as Laplace or Sobel filter for edge detection. In term of NN, Convolutional Neural Network (CNN) uses such filter, also called filter kernel or convolutional kernel to extract important information from data (features) by sliding the kernel through all possible locations in an image. CNN has gained a lot of success in the field of Image Recognition beginning with LeNet, which was proposed in 1998 and is able to read zip-code, digit and so on, following by AlexNet, ZF-Net, GoogleNet, VGG, ResNet which won the ImageNet Image Large Scale Visual Recognition Challenge ILSVRC on millions of images over thousands of categories in the period of 2012-2015 [2].

[7] information is dropped permanently after a layer because of pressing too much dimensions

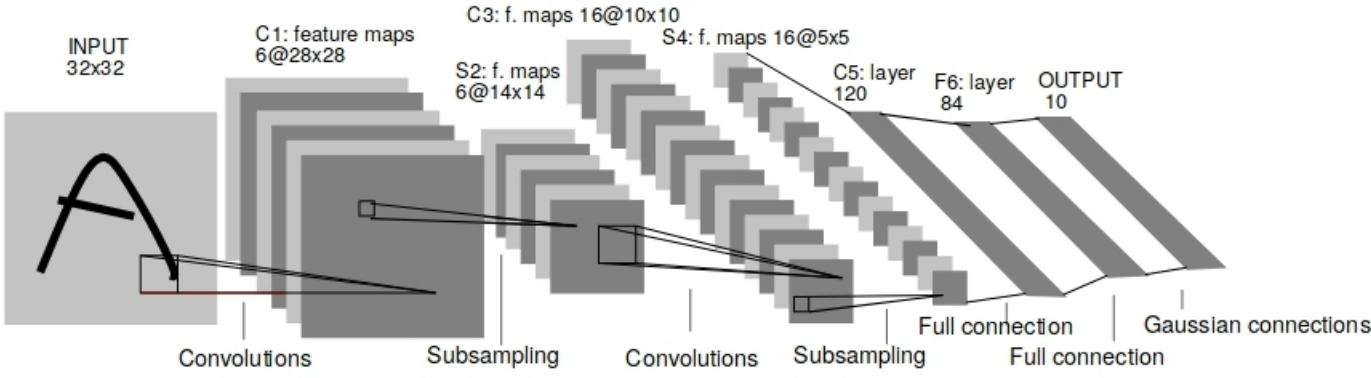


Figure 2.11: Architecture of LeNet-5 for digit recognition

Figure 2.11 shows the architecture of LeNet-5, which was trained and validated on digit grayscale with resolution of 32x32. The typical layers of CNN is to see:

- Convolutional layer CONV
- Pooling layer POOLING
- Fully connected layer FC

CONV is the core block of CNN. It is a 3D arrangement (width, height and depth) of filters, also called feature map since it serves as a feature extraction of the input. Each is constructed by convolving the input with a kernel filter f=nxn by stride s, which means sliding the window filter around input and shifting it by s pixels to the next position. It would continue shrinking the output width and height dimension in deep NN. To deal with this, padding zero p is used by adding p pixels in both dimension. Valid CONV means no padding and same CONV means adding padding so that the output height and width is as in the input. As a result, the output size is:

nxn (padding p) * fxf (stride s) $\rightarrow$ ($\frac{n+2p-f}{s}$ +1) x ($\frac{n+2p-f}{s}$ +1) [40]
For eg. : 32x32 (p=0) * 3x3 (s=1) $\rightarrow$ 30x30

Usually, ReLU is used as activation function after CONV, makes a non-linear information transformation to next layer. ReLU = max(0,x) doesn't require much computation (train faster) and has proven its effectiveness in speed up convergence with stochastic gradient descent.

POOLING is also labeled as downsampling layer. It is constructed by sliding a kernel around the input and taking the average value (AVERAGE-POOLING) or max value (MAX-POOLING) of the subregion in the input. Figure 2.12 shows an example of how MAX-POOLING works.

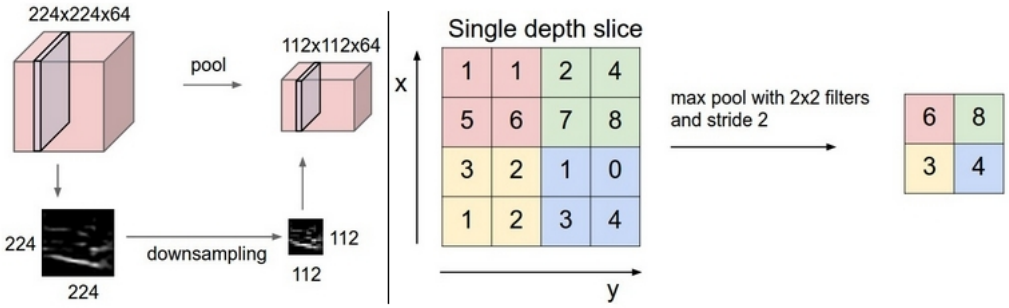


Figure 2.12: MAX-POOLING layer [2]

This layer helps reduce the spatial size while keeping the depth. As a result, it reduces significantly the number of parameters, leading to reducing of computation cost and additionally avoid overfitting. MAX-POOLING is usually used because it is believed that if a feature exists, then a high value should be kept.

FC is like layer in MLP. It locates usually at the end of the model and serves as classification layer.

Stacking all those layers together helps creating a CNN model, which is then able to extract hierarchical features through layers. It is believed that the first layers learn the basic information from an image such as horizontal edge, vertical edge. and in deeper layers more complex features such as a cat eye, a dog nose. Given popular and successful CNN models such as VGG, AlexNet along with others as examples, a similar build pattern is to see: a model consists of a set of CONV layer, followed by a MAX-POOLING layer and some FC layers at the end. Typical kernels are 3x3 and 5x5.

While C2D works well with 2D, 3D images, it does not perform well on 4D data such as video since it preserves only spatial information and not temporal information. [29] proposed in 2010 C3D, also known as 3D ConvNets, which is able to extract features from both spatial and temporal dimensions such as the connection between different frames. Therefore it is best used for action recognition. Figure 2.13 visualizes the difference between C2D and C3D. Clearly to see is that an output feature map of C2D is 2D (spatial features) while of C3D is 3D(spatiotemporal features)

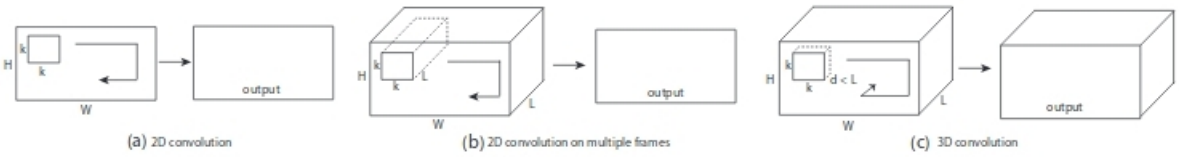


Figure 2.13: 2D and 3D convolution operations: a) 2D on image b) 2D on a video, displayed as a sequence of images c) 3D on a video [50]

2.2.3 RNN and LSTM

Recurrent Neural Network (RNN) is another common used type of NN. While CNN focuses on visual data with fix size such as images, RNN processes typically time series and sequential data with variable length (handle changes over time). For example: stock prices, sales per day, sensor signals, speech, weather data (temperatur, pressure, humid-

ity), web data (clicks and logs) or text (feedback, ratings) [37]. RNN has showed its success in a wide variety of application including: [37]

- Prediction: weather forecast, text auto-completion, next stock price, next mode trend
- Classification: anomaly detection, customer feedback analysis, image and video classification
- Sequence Generation: generate music or text in a given style
- Sequence-to-sequence Transformation: language translation French to German, summary from text, auto caption for image and video

RNN processes each timestep of the data, for e.g. word by word in a sentence. RNN has an internal memory, which allows to remember information of the sequence it has seen so far. Figure 2.14 shows the behavior of **a recurrent neuron**.

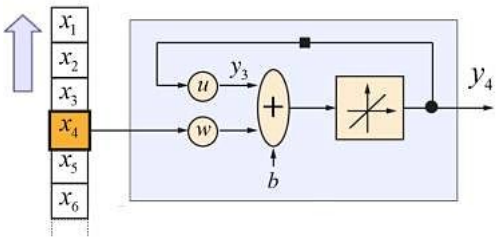
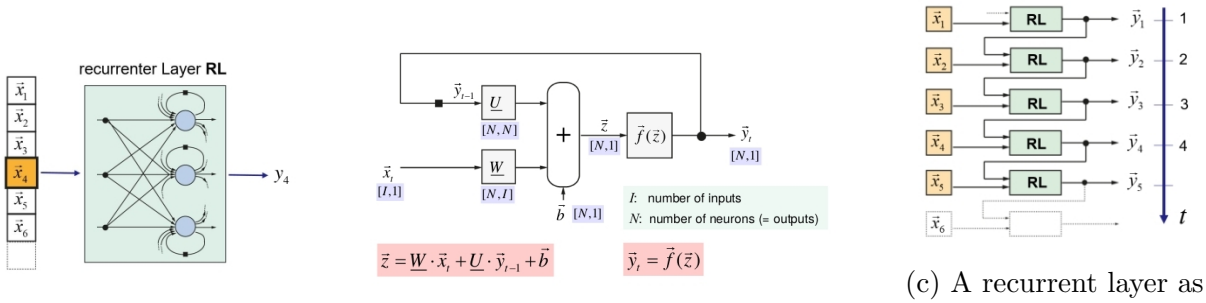


Figure 2.14: A recurrent neuron [37]

A RNN layer contains one or more recurrent neurons. It takes the current timestep data and updates the result of last timestep data by a parameter u and processes from them an output for next timestep. Figure 2.15 shows some kind of visualization of a RNN layer.



(a) A recurrent layer (b) A recurrent layer in matrix form (c) A recurrent layer as unrolled in time

Figure 2.15: Different visualization of a recurrent layer [37]

Figure 2.15 (a) shows a recurrent layer of 3 neurons, processes an input sequence, each timestep data has 3 features (dimensions). (b) shows the matrix form of a recurrent layer of N neurons and processes a timestep of data with I features. The number of parameters can be calculated as P = N*N (feedback weights) + N*I (input weights) + N (bias values) [37]. And (c) shows a recurrent layer as an unrolled network in time. Each timestep is

processed by the same recurrent layer. A model can be created by 1 recurrent layer or by stacking different recurrent layers together.

The main concept of backward training in RNN is Backpropagation Through Time (BPTT). Given a recurrent layer of 4 neurons processing a sequence of T timesteps as shown in Figure 2.16, this network of 1 recurrent layer can be unrolled in time and considered as T feed-forward layers with shared weights. The loss is calculated across all timesteps from left to right and summed up. The weights are then updated from right to left.

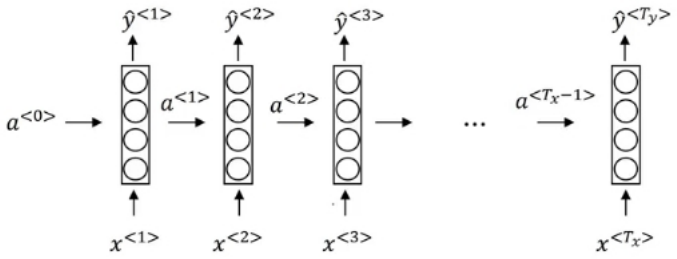


Figure 2.16: A forward recurrent layer [40]

Concretely, consider $a^{<t>}$ is the activation value of timestep t (also usually notated as $h^{<t>}$ [8]), $y^{<t>}, \hat{y}^{<t>}$ is the label and the calculated output of timestep t $\rightarrow L^{<t>} = f(y^{<t>}, \hat{y}^{<t>})$ is the loss of timestep t. L $= \sum_{t=1}^{T} L^{<t>}$ is the total loss of the network. The derivative of L with respect to W can be calculated as

$$\begin{aligned}\frac{\partial L}{\partial W} &= \frac{\partial L}{\partial L^{<T>}}.\frac{\partial L^{<T>}}{\partial y^{<T>}}.\frac{\partial y^{<T>}}{\partial a^{<T>}}.\frac{\partial a^{<T>}}{\partial W} \\ &= \frac{\partial L}{\partial L^{<T>}}.\frac{\partial L^{<T>}}{\partial y^{<T>}}.\frac{\partial y^{<T>}}{\partial a^{<T>}}.\frac{\partial a^{<T>}}{\partial a^{<T-1>}}.\frac{\partial a^{<T-1>}}{\partial a^{<T-2>}}....\frac{\partial a^{<1>}}{\partial W}\end{aligned}$$

In practice, BPTT can be slow while training with a very long input sequence (very big T) because of too much repeating of the derivative from one timestep to another. Plus if the gradient is too big or too small, the process of repeating can lead to exploding and vanishing gradients problem. A solution for speeding up training is Truncated Backpropagation Through Time (TBPTT). TBPTT is a modification of BPTT, which limits the number of timesteps used on forward- and backpropagation, notated as TBPTT(k1,k2) with k1 is the number of timesteps in forwardpropagation until updating and k2 is the number of timesteps used for updating weights in backpropagation. If k1=k2=T then it is the normal BPTT as showed above. A commom configuration is k1=k2=a fixed number c < T means every c forward timesteps, BPTT update is performed for c timesteps. While "clipping the gradients" at a pre-defined threshold seems very effective when dealing with exploding gradients, there is no complete solution for vanishing gradients. A careful weight initialization as well as usage of ReLU activation function might reduce the effect of vanishing gradients. As a consequence, simple RNN (vanilla RNN) is hard to train and the information can not be memorized in long-term. Under RNN, there are some special architectures which performs better on those problems such as Long Short

[8] h as hidden layer output

Term Memory (LSTM), Gated Recurrent Unit GRU.

LSTM was first proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [27]. It solves the struggle of vanilla RNN and is designed to memorize long-term dependencies as its default behaviour. The core idea of LSTM is gate control GC, which manages the flow of information including: [37]

- Select the important information
- Memorize the information until it is not necessary anymore
- Pass the information only if it is important
- Forget the information, that has become unimportant

Main components of LSTM are Input Unit IU, Input/Output/Forget Gate Control notated as i-GC/o-GC/f-GC as shown in Figure 2.17

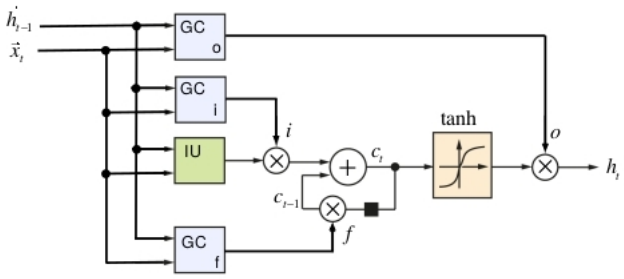


Figure 2.17: A LSTM cell [37]

Unlike RNN, LSTM consists of a cell state which runs through all timesteps. Information in cell state can be added, updated or removed, which is done carefully by 3 GCs. Each GC is composed out of a sigmoid layer with value range (0,1). The bigger it is, the more information is let through. f-GC looks at h_{t-1} and x_t and decides if the information from the cell state should be completely kept ($f_t = 1$), partly kept or totally forgotten ($f_t = 0$).

$$f_t = \sigma(W_f.x_t + U_f.h_{t-1} + b_f)$$

IU calculates the information in the current step $\widetilde{C}$ and based on the value of i-GC, $\widetilde{C}$ is updated respectfully. The current cell state is now calculated as a sum of the left information from old cell state (value of f-GC) and the new information (updated $\widetilde{C}$)

$$\widetilde{C} = tanh(W_c.x_t + U_c.h_{t-1} + b_c)$$
$$i_t = \sigma(W_i.x_t + U_i.h_{t-1} + b_i)$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}$$

Finally o-GC decides which information should be passed to the next step.

$$o_t = \sigma(W_o.x_t + U_o.h_{t-1} + b_o)$$
$$h_t = o_t \odot tanh(C_t)$$

$W_f, U_f, b_f, W_i, U_i, b_i, W_c, U_c, b_c, W_o, U_o, b_o$ are all learnable parameters. The art of LSTM is about **learning how to learn** by learning which information to keep, forget or to output. Those information are then used to solve the tasks.

2.2.4 Hybrid

Each special NN can learn specific features well such as CNN with spatial features and LSTM with temporal features. C3D for example can learn both, but quite shallow while learning representations in time. For tasks which requires both, a better solution is called Hybrid NN. It contains different kind of layers. A popular type of Hybrid NN is known as Long-term Recurrent Convolutional Network (LRCN) which is proposed by [20]. It is mostly used for action recognition, image and video autocaption. According to [20], this kind of model is "doubly deep", "deep in space" thank to CNN as front end layer to encode the spatial features from each frame and "deep in time" thank to LSTM to extract long-term dependencies among frames. Figure 2.18 visualizes the LRCN model

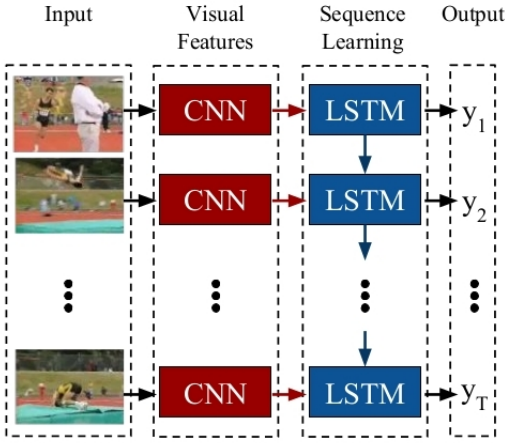


Figure 2.18: LRCN Model [20]

2.3 Training techniques

The aim of training is again to find the set of parameters which maps input to output best. In term of DL, best fit means a model which can generalize well, performs well not only on training data but also on unobserved input. To measure the performance best, the data is generally divided into 3 sets: training set, dev set (also called validation set) and test set. The reason for the need of split of dev set and test set is that dev set is used to validate the effect of parameter tuning during training process and as a result somehow being effected. A neutral test set is therefore necessary for final validation of the model. Since there are a lot of parameters to tune such as number of layers, number of neurons in each layer, type of layer, learning rate, optimizer, weight initialization and so on, training a deep learning model is in general very challenging and time-consuming. In addition, there is also no guideline which guarantees 100% good result on all models but it mostly depends on the kind of problem. Therefore, a good understanding of how the model performs and what to do with that is very important so that the parameters can be tuned in the right direction. 2 main concepts to describe the performance of the model are Overfitting and Underfitting.

2.3.1 Overfitting and Underfitting

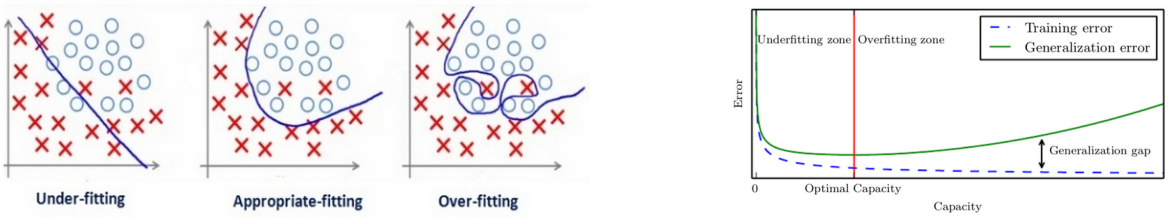
Bias and Variance [9]
Bias: an error from erroneous assumptions in the learning algorithm. Bias shows the

ability of learning from training data. A high bias shows that the model might miss the relevant information and as a result is not able to map the input to the output correctly. **Variance**: an error from sensitivity to small fluctuations in the training set. Variance shows stability of the model when it comes to different data and as a result the ability of generalization of the model. A simple estimation of bias and variance is: (Bias = Train error - Human error) and (Variance = Validation error - Train error) with examples showing in Table 2.2

Human error	0%	0%	0%
Train error	15%	1%	0.5%
Validation error	16%	11%	1%
Bias	15%	1%	0.5%
Variance	1%	10%	1%
	high bias + low variance	low bias + high variance	low bias + low variance

Table 2.2: Example of bias and variance

A high bias and low variance is normally symptom of Underfitting, where the model performs poorly on the training data and consequently not able to generalize to new data. A low bias and high variance is then of Overfitting, where the model learns too well on the training data, even detail and noise, leads to negative impacts on the generalization of the model to new data. A low bias and variance is in most case the best fit. Figure 2.19 shows a visual example of Overfitting and Underfitting from the viewpoint of data and error.



(a) data view [40]

(b) error view [32]

Figure 2.19: Overfitting and Underfitting

A basic recipe to deal with Overfitting(low bias + high variance) and Underfitting(high bias) is summarized by [40]

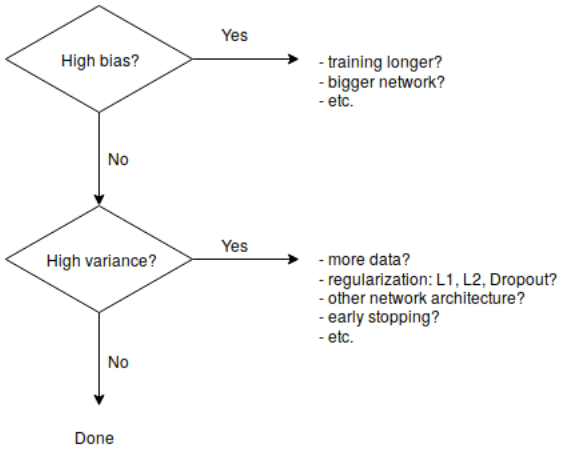


Figure 2.20: ML recipe

Some of those techniques will be introduced in the following.

2.3.2 Data Augmentation

Data is the center key of learning. To be able to map the input to the output correctly by a huge set of parameters, the model needs to see enough examples to figure out the pattern, muster and rule. Besides that, the data must be reliable, means good quality of data and correct label. Data from different distributions need to be handled carefully. With the rise of internet, it's now partly simpler to collect data, search for available datasets and create ground truth. Still getting enough reliable data for training is never an easy task, especially when it comes to specific tasks. Data is expensive, in the sense of money for its value, time to collect it, human and computation resources to prepare it. In many cases, access to data is heavily protected due to privacy or security concern such as in medical or military industry, getting more data is hardly possible. For task like cancer detection, Data Augmentation is the essential key. Data Augmentation is widely used in many ML/DL applications and has proven its effectiveness in reducing overfitting and improving model performance. The art of Data Augmentation is "inventing" more data from the available data by a variety of transformation methods. Not only it creates more data, but it also helps increase the diversity of data by inventing more context and background for available data, which improves generalization. Data Augmentation is divided in 2 types:

- **online**: augment data and save it to create a new dataset. It is suitable for small datasets.
- **offline**: big datasets for example can not suffer from exploding in data size. Augmentation of data is then executed at run time and in mini-batch. Keras offers for example ImageDataGenerator for generating more data during training. Some examples are to see in Figure 5.11

(a) original (b) vertical flip (c) shift (d) rotate (e) horizontal flip

Figure 2.21: Examples of Data Augmentation

Some classical techniques of augmenting visual data can be counted: flipping horizontal and vertical, random rotation, shifting, random cropping, adding noise (gaussian noise, salt and pepper noise), equalizing histogram, changing light condition etc. Choosing the right methods depends on the type of data and problem. Flipping a car horizontally might make sense, but vertically is hardly to see except in car accidents.

2.3.3 Regularization (L1, L2, Dropout)

A very common and effective technique to avoid Overfitting is Regularization. Regularization is defined as any supplementary technique that aims at making the model generalize better [30]. [30] also pointed out and discussed about 5 main elements which can contribute to regularization

- The training set
- The selected model family
- The error function
- The regularization term added to cost function
- The optimization procedure itself

Within the scope of the thesis, only L1, L2 and Dropout will be introduced as next.

The core idea of L1,L2 Regularization is to add a regularization term to the cost function J. Unlike J, the regularization term assigns a penalty to the model not based on the target but on other criteria. In term of L1,L2 it is the weights. Bias can be left unregularized in this case since the number of bias parameters is quite small in comparison to weights and as a result does not have much impact on the regularization.

$$
\begin{aligned}
J &= \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}_i, y_i) \\
\tilde{J} &= J + R \\
R &= \frac{\lambda}{2m}\|w\|^2 = \frac{\lambda}{2m} W^T.W (L2) \\
R &= \frac{\lambda}{2m}\|w\| \, (L1) \\
\|w\|^2 &= \sum_{i}^{n^{[l-1]}} \sum_{j}^{n^{[l]}} w_{ij}^2 (\text{Frobenius norm}) \\
\|w\| &= \sum_{i}^{n^{[l-1]}} \sum_{j}^{n^{[l]}} w_{ij}
\end{aligned}
$$

While λ is called the regularization parameter. Higher λ corresponds more regularization. The added sum of (square) weights assigns that if weights are big, then cost increases too which forces the learning algorithm to keep weights small. Small weights and in many case close to 0 weights deactivate corresponding some neurons leads to a smaller network and consequently avoid overfitting. L1 Regularization sums the absolute value of weights and therefore can provide sparse output as feature selection (zero irrelevant features)

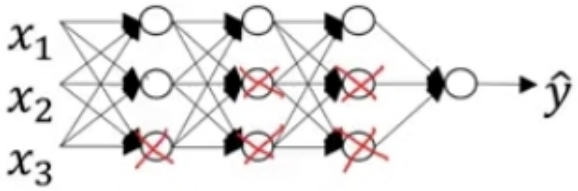


Figure 2.22: L1,L2 Regularization behaviour [40]

L2 Regularization is also known as weight decay. On each step, the weights are updated as follow:

$$\begin{aligned} dW^{[l]} &= \frac{\partial \tilde{J}}{\partial W^{[l]}} = \frac{\partial J}{\partial W^{[l]}} + \frac{\lambda}{m}.W^{[l]} \\ W^{[l]} &= W^{[l]} - \mu dW^{[l]} = W^{[l]} - \mu(\frac{\partial J}{\partial W^{[l]}} + \frac{\lambda}{m}.W^{[l]}) \\ W^{[l]} &= W^{[l]}(1 - \mu\frac{\lambda}{m}) - \mu.\frac{\partial J}{\partial W^{[l]}} \end{aligned}$$

In comparison to the old parameter update $W^{[l]} = W^{[l]} - \mu.\frac{\partial J}{\partial W^{[l]}}$, the weights multiplicatively shrink by a constant factor $(1 - \mu\frac{\lambda}{m})$ in each update step.

A well known stochastic regularization technique is Dropout. The key idea of Dropout is to randomly drop units. It means removing it as well as all its incoming and outgoing connections during training as shown in Figure 5.14. By doing this, in each training step a new "thinned" network is trained and evaluated. Training a neural network of n units then can be seen as training 2^n possible thinned networks [48].

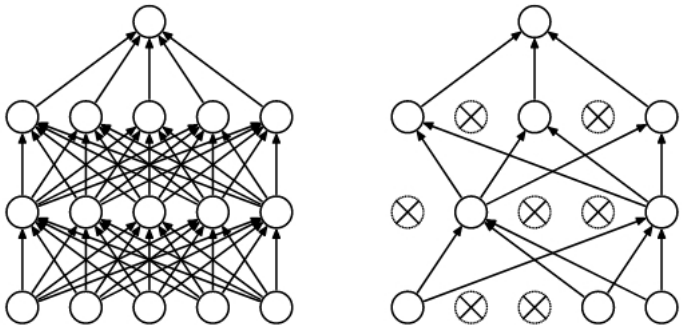


Figure 2.23: Dropout Neural Net Model. Left: A standard neural net with 2 hidden layers. Right:An example of a thinned net produced by applying dropout to the network on the left. Crossed units have been dropped.[48]

According to [48] models combination nearly always improves the performances of machine learning methods. Dropout issues exactly the problem and provides an efficient way of training different models. Also by deactivating a number of neuron, the network becomes smaller and consequently avoids overfitting. Dropout is therefore one of the best regularizer currently. In test time, a single network without dropout as a combination of all subnetworks at training is used. Since the existence of neurons depends on a probability keep_prob ([0,1] with 0 is drop all and 1 is keep all neuron) during training, the weights need to be rescaled at test time to have the expected output with respect to the distribution used to drop units. A simple implementation using "inverted dropout technique" in python is to see, where the expected activation output at training is rescaled. By doing that, rescaling weights at test time can be skipped.

```
# implement dropout for layer 3
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
a3 = np.multiply(a3,d3)
a3 /=keep_prob #inverted dropout
```

2.3.4 Normalization (Input, Batch)

The problem of input with different range is that the parameters associate to it will too. Consequently, a small change on some parameters might have huge impact and their gradients will dominate the parameter updating. This unbalance can make the processing of finding global minimum of loss function slower. Input Normalization therefore helps speed up training by resolving this problem. It corresponds 2 steps: subtract the mean μ and normalize the variance σ of features vector x

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \text{(m is number of samples)} \\ x &= x - \mu \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m} (x^{(i)})^2 \text{(element square)} \\ x &/= \sigma^2 \end{aligned}$$

with μ and σ are the mean and variance vector for all features. The same μ and σ should be applied for test set to ensure that all data is transformed in the same way.

In a deep NN, the distribution of each layer's input changes during training because of the change of parameters in last layer. [28] refers this phenomenon as internal covariate shift and proposed a technique called Batch Normalization to reduce this effect. Unlike Input Normalization, Batch Normalization performs the normalization for each training mini-batch and in also hidden layers. This results a more stable distribution of input to each layer and accelerate training. By fixing the mean and variance of each layer input, consequently reduce the dependency of gradients on parameter scale, it allows higher learning rate without the risk of divergence and more flexible at weight initialization. In addition, [28] also pointed out the regularization effect of Batch Normalization, in many case reduces the need of Dropout. There are many discussions around whether the layer input Z or the activation value of that layer A = f(Z) should be normalized. In practice

the input before activating is more often used.

$$\begin{aligned} \mu &= \frac{1}{m}\sum_{i=1}^{m} Z^{(i)} \\ Z^{(i)} &= Z^{(i)} - \mu \\ \sigma^2 &= \frac{1}{m}\sum_{i=1}^{m}(Z^{(i)})^2 \text{(element square)} \\ Z^{(i)}_{norm} &= \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}} \\ \tilde{Z^{(i)}} &= \gamma . Z^{(i)}_{norm} + \beta \end{aligned}$$

with ϵ is a very small value to avoid zero division, γ and β are learnable parameter and can be updated as weights, which allows different distribution value.

2.3.5 Transfer Learning

Transfer Learning is as its name about transfer what have learnt. The art of Transfer Learning is to apply knowledge learnt from a task to other. Training a model from scratch is hard in sense of consuming resources (time, money, human, computation). Since many institutions and researchers release their work every year, reusing their pre-trained models is a much better idea if the learnt features can be transferred. [53] investigated exactly the transferability of feature and factors which effect it. Their results show that the transferability of features decreases as the distance between the base task and target task increases, but even transferring features from distant tasks can be better than using random features. Transfer Learning is not a new approach in Computer Vision tasks. Since CNN has shown its ability of learning general features to specific as the network goes deeper, Transfer Learning is widely used in visual tasks as the general features can be applied anywhere. A very common approach is to use a pre-trained model as baseline, leave the general features layers frozen or fine-tuning some at the end and train a classifier with new data on top of it. There are a variety of pre-trained models on a huge set of data in image recognition task, which is integrated directly in many frameworks such as VGG16, VGG19, ResNet, Inception etc. Transfer Learning in Natural Language Processing (NLP) is currently limited at the transfer of Word Embedding. There are thus more and more research and work on generating a transferable baseline for NLP tasks. The result is expected in a near future.

2.4 Evaluation techniques

DL is a empirical process. It includes having an idea, implementing it, deploying it and based on the result to create new trial idea. Therefore, setting up the way to evaluate the model is one of the essential step of DL.

Optimizing and satisfying metrics
For each task, there might be different requirements, which need to be considered such as accuracy, computation time, error rate, along with others. In a contrast, evaluating a classifier during training based on different evaluation metrics might be very complicated.

Therefore, a better approach is known as choosing optimizing and satisfying metrics. Among N requirements, there should be 1 optimizing metric and N-1 satisfying metrics. The optimizing metric is as its name to optimize the best model while the other satisfying metrics only need to overcome a defined threshold. Accuracy of the model is common used as optimizing metric. The common understanding of accuracy in classification is

$$Accuracy = \frac{\text{Number of correct predictions}}{\text{Total number of data}}$$

In case of unequal proportion of samples for different classes, this kind of accuracy could be unreliable. For example a 97% accuracy in a model of 2 classes: 97% data for class 1 and 3% for class 2 could still fall in the case where the model predicts all class 2 incorrectly, which makes the model a bad choice. In addition, in case of classification of small set of classes, the coincidence factor should be considered carefully too. To have a clean and detailed view of model performance, Confusion Matrix is used.

Confusion Matrix presents all classes and their prediction results in a matrix. The diagonal elements represent the number of correctly predicted data while off-diagonal elements are those that are mislabeled. The higher the diagonal values are, the better the model predicts. In case of class imbalance, normalization the Confusion Matrix with respect to the number of data in each class will contribute to a better view of model performance. Figure 2.24 shows an example of Confusion Matrix of a classifier of Iris flowers, as well as its normalized version.

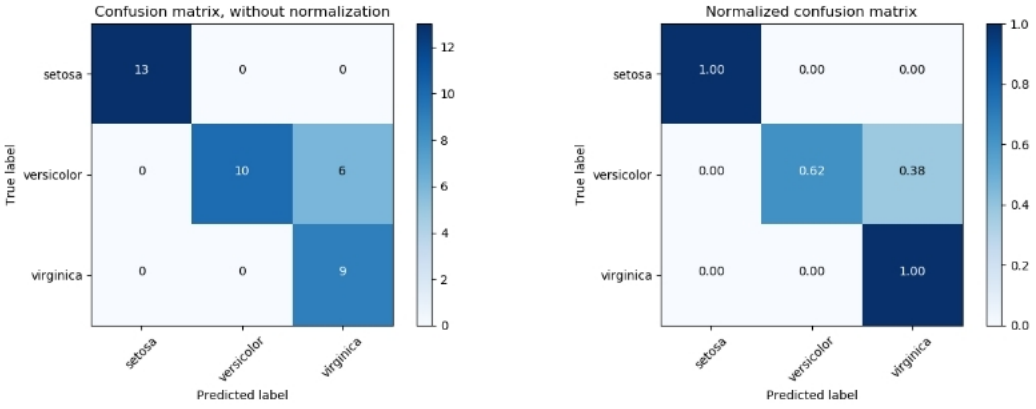


Figure 2.24: Confusion Matrix [13]

Another well-known metric for ML/DL is F1 Score

F1 Score is a combination of the precision P and the recall R, also normally called Harmonic Mean.

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

while the precision is defined as the number of items correctly predicted as positive (True Positive) out of total item predicted as positive (True Positive + False Positive) and the recall is defined as the number of items correctly identified as positive (True Positive) out

of total positive data (True Positive + False Negative).

$$P = \frac{TP}{TP + FP}$$
$$R = \frac{TP}{TP + FN}$$

Clearly, a low precision corresponds a high number of False Positive and a low recall shows a high False Negative. Therefore, precision can be seen as a measure of exactness while recall is a measure of completeness.

Beside the model itself, there could be other factors which effect the model performance such as the preparation of data. Unclean data or data mismatch can sometimes cause many pain. To issue those problems, a simple metric might fail. In those case, a detailed Error Analysis is necessary.

Error Analysis is executed in all wrong labeled data with detailed notice. It is a manual job but will have huge contribution to training by orientating training in the right direction.

Chapter 3

Tools

This chapter is an introduction to the tools and libraries which are used for creating, training and evaluating models as well as for preparing data.

3.1 OpenCV

OpenCV is an open-source library under BSD license for Computer Vision and Machine Learning. OpenCV is written in C/C++ and is designed for computational efficiency. Its main goal is real-time image/video processing. It provides interfaces in C++, Python and Java and supports Windows, Linux, Mac OS, iOS and Android. OpenCV is used in this work mostly for face detection, tracking and data augmentation.

3.2 H5py

H5py is a Pythonic package for HDF5 format. HDF5 (Hierarchical Data Format v5) is an open-source technology to store and manage extremely large and complex data collections. HDF5 supports principally datasets of any size and n-dimensional data unit. Simply a HDF5 file could be thought as a folder of different subfolders. It stores data in binary format and allows access to a part of data without reading the entire file. Therefore, it is widely used in Deep Learning application. H5py provides an easy-to-use high level interface to work with HDF5 files. H5py is applied in this work to create the final datasets used for training and evaluating.

3.3 Keras and Tensorflow

Tensorflow was originally developed by Google and is now an open-source library under Apache Open Source license. Tensorflow is used broadly in ML/DL applications. Basically it provides efficient numerical computation through a flow of operations with tensors (data unit of n-dimension). Because of its flexible architecture, Tensorflow can be deployed in a variety of platforms such as CPU, GPU or TPU. Keras serves as the official high-level API of Tensorflow [14]. Besides that, it can also run on top of other backends such as CNTK or Theano.

Figure 3.1: Keras [14]

Keras is an open-source library under MIT license and is written in Python. It contains implementations of many NN layers, optimizers, activation functions, pre-trained models, datasets etc. In addition to its functionality, it focuses strongly on user's experience by providing a simple, consistent and friendly API. It allows trying more ideas and faster. [14] shows 3 API styles including:

- Sequential Model: dead simple, single input single output and good for 70+% of use cases
- Functional API: more control over every layer, multi input multi output and good for 95% of use cases
- Model subclassing: maximum flexibility but larger potential error surface

In this thesis, Sequential Model is used.

3.4 Others

matplotlib is a plotting library. It is a very easy-to-use tool for visualizing any kind of graphs in a variety of setting from size, color, style etc. It is especially helpful in this thesis by showing the loss and accuracy during training.

scikit-learn is a well-known machine learning library. It is characterized as a clean, uniform and streamlined API with very useful and complete online documentation with examples. It contains a variety of supervised and unsupervised learning algorithms, as well as data statistic tool and evaluation. Scikit-learn is used in this work only for data analysis, gridsearch for parameter tuning and confusion matrix for model evaluation.

numpy is a Pythonic library, which provides a high performance of large, multi-dimensional arrays and operations with them. As a result, it is used broadly by many other libraries.

jupyter notebook is an open-source web application, which allows create code, its output and instructions in the same page

To summarize, the following versions of those libraries are used:

- OpenCV 3.4.3
- H5py 2.7.1
- Tensorflow 1.12.0
- Keras 2.2.4
- Matplotlib 2.2.3
- Scikit-learn 0.19.1

Chapter 4

Workflow

A general workflow of a DL project is described in Figure 4.1:

Figure 4.1: Deep Learning general workflow

There are 3 main steps. Step 1 is called Data or "procure" data. It is about collecting data from different resources, creating a new dataset, creating ground truth for data, "inventing" more data from available data, cleaning mismatched data, preprocessing data and storing data in hard disk or cloud etc. In summary, it provides data for step 2 called Model Training. This step includes making decision about model architecture, building it and training it. Training is in the sense of tuning hyperparameters of the model. This process is empirical. Resources such as human knowledge, hardware, memory, frameworks, toolkits etc. play an essential role in this step. Also a good metric for evaluating the model during training is very important in term of driving the tuning process in the right direction fast. The result of this step is the best trained model. It is then evaluated one last time in step 3 called Model Evaluation. This step shows the performance of the trained model in the test set and a detailed overview of the result. This workflow is not always sequential. In many cases, Model Training (Step 2) takes too much time and does not output any reasonable result, an error analysis might show data mismatch, which needs to be cleaned up again in step 1 (Data). In other cases, the trained model performs well in Step 3 (Model Evaluation) but fails while deploying in real application, the model might need to be trained again (Step 2) with more/different data (Step 1).

The workflow in this thesis is expanded from the general one above.

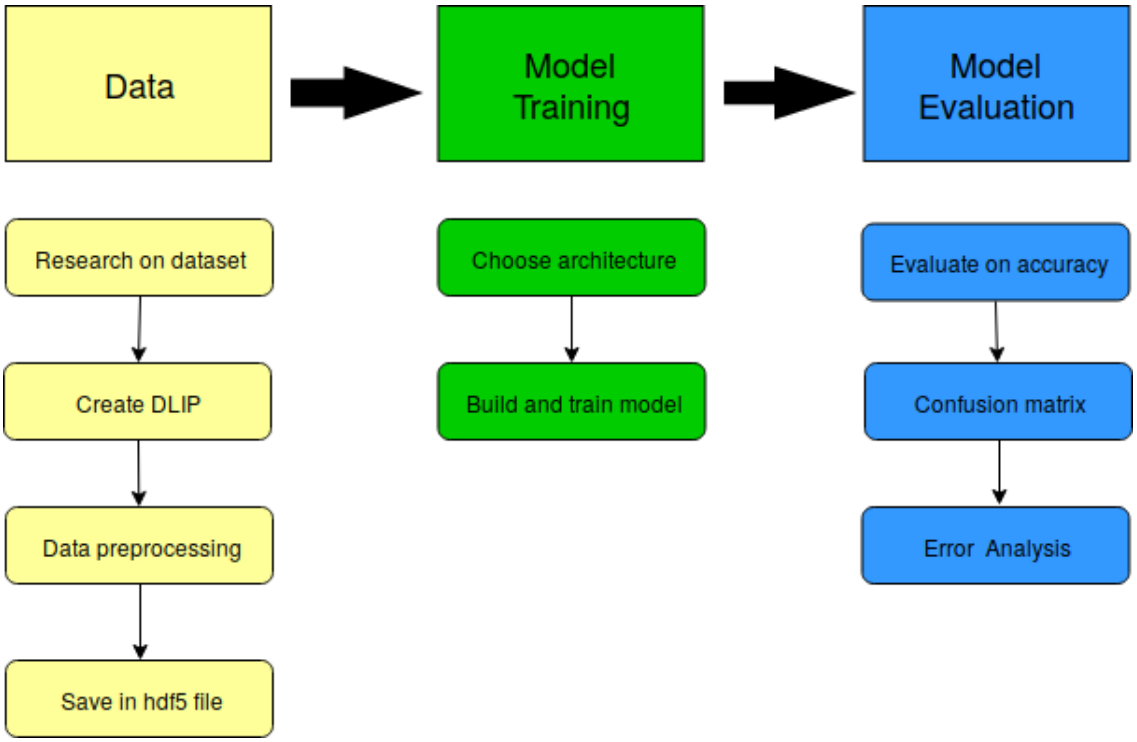


Figure 4.2: expanded Deep Learning Workflow

To the best of the author's knowledge, most work on Lip Reading focus on the English language and there is no existed work on the German language. As a result, there is no available LR dataset in German. DLIP (**D**eutsches **Lip**penlesen) is created. DLIP records short videos, each contains a person speaking a word in a quiet environment. The videos are then preprocessed one by one as following: detecting the face in the first frames, initializing a tracker to track the face through the video, cutting the tracking area, resizing it and saving it as gray frames sequence in hard disk. Lastly, the data is augmented, divided in train, validation and test set and stored respectfully in hdf5 file.

For training, 3 model architectures are chosen: C3D, LSTM and LRCN. Each model is built and trained on Keras with Tensorflow backend. The trained model is evaluated during training on the validation set based on loss function and accuracy metric.

Finally, from each architecture, a best trained model is selected, taken into comparison and validated on test set. A confusion matrix gives the detailed prediction of every word and an error analysis shows a table of possible reason for the wrong prediction.

The whole training and validation process is done in a jupyter notebook.

Chapter 5

Implementation

This chapter provides a step-by-step implementation of the workflow mentioned in the previous chapter.

5.1 Datasets

5.1.1 Research

A search for data online and offline, especial in Google Dataset Search and Kaggle resulted:

- most datasets are for the English language, 1 for the Urdu language and none for the German language
- many datasets are created in the laboratory while other are collected from popular television news, talk shows such as BBC, CNN, TEDTalk
- most datasets focus on word level or phrase level while some record short sentences or a part of a sentence
- most datasets are recorded isolatedly, means a person speaks a word/a sentence while some do continuously

Some datasets and their character as well as their best recorded performances can be seen at the table 1.1

5.1.2 Creating DLIP dataset

Consequently, a new dataset for german LR needs to be created. DLIP is inspired by MIRACL-VC1 dataset [25]. 15 people were asked to read 12 German words in a quiet environment, each 10 times. Out of 15, 9 are men and 6 are women. The words are chosen due to the ranking of the most common German words from Duden [1] [19]. In addition, the words are selected in 3 categories: noun, verb and adjective. Lastly they are orthogonal in lip movements. Meeting all those requirements are the following selected words:

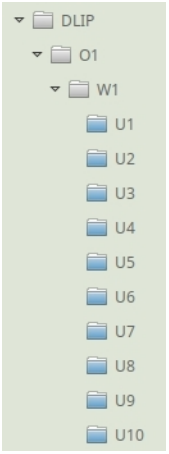
[1] Duden is a dictionary of the German language and serves as the official standard for German spelling. The ranking is from the online portal of Duden.

1.Prozent	5.wenig	9.werden
2.Million	6.vergangen	10.haben
3.Mann	7.lang	11.können
4.Ende	8.hoch	12.stellen

Table 5.1: Recorded words in DLIP

DLIP consists of 15 people*12 words*10 times/each word = 1800 short videos, recorded with resolution of 640x480 and 30 fps. DLIP is structured in 15 folder for each object person (O1-O15), each contains subfolder of 12 words (W1-W12), each word folder contain 10 unit folder corresponding 10 frame sequences of that word (U1-U10). Figure 5.1 shows an example of a setup and DLIP's folder structure.

(a) DLIP setup



(b) DLIP folder structure

Figure 5.1: DLIP setup and folder structure

5.2 Preprocessing

Be aware of the limit of resources, the final data unit was tried to keep as small and as informative as possible. Facial expression such as forehead, eyebrows or cheeks can deliver in many cases some related information. However, to pretend the model from focusing on less important features, only mouth region is used as final data since the lip movements contain most information. From original data, Haar Cascade classifier and tracker in OpenCV was used to cut the mouth area.

5.2.1 Face and mouth detection with Haar Cascade, Tracking with CSRT

There are available pre-trained Haar Cascade classifier, where the parameters are saved in xml files and can be loaded in OpenCV. Since the object person sat directly in front of the camera, pre-trained frontal face and mouth detector were used, which can be downloaded from OpenCV github [3] [4].

```
# loading pre-trained detector
ap = argparse.ArgumentParser()
ap.add_argument("-f", "--face",
```

```
                default="/haar_cascade/haarcascade_frontalface_default.
                                                        xml",
                help="path to haar face detector")
ap.add_argument("-m", "--mouth",
                default="/haar_cascade/haarcascade_mcs_mouth.xml",
                help="path to haar mouth detector")
args = vars(ap.parse_args())
face_detector = cv2.CascadeClassifier(args["face"])
mouth_detector = cv2.CascadeClassifier(args["mouth"])
```

Detecting face and mouth in all frames was the first option since the frame sequence for a word is in general short and contains only small motion of the face and the background is stable. Applying this method, the detector failed in many cases when the mouth shape is too small (Mann) or too big können, haben). Figure 5.2 shows some examples where the face is detected correctly but mouth is not detected.

(a) haben (b) können (c) Mann

Figure 5.2: Example of bad mouth detection

A much better solution was Detection + Tracking. The detectors were applied to all the data unit (a data unit = a frame sequence). A tolerance of detect_frame_allowed = 10 frames was given, means trying to detect face and mouth in the first 10 frames. In case of no face or mouth detected, that data unit would be skipped.

```
# detect mouth in the first detect_frame_allowed frames
detected = False
fcount = 0
while not detected and fcount < detect_frame_allowed:
  fcount = fcount + 1
  frame = images.pop(0) # images is a sorted array of all frames in a
                                        data unit
  # detect face
  faces = face_detector.detectMultiScale(
                frame,
                scaleFactor=1.1,
                minNeighbors=10,
                minSize=(30,30),
                flags = cv2.CASCADE_SCALE_IMAGE)
  # face detection incorrect -> next frame
  if (len(faces) != 1):
    continue
  else:
    # detect mouth in haft bottom of the face
    (fx,fy,fw,fh) = faces[0]
```

```
    faceROI = frame[fy+ fh/2:fy+fh, fx:fx+fw]
    mouths = mouth_detector.detectMultiScale(
                    faceROI,
                    scaleFactor = 1.1,
                    minNeighbors=10,
                    minSize=(10,10),
                    flags = cv2.CASCADE_SCALE_IMAGE)
    # mouth detection incorrect -> next frame
    if (len(mouths) != 1):
          continue
    else:
      # create bounding box for tracking, extend mouth region by
                                    extend_pixel pixels in both
                                    direction
      x_scale = fx+ mx - extend_pixel
      y_scale = fy + fh/2 +my -extend_pixel
      h_scale = mh+extend_pixel*2
      w_scale = mw +extend_pixel*2
      mouth_box = (x_scale, y_scale, w_scale,h_scale)
      detected = True
      break

if (fcount >= detect_frame_allowed):
  print ("Could not detect mouth in the first 10 frames")
  # skip this data unit and process next data unit
else:
  print ("Mouth detected after {} frames".format(fcount))
  # save first detected mouth
```

To detect a face or mouth **detectMultiScale** was used. It detects face or mouth of different sizes and returns a list of rectangles containing the detected object. It has 5 parameters as following:

- image
- scaleFactor: How much the image size is reduced at each image scale. This value is used to create the scale pyramid in order to detect faces at multiple scales in the image (some faces may be closer to foreground, and thus be larger, other maybe smaller in background, thus the usage of varying scales). A value of 1.05 indicates that the size of image is reduced by 5% at each level of the pyramid [44]
- minNeightbors: how many neighbors each window should have for the area in the window to be considered a face. The cascade classifier will detect multiple windows around a face. This parameter controls how many rectangles need to be detected for the window to be labeled as a face.
- minSize: a tuple of width and height indicating the minimum size of the window. Bounding boxes smaller than this size are ignored.
- flags: cv2.CASCADE_SCALE_IMAGE will downscale the image rather than "zoom" the feature coordinates in the classifier cascade for each scale factor
- maxSize: default as image size and is not specific configured in this case

The default setting of those parameters does not always work well. It sometimes labels the wrong area as face or misses face. According to [44], tuning with scaleFactor and

minNeighbors used to have good affect. Using trial and error with parameters (scaleFactor, minNeighbours and minSize) on a part of the dataset, the parameters were chosen as shown in code. It is also to see, that eyes area was sometimes mistaken as mouth. Therefore, in the final version, mouth was detected only in the haft bottom of the face. After mouth was detected, the mouth area was used as bounding box for tracking. To improve tracking result, the mouth area was extended by extend_pixel=9 in both height and width dimension. Table 2.1 shows different trackers of OpenCV with their own characters, advantage and disadvantage. Among them, MedianFlow, KCF and CSRT were tested on 120 samples and their performances are summarized in table 5.2

Tracker	Failed samples	Time/sample
KCF	47	0.478 s
MedianFlow	4	0.32 s
CSRT	0	2.14 s

Table 5.2: Result of MedianFlow, KCF and CSRT tracker

While KCF and MedianFlow have big advantage of processing time, they failed sometimes, especial KCF. MedianFlow performed actually well enough, but according to observation, the bounding box was extended very large and it failed at even small movement. Since the data is expensive in term of resources (time, object people), DLIP is a quite small and clean dataset. Therefore, to avoid losing more data, accuracy is chosen over time. CSRT had the highest accuracy and was the final used tracker. Successfully detected mouth was finally cut, transformed to grayscale, resized to 48x32 and saved to harddisk. Grayscale was used to keep the data small since the lip movements remains and the color channels are not so important. Cropped mouth was resized by ratio 6:4 with interpolation option cv2.INTER_LINEAR since it has an average result at zooming as well as shrinking image [18].

```
# start tracking
tracker = # create CSRT tracker
tracker.clear()
# initialize tracker with mouth_box from detection part
bbox = tracker.init(frame, mouth_box)
while len(images) > 0: # images is an array of frame sequence
  frame = images.pop(0)
  ret, bbox = tracker.update(frame)
  if ret:
    # tracking successully
    x_value = int(bbox[0])
    y_value = int(bbox[1])
    w_value = int(bbox[2])
    h_value = int(bbox[3])

    #resize and write result to file
    gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
    mouth_crop = gray[y_value:y_value+h_value, x_value:x_value+w_value
                                        ]
    mouth_crop_resize = cv2.resize(mouth_crop, (48, 32), interpolation
                                        = cv2.INTER_LINEAR)
    # save mouth_crop_resize to harddisk
  else :
    # tracking failed
```

Below is an example of the processing for the word **Prozent**. Figure 5.3 shows the Detection + Tracking of face and mouth. For the sake of visualization, only the face is showed instead of the whole frame. Figure 5.4 shows the final cut, resized and rescaled mouth area, which is used for training and testing later.

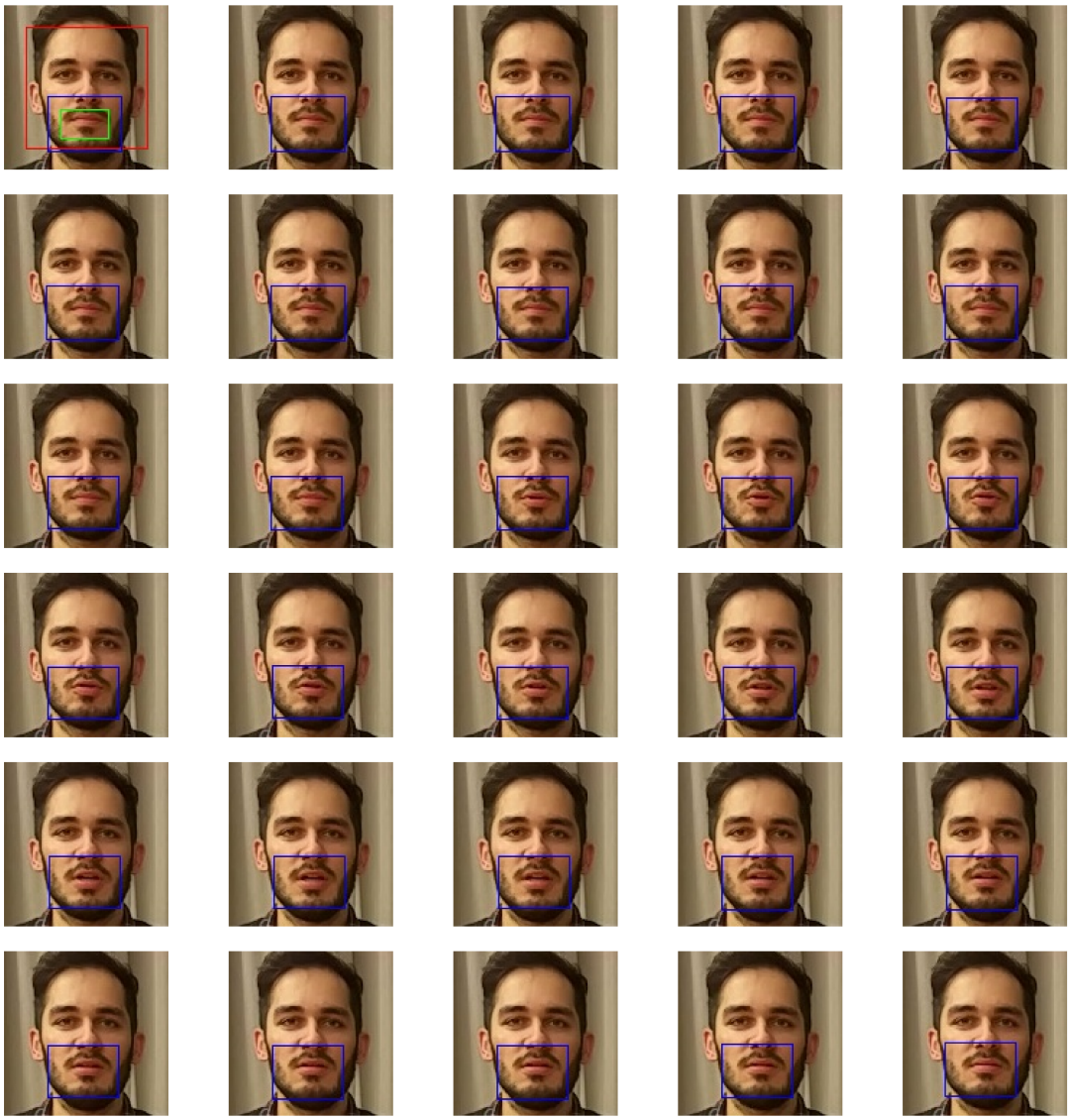


Figure 5.3: Detecting + Tracking mouth

Figure 5.4: Final result

The processed data was stored in hard disk after the original structure of DLIP. The length of frame sequence for a word is from 17 to 71 corresponding to 0.56 → 2.37 seconds video. Graph 5.5 shows the distribution of the length of the frame sequence (video) for each word measured by second.

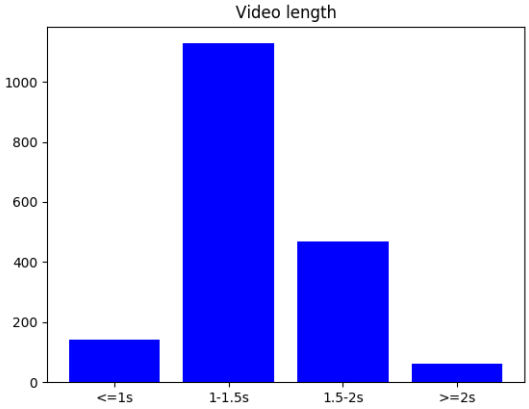


Figure 5.5: Word length statistic

2 datasets were then created:

- **seen dataset**: From each person, for each word, 2 data unit are used for testing, 2 for validation and the rest for training. By doing that, the validation/test set contains all words spoken by all recorded people with the same percentage. The dataset is called "seen" because all the people in the validation/test set are already seen during training. In conclusion, there are 15 people*12 words*6 units=1080 samples for training set, 360 for validation and 360 for testing.
- **unseen dataset**: Out of 15 people, 2 people are chosen for validation, 2 for testing and the rest for training. In summary, the train set contains 11 people*12 words*15

units = 1320 samples, validation set 240 and test set 240. This dataset is created to figure out how robust is the trained model on completely new people.

Be aware of the small dataset, the training data is augmented.

5.2.2 Data Augmentation

The data size is 304 MB in total. So "offline" Data Augmentation was used. It means augmenting the data offline and saving it all together. The same augmentation method was applied to all the frame from a frame sequence. They were chosen so that the lip movement of the word is still able to be recognized. Random cropping, shifting or rotating aren't good choices since they have the potential of cutting off a part of the lip or transforming it in a weird angle. Avoiding those problems, the following augmentation methods were chosen:

- flipping horizontally
- using Guassian blur
- adding salt and pepper noise
- equalizing histogram

```
def flip_image(image, output):
  flipped = cv2.flip(image, 1)
  cv2.imwrite(output, flipped) # save new data to hard disk
def gaussian_blur(image, output):
  blur = cv2.GaussianBlur(image, (3, 3), 0)
  cv2.imwrite(output, blur)
def salt_and_pepper(image, output):
  s_vs_p = 0.5
  amount = 0.02
  out = image
  # Salt mode
  num_salt = np.ceil(amount * image.size * s_vs_p)
  coords = [np.random.randint(0, i - 1, int(num_salt))
              for i in image.shape]
  out[coords] = 255
  # Pepper mode
  num_pepper = np.ceil(amount* image.size * (1. - s_vs_p))
  coords = [np.random.randint(0, i - 1, int(num_pepper))
              for i in image.shape]
  out[coords] = 0
  cv2.imwrite(output, out)
def equalize_histogram(image, output):
  eq = cv2.equalizeHist(image)
  cv2.imwrite(output, eq)
```

Flipping, blurring or histogram equalization are available functions in OpenCV. Gaussian blur was used to have a naturally blurred image in comparison to other blur methods such as average, median. An odd kernel kxk slides through the image from left to right, top to bottom and replaces the center pixel in this window by a weighted mean from the neighborhood pixels. The closer the pixel is, the more weight it has. The bigger the kernel k is, the more blurred the result image is. Therefore, a small k=3 is used to have a small blur effect. Equalizing histogram is a method to increase the contrast of an image

by “balancing” the distribution of pixels. Figure 5.6 visualizes an example of histogram equalization effect.

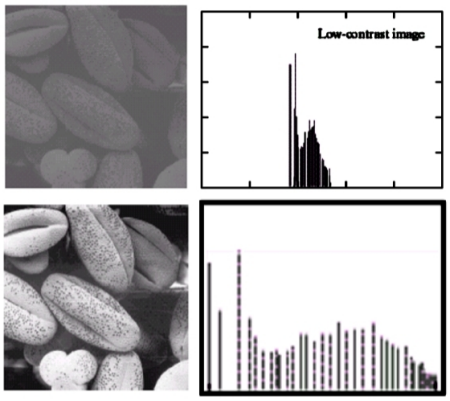
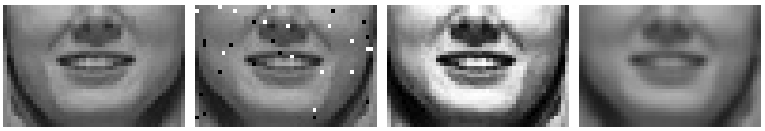


Figure 5.6: Histogram equalization [38]

Adding salt and pepper noise is adding black and white point to an image, means setting random pixel to 0 (black) or 255(white). In this work, an amount=2 % of salt and pepper was added to each image. The percentage and salt and pepper noise are equal (s_vs_p = 0.5). All the used augmentation methods remain the lip movements respectfully. Figure 5.7 visualizes an example of augmented images.

(a) origin



(b) horizontal flip (c) salt and pepper noise (d) histogram equalization (e) gaussian blur

Figure 5.7: Augmented result

5.2.3 Saving dataset in hdf5 files

Finally all data was saved in hdf5 files. It was done by 3 steps:

1. read_video_to_numpy(data): reading each frame sequence for a word to a numpy array of shape (frame length x frame width x frame height). This method returns a list of frame sequence x and a list of label y respectfully (y[i] is label of x[i]).

2. since the frame sequences have variable length, padding zero was used to have a consistent length of 71. In all samples with number of frame smaller than 71, zero frames were added at the end.

```
from keras.preprocessing import sequence
x_padded = sequence.pad_sequences(x, maxlen=71, padding='post')
```

3. The padded data was then written to hdf5 file.

```
def write_to_h5py(filename, train_x, train_y, val_x, val_y, test_x,
                                        test_y):
  file = h5py.File (filename,'w')
  file.create_dataset("train_x", dtype=np.dtype('uint8'), data =
                                        train_x)
  file.create_dataset("train_y", dtype=np.dtype('uint8'), data =
                                        train_y)
  file.create_dataset("val_x", dtype=np.dtype('uint8'), data =
                                        val_x)
  file.create_dataset("val_y", dtype=np.dtype('uint8'), data =
                                        val_y)
  file.create_dataset("test_x", dtype=np.dtype('uint8'), data =
                                        test_x)
  file.create_dataset("test_y", dtype=np.dtype('uint8'), data =
                                        test_y)
  file.close()
```

In conclusion, 4 .h5 files were created. Their statistic is shown in table 5.3

File name	Description	train set	validation set	test set
dlip_seen_no_aug.h5	seen dataset without Data Augmentation	1080	360	360
dlip_seen.h5	seen dataset with Data Augmentation	5400	360	360
dlip_unseen_no_aug.h5	unseen dataset without Data Augmentation	1320	240	240
dlip_unseen.h5	unseen dataset with Data Augmentation	6600	240	240

Table 5.3: Table of created datasets in hdf5 files

5.3 Hardware setup

OS : Xubuntu 18.04

GPU : **NVIDIA GeForce GTX 1080**

CPU: Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz 4 cores 8 threads

RAM: 16GiB

5.4 Models

5.4.1 Choices of architecture

Again, the task is to predict the word correctly from a sequence of gray frames. In order to success it, the chosen model need to learn how the lips look like in each frame and how

they change through all the frames. Technically it should be able to capture spatial as well as temporal features from the frame sequence. Taking into consideration the requirement of this task, as well as the character of different neural networks, 3 potential architectures were chosen:

- C3D: CNN is popular for its good performance at extracting features in space. Instead of using a 2D kernel to figure out only the relationship among pixels in a frame, C3D slides a 3D kernel through the whole frame sequence and as a result is able to capture also some features in time.

- LSTM: RNN is common for extracting temporal features. LSTM is one of the RNN, which is specially designed to hold long-term dependencies. In this task, LSTM could be the best solution to remember how the lips move in time. Unfortunately, it is not clear how good it could extract the spatial features in each frame. As it sees each frame as an input for a timestep, the author believes it will also be able to figure out some relationship among pixels in each frame.

- LRCN: a hybrid NN, which is combined by CNN and LSTM layers, is as the author's opinion the best architecture for this task. Using some CNN layers firstly to capture the most important features in each frame and then using LSTM layers to remember the change in time. By doing this, both spatial and temporal features are extracted respectfully.

In many cases, Transfer Learning has proven its effectiveness on extracting spatial features. Thus, it is not implemented in this task due to the following reasons:

- Most popular pre-trained networks are trained on quite big and colorful image (e.g. VGG16 224x224x3). In Keras, it is able to apply those networks on smaller image. However, the author believes that those complex networks are not necessary for this task.

- In addition, the final frame is very informative. A couple of CONV layers should be capable of learning enough features in space. Plus the training process will be much faster due to the small set of parameters.

5.4.2 Training strategy

Models of 3 chosen architectures were trained on seen datasets separately. Out of each architecture, a best trained model was chosen for final evaluation. Out of 3 best models, the final one was tested one last time on unseen dataset. During training, accuracy and loss function were used as evaluation metrics. The computing time served as satisfying metric and was observed and noticed only. Cross-validation was not used because of limitation of computing and to ensure that each data set contains the wanted data. A random split of all data into different folds in cross-validation would make the result less reliable. A fixed split of train, test, validation set is provided inclusive in the hdf5 files.

General setup: All models were built with Keras Sequential Model. The output layer used softmax as activation function and contains 12 neurons. The input was always normalized to [0,1], the output was one hot vector [2]. Train and validation set in hdf5 files were used during training. Through some experiments, Adam was chosen as optimizer.

[2] a vector which is filled in with all 0s and only one 1

General techniques: Firstly, some random base models and some models from similar problems were tested. Secondly, Gridsearch was used for tuning hyperparameter. Gridsearch is a widely used hyperparameter optimization technique, which evaluates each model on a combination of given parameters. It then reports the accuracy and loss of all possible combination. In this work, GridSearchCV class from scikit-learn library was used. Keras model can be wrapped in scikit-learn by KerasClassifier class. GridSearchCV then constructs and evaluates each model with default 3-fold cross-validation. Below is a simple example of Gridsearch for hyperparameter droprate

```
def build_model_base(droprate=0.0):
  ...
model = KerasClassifier(build_fn=build_model_base, epochs=...,batch_size
                                        =..., verbose=0)
droprate=[0.1, 0.2, 0.3, 0.5]
param_grid = dict(droprate=droprate)
grid = GridSearchCV(estimator=model, param_grid=param_grid, cv=5)
grid_result = grid.fit(X, Y)
```

For each type of architecture, different hyperparameters were tuned. Based on the course of the loss and accuracy, it is to identify if the model is in overfitting or underfitting area and corresponding the methods to apply for each direction as mentioned in chapter 2.

- CONV layers: kernel size, number of filters, learning rate, regularization (dropout+l2), initialization methods etc.
- Recurrent layers: learning rate, number of hidden units and especially regularization since LSTM usually overfits very easily. There are 2 kinds of way to drop neurons in RNN: input dropout and recurrent dropout. Figure 5.8 visualizes the different between them.

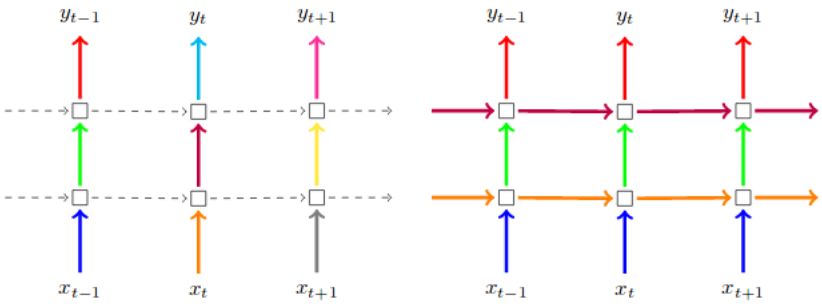


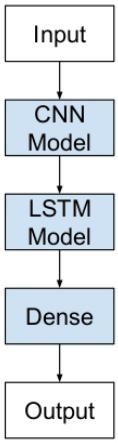
Figure 5.8: Difference between input and recurrent dropout [23]. Left: Naive dropout (input dropout), Right: Variational dropout (input dropout + recurrent dropout), Colored connections represents dropped-out input, different colors show different dropout masks. Dashed lines correspond to standard connections without dropout

Ignoring the color, it is easy to see that on the left side dropout is only applied to vertical connection while in the right side to both directions. Input dropout, also known as naive dropout, showed on the left side is applied to the input or output in each timestep (vertical connections), from x_t to y_{t-1}. This type of dropout is exactly like dropout in a fully connected layer, repeated number of timesteps times.

Recurrent dropout is on the other hand applied on the recurrent connections from timestep to timestep (horizontal connections). [23] also pointed out the effect of keeping the same dropout mask at each timestep, including the recurrent layers. This technique is called Variational dropout.

An example of step-by-step training process of LRCN is shown in the following. Parallel is the evaluation of some hyperparameter tuning techniques. The step-by-step training process of C3D and LSTM is in some manner similar and can be skipped.

5.4.3 Hyperparameter tuning with LRCN

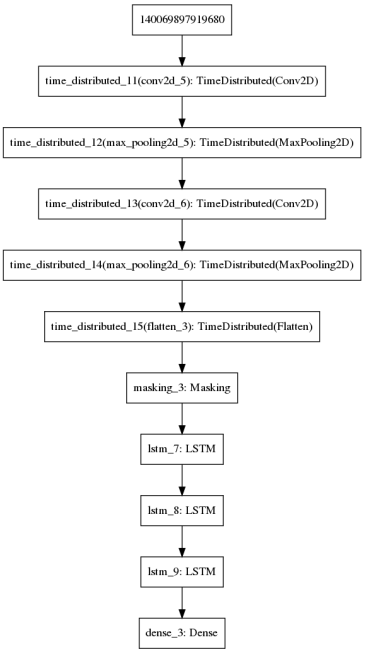


(a) lrcn

Layer (type)	Output Shape	Param #
time_distributed_11 (TimeDis	(None, 71, 30, 46, 16)	160
time_distributed_12 (TimeDis	(None, 71, 15, 23, 16)	0
time_distributed_13 (TimeDis	(None, 71, 13, 21, 16)	2320
time_distributed_14 (TimeDis	(None, 71, 6, 10, 16)	0
time_distributed_15 (TimeDis	(None, 71, 960)	0
masking_3 (Masking)	(None, 71, 960)	0
lstm_7 (LSTM)	(None, 71, 256)	1246208
lstm_8 (LSTM)	(None, 71, 128)	197120
lstm_9 (LSTM)	(None, 128)	131584
dense_3 (Dense)	(None, 12)	1548

Total params: 1,578,940
Trainable params: 1,578,940
Non-trainable params: 0

(b) model summary



(c) model

Figure 5.9: LRCN baseline

Figure 5.10 shows the baseline model for training. In general LRCN contains CNN model as front end, followed by LSTM model and finally Dense layers (Fully connected layers). The baseline model contains 2 CONV 2D layers, each followed by a MAX POOLING. Those layers can only process an image at a time and output a representation vector (frame features). In order to pass the representation vector of each frame as an input for a timestep in LSTM, those layers need to be wrapped in TimeDistributed layers. Following are 3 LSTM layers with 256, 128 and 128 hidden units. As output layer, a Dense layer with activation function softmax of 12 classes was used. Some parameters were set up as following:
Optimizer: Adam
Learning rate $\mu = 0.0005$
Batch size = 128
Epoch = 100
Shuffle data during training shuffle=True
EarlyStopping based on training accuracy with patience=15
Using callback to save training history

Using ModelCheckPoint to save best model based on loss of validation set

Many experiments showed that the model could not learn with a bigger learning rate. A smaller learning rate was fine but slowed down the training process. Early Stopping is a technique to pretend the model from overfitting. It is sometimes considered as a regularization method. The model should be stopped when the training is going to the unwanted direction when the training accuracy increases but validation accuracy starts to decrease.

The training process contains 5 steps
1. Baseline
2. Applying Data Augmentation
3. Apply Batch Normalization
4. Apply L2
5. Apply Dropout

1. Training the baseline model on the seen dataset without Data Augmentation dlip_seen_no_aug.h5 showed a very hard overfitting. The train set accuracy increased and the train loss decreased continuously but quite slow. The validation set accuracy increased at the beginning and then stayed around 0.5 no matter how long the model was kept training. It seems like the model can learn well from train data, but is not able to generalize since the train and validation accuracy is 100% and 50%. With 12 classes, a random baseline for this classifier is 8.33%. This baseline model is therefore over 5 times better than random choice metric

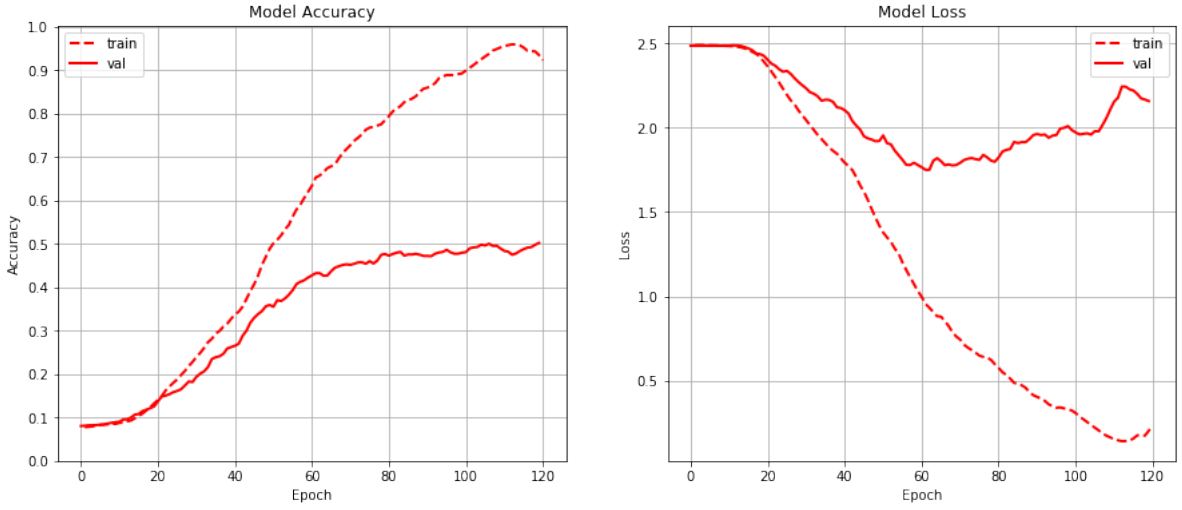


Figure 5.10: Baseline Accuracy and Loss

A remarkable point is that while the validation accuracy stays, the validation loss tends to increase (epoch 60 to 100). A simple possible explanation for this phenomenon is because of the softmax distribution. For example: Sample s belongs to class C. In the early epochs, it is predicted as class C with the probability of 0.9. Unfortunately in the later epochs, the classifier for this class is getting worse and predicts s as C but with the probability of only 0.55. In this case, the accuracy stays but the loss will increase. This could happen at some of the validation data and resulted the unexpected increase of loss

as seen in figure 5.10

2. Since the origin dataset is quite small, an overfitting in baseline was no surprise. To fight against it, Data Augmentation was applied firstly. The same model was then trained on the seen dataset with Data Augmentation dlip_seen.h5 . Data Augmentation proved an significant improvement in both train and validation set. The accuracy plot suggests a huge speed up in the training process. The train accuracy achieved 100% around 60 epochs while without augmentation around 120 epochs. It seems like the model needs to see enough data to accelerate the learning process, as well as generalize better. Still it shows a very big overfitting. Both train and validation accuracy reached their maximum very early and then stayed stable at 1.0 and 0.65. It suggests that the capacity for learning of the model fulfills, for generalization not yet. More regularization methods need to be applied

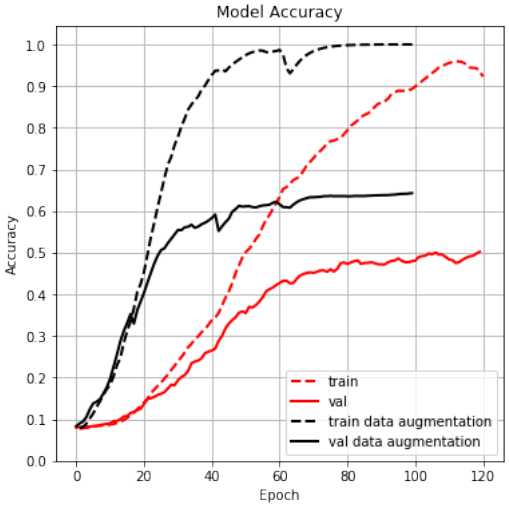


Figure 5.11: Data Augmentation Effect

3. Batch Normalization was used in CONV layers firstly. It was to expected that the training would be faster and the generalization would be better due to the fact that Batch Normalization produces a more stable distribution of input to each layers. An increasing in speed was acknowledged while Batch Normalization surprisingly hurt the validation performance. The validation accuracy dropped almost 10% in general in comparison to the baseline. There is no firm explanation for this phenomenon but the author suspects that Batch Normalization unfortunately mess up the distribution of representation in this case. The default representation values without normalization, which are not in the same scale, might be the correct one in order to emphasize the area of lips (big values) and the area without lips (very small values). Batch Normalization might by default mix those values to have a balance distribution and consequently drives the learning in the wrong path, focusing in the wrong area. Since accuracy was the primary metric and speed was only satisfying metric, Batch Normalization was removed for all further steps.

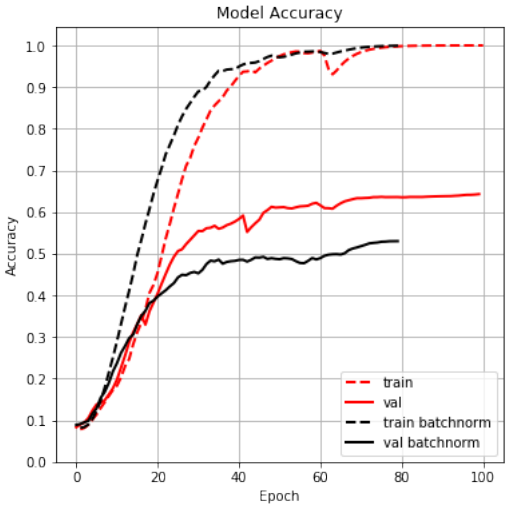


Figure 5.12: Batch Normalization Effect

4. L2 was the next chosen regularization method. L2 is a classic technique, which assigns a penalty on weights to keep it small, almost close to zero, results a smaller network.

Gridsearch unfortunately showed that applying L2 or even L1 in LSTM outcomes very poor results. Testing some values of l2 from 0.001 to 0.2 on the whole dataset strengthened the result from gridsearch.

```
Best: 0.082031 using {'l2': 0.06}
0.019531 (0.027460) with: {'l2': 0.0001}
0.078125 (0.044718) with: {'l2': 0.001}
0.054688 (0.020130) with: {'l2': 0.01}
0.062500 (0.036146) with: {'l2': 0.02}
0.078125 (0.039785) with: {'l2': 0.04}
0.082031 (0.043844) with: {'l2': 0.06}
0.070313 (0.044002) with: {'l2': 0.1}
0.066406 (0.027833) with: {'l2': 0.2}
0.074219 (0.038686) with: {'l2': 0.5}
```

Figure 5.13: Gridsearch result L2

5. One of the most powerful regularization technique nowadays is dropout. It produced indeed a great improvement in validation accuracy. By applying a 0.2 dropout probability, it shrank the gap of overfitting from 35 to 20% as shown in figure 5.14. The validation accuracy reached over 75% and its loss decreased continuously.

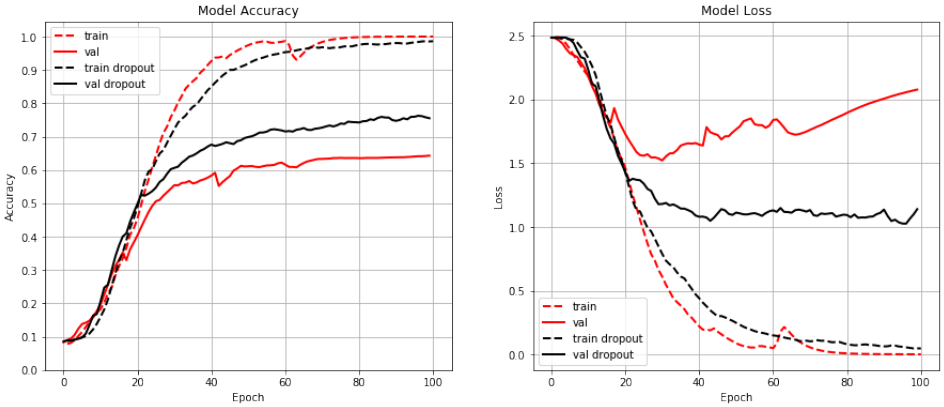


Figure 5.14: Dropout Effect

It is a proof that the model is getting better at generalizing and suggests a bigger dropout will lift the accuracy and loss scores. Figure 5.15 visualizes the result of increasing dropout probability to 0.5, 0.7 .

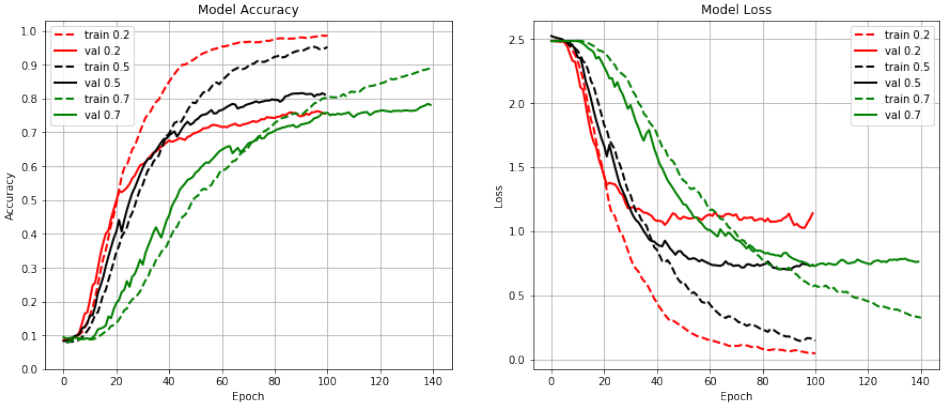


Figure 5.15: Comparison of different dropout probabilities

By looking at the dashed lines in Model Accuracy, it is easy to see that the more regularization is applied, the slower the training process is. Sacrificing the speed for accuracy is acceptable. Indeed, by increasing dropout to 0.5, the black normal line is 0.5 over the red line of 0.2 dropout and the model reached an validation accuracy of over 80%, reduces significantly the gap of overfitting (see the distance of red dashed and normal lines and of black dashed and normal lines). Keep on increasing the dropout probability to brutal 0.7, the training process slowed down fast (see the dashed green line is way far under the dashed red and black lines). Unfortunately it didn't improve the validation accuracy either. A possible explanation for this behavior is that dropping too many neurons might reduce the learning capacity of the model, cause missing important features. The loss on the right side shows the continuous decreasing, corresponds to the increase of accuracy on the left side. In conclusion, 0.5 was the best dropout probability in this case.

The dropout, which was applied and compared above is only input dropout, implemented in Keras by parameter "dropout". Another kind of dropout is dropping the unit on the recurrent connection, used as "recurrent_dropout". Apply the same value of 0.5 for recurrent dropout in all LSTM layers, figure 5.16 shows its effectiveness in comparison to input dropout.

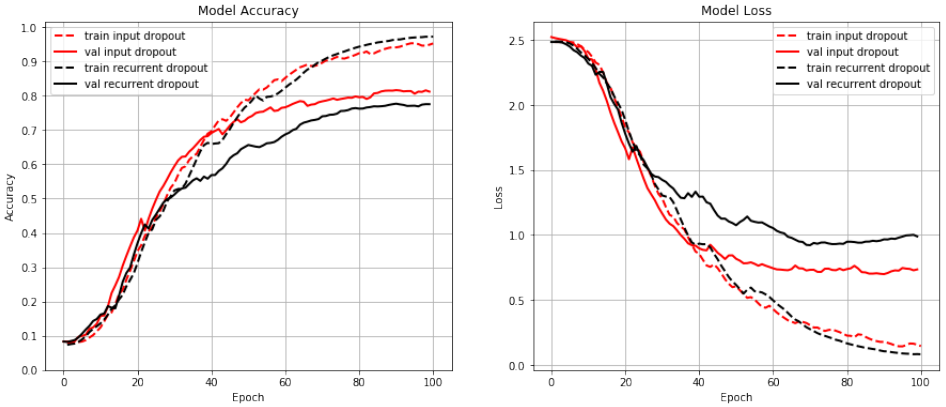


Figure 5.16: Comparison of input and recurrent dropout

In both Model Accuracy and Model Loss plot, it is easy to notice that validation accuracy of model with input dropout is clearly better and the loss is clearly smaller than with recurrent dropout (see the normal lines). It proves that input dropout has better impact on the capacity of generalization of the model in this case. One last experiment which was executed is to combine two kind of dropout together. Indeed, it ended up with the best LRCN model. Among different tested combinations, a 0.3 input dropout probability plus 0.4 recurrent dropout probability gave the best result. The model loss of validation set was clearly smaller than the best tested model so far.

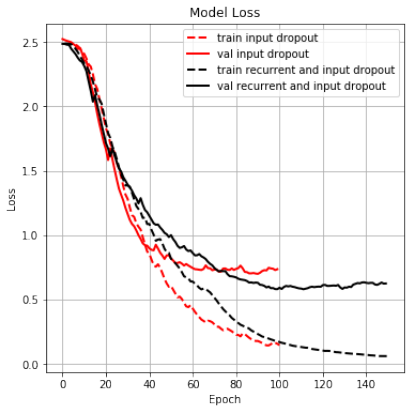


Figure 5.17: Combination of input and recurrent dropout

Table 5.4 summarizes again the whole training process of LRCN with some representatives for each technique.

Model Nr.	Data Augmentation	Regularization	Epochs	Train Accuracy	Validation Accuracy
1	No	No	#120	90	**52**
2	Yes	No	#100	100	**64.72**
3	Yes	Batchnormalization	#80	100	**53.06**
4	Yes	L2	-	-	-
5.1	Yes	input dropout 20%	#100	98.2	**74.44**
5.2	Yes	input dropout 50%	#100	96.74	**83.33**
5.3	Yes	input dropout 70%	#140	89.59	**79.17**
5.4	Yes	recurrent dropout 50%	#100	97.43	**77.5**
5.5	Yes	input dropout 30% + recurrent dropout 40%	#150	98.06	**86.11**

Table 5.4: LRCN training summary

5.5 Evaluation

5.5.1 Comparison among different architectures

Among all tested models from 3 chosen architectures, a best model of each kind was picked and evaluated one last time on the test set in hdf5 files.

Arch	Parameters	Time/epochs	Epochs	Test accuracy
LRCN	#1,578,940	20s	#150	**88.61%**
LSTM	#5,512,204	14s	#68	66.38%
C3D	#145,884	18s	#70	65.28%

Table 5.5: Best models

C3D: It's very hard to find a baseline for this task since C3D model mapped the training data very fast but it learned features by heart and performed the generalization very poorly. Figure 5.18 shows some setups of different number of filters. No matter how big the number of filters is, the model maps the train data perfectly very soon but the validation accuracy is quite bad and does not really improve through time.

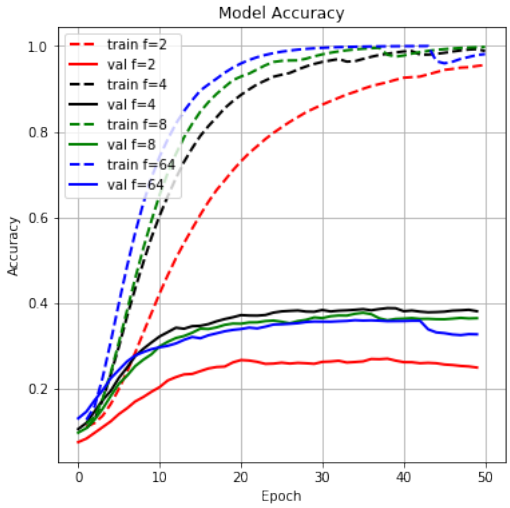


Figure 5.18: C3D with different filters setup

The final model of this kind used a brutal dropout of 0.7. Still the course of the training showed a very large overfitting, which the author found very difficult to reduce. Experiments also showed that a bigger filter kernel can hold more features in time and as a result has higher accuracy score. The final model used a kernel size of 7x7x7 for all CONV layers.

Layer (type)	Output Shape	Param #
conv3d_1 (Conv3D)	(None, 65, 26, 42, 16)	5504
batch_normalization_1 (Batch	(None, 65, 26, 42, 16)	64
max_pooling3d_1 (MaxPooling3	(None, 32, 13, 21, 16)	0
dropout_1 (Dropout)	(None, 32, 13, 21, 16)	0
conv3d_2 (Conv3D)	(None, 26, 7, 15, 16)	87824
batch_normalization_2 (Batch	(None, 26, 7, 15, 16)	64
max_pooling3d_2 (MaxPooling3	(None, 13, 3, 7, 16)	0
dropout_2 (Dropout)	(None, 13, 3, 7, 16)	0
flatten_1 (Flatten)	(None, 4368)	0
dense_1 (Dense)	(None, 12)	52428

Total params: 145,884
Trainable params: 145,820
Non-trainable params: 64

Figure 5.19: C3D final model summary

Many regularization methods such as l1, l2, dropout etc. were applied but did not improve the performance much. C3D seems not capable of capturing enough features to give confident predictions. In many cases where the model labeled correctly, the confident score stayed quite low. For example:
vergangen ['0.34', '0.37', '0.40', '0.41', '0.47', '0.55', '0.69', '0.84', '0.87', '1.00']

werden ['0.30', '0.34', '0.35', '0.35', '0.38', '0.54', '0.59', '0.60', '0.60', '0.63', '0.69', '0.79', '0.79', '0.81', '0.86', '0.99']
Probably, the model needs to see more data to figure out the correct features to learn. An unnormalized confusion matrix was constructed. For each class, there are 30 samples. The diagonal line shows the number of correctly labeled samples. The bigger the number is, the stronger the color blue is. It is to see that class "vergangen" has the poorest result with an accuracy of about 33% and class "hoch" has the best performance with an accuracy of over 80%. It strengthens the suspect that C3D is not good at holding information in a long time since "vergangen" has in general the longest sequences of frame and "hoch" has the shortest one.

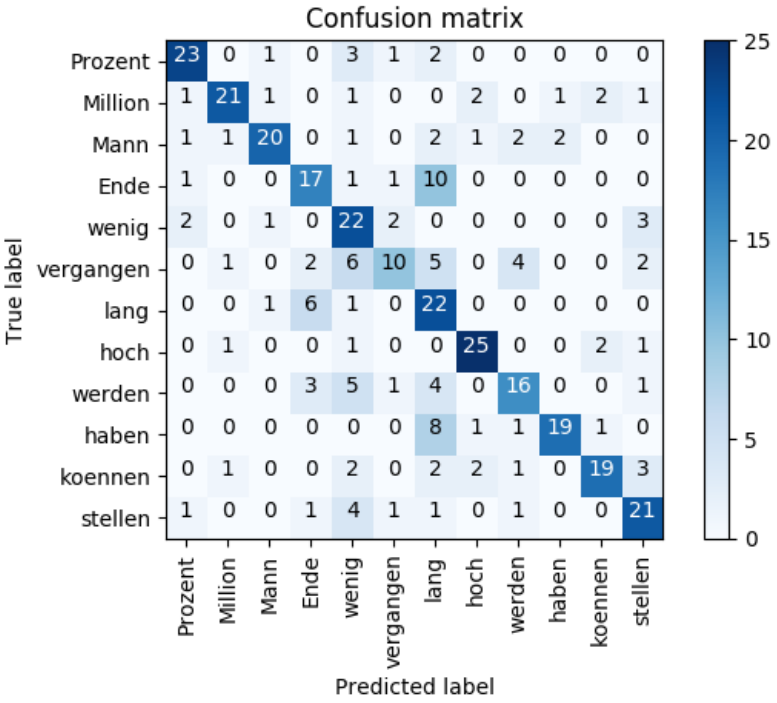


Figure 5.20: C3D Confusion Matrix

LSTM: According to the observation during training, LSTM models were able to learn features well but quite slow. It seems like it is quite hard for LSTM models to figure out the right features to learn in this task because of its disadvantage at extracting spatial features. The final LSTM model had a very big set of parameters, over 5 millions.

Layer (type)	Output Shape	Param #
masking_6 (Masking)	(None, 71, 1536)	0
lstm_16 (LSTM)	(None, 71, 512)	4196352
lstm_17 (LSTM)	(None, 71, 256)	787456
lstm_18 (LSTM)	(None, 256)	525312
dense_7 (Dense)	(None, 12)	3084

Total params: 5,512,204
Trainable params: 5,512,204
Non-trainable params: 0

Figure 5.21: LSTM final model summary

The train accuracy reached almost 100% very early but the validation accuracy stayed with a great distance, achieved only 70%. In most cases where the model predicted correctly, the scores were pretty high. For example:
Prozent ['0.57', '0.66', '0.78', '0.88', '0.89', '0.93', '0.95', '0.96', '0.97', '0.98', '0.98', '0.98', '0.98', '0.99', '0.99', '0.99', '0.99', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00']
Million ['0.38', '0.49', '0.92', '0.94', '0.95', '0.99', '0.99', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00']
Like C3D, the author found it very hard to regularize the network. A best regularization of 0.2 dropout probability shrank 10% the overfitting gap in comparison with no regularization at all.

LRCN: provided the best trained model as expected. The final model of this kind achieved a train accuracy of almost 100%, a validation accuracy of 86% and a test accuracy of 88%. The confusion matrix shows a very good result in general. All classes reaches very high accuracy in comparison to the best LSTM and C3D models.

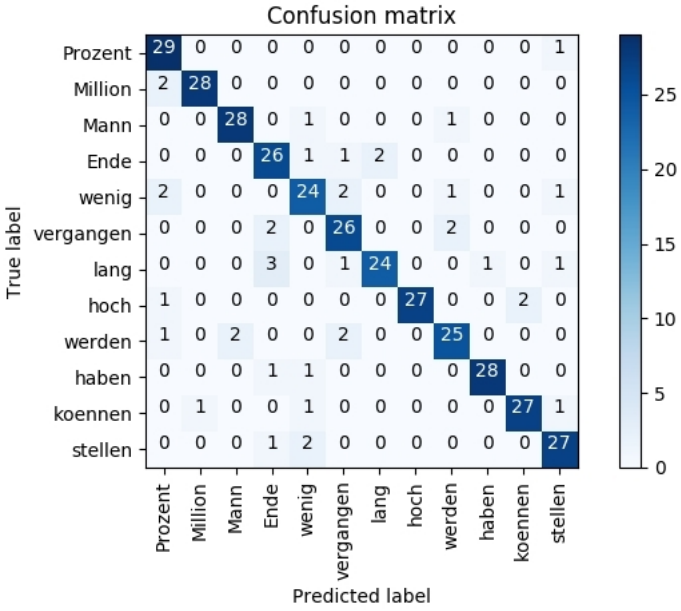


Figure 5.22: LRCN Confusion Matrix

In most correctly labeled samples, the model showed a very high confidence at its predictions showing that it captured the important features and could generalize the pattern

of this task very well. Proof is:
Prozent ['0.97', '0.98', '0.99', '1.00']
Million ['0.99', '1.00']
Mann ['0.99', '0.99', '0.99', '1.00']
haben ['0.97', '1.00']

Still there are 41 out of 360 mislabeled data. The following error analysis will give a better overview of all data, which were predicted wrongly.

Index	Real	Predict	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
168	1. Prozent	12. stellen	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.66
234	10. haben	4. Ende	0.00	0.00	0.00	0.49	0.00	0.00	0.02	0.00	0.00	0.48	0.00	0.00
330	10. haben	5. wenig	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.00	0.02	0.00	0.00	0.00
44	11. koennen	12. stellen	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.35	0.65
45	11. koennen	2. Million	0.00	0.89	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.00	0.00
333	11. koennen	5. wenig	0.00	0.00	0.00	0.00	0.48	0.45	0.00	0.00	0.03	0.00	0.00	0.04
46	12. stellen	5. wenig	0.00	0.00	0.00	0.00	0.51	0.00	0.00	0.00	0.00	0.00	0.00	0.49
142	12. stellen	4. Ende	0.00	0.00	0.00	0.97	0.00	0.00	0.02	0.00	0.01	0.00	0.00	0.00
239	12. stellen	5. wenig	0.00	0.00	0.00	0.00	0.59	0.20	0.00	0.00	0.01	0.00	0.00	0.20
51	2. Million	1. Prozent	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
122	2. Million	1. Prozent	0.95	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
197	3. Mann	5. wenig	0.00	0.00	0.30	0.00	0.40	0.03	0.00	0.00	0.27	0.00	0.00	0.00
316	3. Mann	9. werden	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99	0.00	0.00	0.00
102	4. Ende	5. wenig	0.00	0.00	0.00	0.06	0.90	0.00	0.01	0.00	0.00	0.00	0.00	0.02
103	4. Ende	6. vergangen	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.01	0.00	0.00	0.00
175	4. Ende	7. lang	0.00	0.00	0.00	0.46	0.00	0.00	0.53	0.00	0.00	0.00	0.00	0.00
198	4. Ende	7. lang	0.00	0.00	0.00	0.37	0.00	0.00	0.63	0.00	0.00	0.00	0.00	0.00
33	5. wenig	1. Prozent	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
56	5. wenig	12. stellen	0.10	0.00	0.00	0.00	0.18	0.13	0.00	0.00	0.03	0.00	0.00	0.56
176	5. wenig	1. Prozent	0.60	0.00	0.01	0.00	0.38	0.00	0.00	0.00	0.00	0.00	0.00	0.00
249	5. wenig	6. vergangen	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00
297	5. wenig	6. vergangen	0.00	0.00	0.00	0.00	0.02	0.97	0.00	0.00	0.01	0.00	0.00	0.00
320	5. wenig	9. werden	0.00	0.00	0.00	0.00	0.12	0.01	0.00	0.00	0.88	0.00	0.00	0.00
34	6. vergangen	9. werden	0.00	0.00	0.00	0.00	0.00	0.07	0.00	0.00	0.92	0.00	0.00	0.00
106	6. vergangen	4. Ende	0.00	0.00	0.00	0.68	0.00	0.30	0.01	0.00	0.00	0.00	0.00	0.00
251	6. vergangen	9. werden	0.00	0.00	0.00	0.00	0.00	0.18	0.00	0.00	0.81	0.00	0.00	0.00
274	6. vergangen	4. Ende	0.00	0.00	0.00	0.52	0.00	0.44	0.03	0.00	0.00	0.00	0.00	0.00
61	7. lang	12. stellen	0.00	0.00	0.00	0.36	0.10	0.00	0.05	0.00	0.00	0.00	0.00	0.47
205	7. lang	4. Ende	0.00	0.00	0.00	0.91	0.00	0.00	0.09	0.00	0.00	0.00	0.00	0.00
252	7. lang	4. Ende	0.00	0.00	0.00	0.83	0.00	0.00	0.17	0.00	0.00	0.00	0.00	0.00
253	7. lang	4. Ende	0.00	0.00	0.00	0.95	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.00
300	7. lang	10. haben	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.99	0.00	0.00
349	7. lang	6. vergangen	0.00	0.00	0.00	0.01	0.00	0.78	0.21	0.00	0.00	0.00	0.00	0.00
110	8. hoch	11. koennen	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.12	0.00	0.00	0.86	0.00
182	8. hoch	1. Prozent	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
206	8. hoch	11. koennen	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.46	0.00	0.00	0.54	0.00
64	9. werden	1. Prozent	0.93	0.00	0.00	0.00	0.06	0.00	0.00	0.00	0.00	0.00	0.00	0.00
89	9. werden	3. Mann	0.00	0.00	0.90	0.02	0.00	0.01	0.00	0.00	0.07	0.00	0.00	0.00
112	9. werden	3. Mann	0.00	0.00	0.51	0.01	0.00	0.01	0.00	0.00	0.47	0.00	0.00	0.00
232	9. werden	6. vergangen	0.00	0.00	0.00	0.00	0.00	0.98	0.00	0.00	0.01	0.00	0.00	0.00
328	9. werden	6. vergangen	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 5.23: LRCN Error Analysis

The table of mislabeled samples 5.23 describes how the model predicts each sample into all classes. The color green marks the probability of the correct label and the label which is wrongly predicted. The red color marks all other classes, which have also a probability of bigger than 0.01. Clearly, there are very small amount of red number. It points out that in most cases, the model was only confused between 2 classes, which is

probably caused by the similarity in the lip position, movement or the length of the frame sequence. In those cases, the model also gave the correct label a quite big probability showing that it did learn some correct features which drove the model into that prediction. For example: :
1. Prozent (168) 0.33 labeled as 12. stellen 0.66
10. haben (234) 0.48 labeled as 4. Ende 0.49
12. stellen (46) 0.49 labeled as 5. wenig 0.51
4. Ende (175) 0.46 labeled as 7. lang 0.53
6. vergangen (274) 0.44 labeled as 4. Ende 0.52
In some other cases where the model was quite confident with the wrong predictions, the samples were checked individually showing that it was almost impossible for human to predict correctly as well. For example 8. hoch (182), the recorded person has a big beard and did not move the lips at all while speaking "hoch" or 11. können (45) looks like an unclean data or the recorded person moved too fast from one direction to another at 9. werden (64). Other samples showed also very untypical lip movements for the spoken words.

5.5.2 Unseen dataset

The best model was as mentioned above tested on unseen dataset. The result was clearly worse than on seen dataset. The validation and test accuracy dropped from almost 90% to over 50%. In the test set of unseen dataset, there are all data from 2 people, corresponding 20 samples per class. The confusion matrix suggests a worse prediction in general.

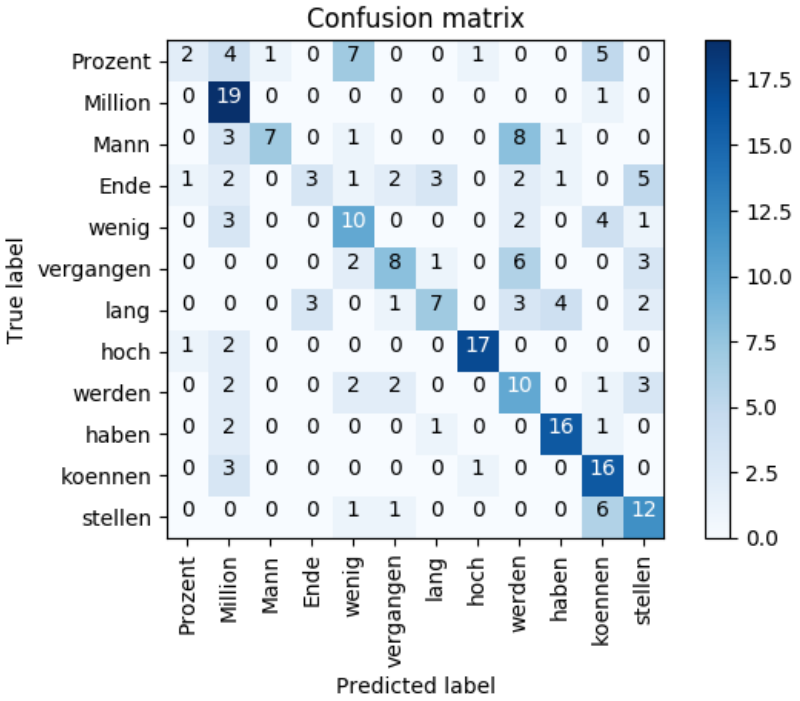


Figure 5.24: Confusion Matrix for unseen dataset

Among all, only 4 classes achieved good results: 2. Million (19/20), 8. hoch (17/20), 10. haben (16/20), 11. können (16/20). The other scored very poorly. It seems like the model only focused on learning some specific words. It predicted haft of the test data

into those 4 classes and its confidence at predicting those classes is also surely higher. For example:
hoch ['0.70', '0.98', '0.98', '0.98', '0.98', '0.99', '0.99', '0.99', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00', '1.00']
werden ['0.36', '0.50', '0.52', '0.56', '0.63', '0.68', '0.68', '0.86', '0.87', '0.91']

Further analysis in figure 5.25 showed that more samples from the first person was predicted wrongly than the other person. Probably because the first person has a big beard, spoke untypically and moved very fast and much during recording. Especially class 5. wenig where person 1 scored 1/10 while person 2 scored 9/10. Checking the data from person 1 showed that he moved his face from one side to another very often while speaking 5. wenig. Surprise was the poor results from class 1. Prozent and class 4. Mann which were very good at seen dataset.

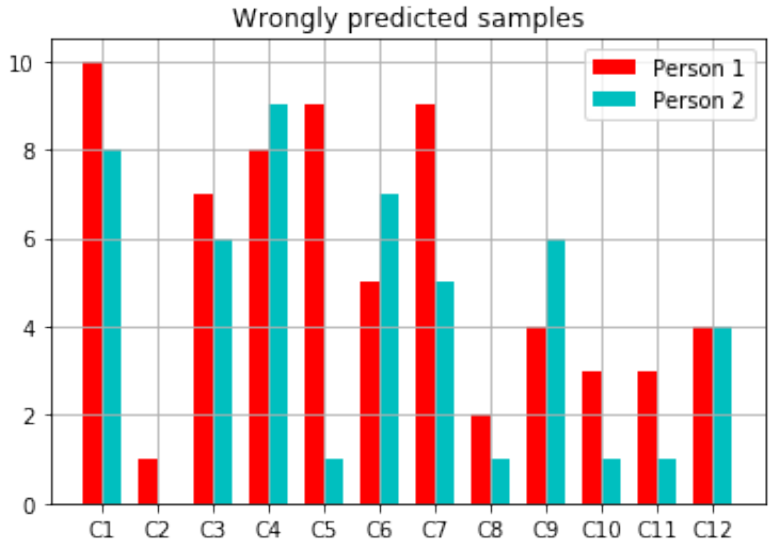


Figure 5.25: Error Analysis for unseen dataset

It seems very hard for the model to learn from some people and give prediction on some other people. With the small set of data, the model could probably only learn some easy to recognize words and fails right away at some anomalies in position of lips and speed of speaking.

Chapter 6

Summary

6.1 Conclusion & future work

The word classification among 12 classes has a random metric of 8,3%. The work shows that all best trained models of the 3 chosen architectures achieved assuredly better result than a random metric. Especially LRCN, with its advantage of extracting spatiotemporal features, has proven its outstanding result with an accuracy of 88% on seen dataset. Further analysis on mislabeled data shows that the model often fails on untypical lip movements of spoken words. It could be lips covered by beard, the small lip movements, the speed of opening lips or the direction of face. By that, it points out the limitation of even the best model in real application where more and more barrier for LR could occur since human is individual and has very different mouth shapes and speaking habits. Another proof of the impact of the speaking habit on LR is further proved in the result of unseen dataset where the model had to learn from some people and give prediction on other people. The model seems to capture the way the people in train set speak and fails a lot in the test set where the people have a different way of speaking some specific words. A solution for this could probably be the data. Giving a bigger dataset, with more data of different ways of speaking, different speed of speaking, the author believes it will feasibly improve the performance.

Through the training process of different models, the author found it extensively difficult to avoid overfitting. The author also acknowledged the effectiveness of Data Augmentation and Dropout at fighting against overfitting. Training the same best model of seen dataset on unseen dataset resulted in a notably worse accuracy. It suggests a different set of parameters for training unseen dataset. While randomness is the beauty of DL, the author met some difficulties in evaluating the model during training because of the unstableness of the model. Reproducing the exactly same result is almost impossible due to the degree of randomness used in Tensorflow as well as cuDNN the GPU-accelerated library, which were used for the implementation. To ensure the training flowed in the right direction, for each step of tuning hyperparameter, the author had to train many times to have a better and more secure view of the tuning effect.

In conclusion, machine LR is robust on the tested German words. The data plays an essential role. Therefore, it needs to be collected and preprocessed very carefully. Besides, applying regularization methods with the right amount is not less important. Deep Learning is the right choice for this task and LRCN is the best fit architecture. The author also believes that with a bigger dataset, perhaps collected from German talkshow or television, building a model, which could classify a large German vocabulary, is not

an unthinkable mission. As a consequence, it would be the base step for building an automatic LR systems, which can be applied in real life.

As final words, the author would like to share some opinions on the further development of LR.

6.2 Real-time and sentence-level Lip Reading

The main final goal of Lip Reading is to understand conversations, news etc. and for the best in real-time. They are mostly continuous sequences of sentences, each is again continuous sequence of words/phrases. In order to use Word Level Lip Reading model, all words need to be splitted seperately. This task is almost impossible since many words are usually spoken together and the lips move from one word to another so fast that it is unbelievably difficult to identify the stop of one word. One possible solution for this is to apply the model on a window of frames. The window will be expanded until the model can recognize a word with a high enough confidence. It is then moved to the next frame sequence. The base element of a conversation is actually a sentence. In many cases, the end of a sentence or some sentences is marked as the long stop of lip movements. Splitting the conversation into sentences or small conversations and then applying Sentence Level Lip Reading is as the author's opinion more potential. To the best of the author's knowledge, LipNet [6] is the first model, which does end-to-end sentence-based LR. It was trained and tested on the GRID corpus, which contains videos of 34 speakers with 1000 sentences each, gives a total dataset of 34000 sentences in 28 hours. LipNet can map variable-length sequences of video frames to text sequence with an accuracy of 95,2% on the GRID dataset. LipNet is built from 3 layers of spatiotemporal CNN (STCNN) [1], followed by bi-directional gated recurrent units (Bi-GRU), and finally a connectionist temporal classification (CTC) which is responsible for generating the output text sequence [6]. The similar architecture can be adapted for German Sentence/Short Conversation Level Lip Reading. Another challenge of LR in many languages is that there are many words/phrases which is not distinguishable in the lip movements. In order to give correct prediction, the context is needed. With the auto completion as an example and NLP in general is in a fast track of development, it can be very helpful for the LR model at predicting confusing words/phrases. For example: 2 German words "gejagt" (hunt) and "gesagt" (say) have the same lip movements. In such a sentence "Darüber hast du kein Wort ..." (you did not ... a word about it), the auto completion can tell the LR model clearly that the next word is "gesagt".

There are 2 points about real-time application for LR. Firstly it requires good hardware to be able to process the vastly big model when it comes to real large vocabulary. FPGA and ASIC are already in trend for DL application and will be certainly interesting for LR application as well. Secondly with variety and the continuous changing of the languages, collecting data in real-time and retraining the model respectfully might be very important as well.

[1] 3 3D CNN layers, followed by MAX POOLING layers which are wrapped into TimeDistributed layer. More detail see the implementation of LipNet in [5]

Bibliography

[1] URL: http://www.robots.ox.ac.uk/~vgg/data/lip_reading/index.html#about (visited on 02/02/2019).

[2] URL: http://cs231n.github.io/convolutional-networks/ (visited on 02/02/2019).

[3] URL: https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_default.xml (visited on 02/02/2019).

[4] URL: https://github.com/opencv/opencv_contrib/blob/master/modules/face/data/cascades/haarcascade_mcs_mouth.xml (visited on 02/02/2019).

[5] URL: https://github.com/rizkiarm/LipNet/blob/master/lipnet/model2.py (visited on 02/02/2019).

[6] Yannis M. Assael et al. "LipNet End-to-end Sentence-level Lipreading". In: (2017).

[7] Boris Babenko and Ming-Hsuan Yang Serge Belongie. "Robust Object Tracking with Online Multiple Instance Learning". In: 2011.

[8] Helen L. Bear. "Decoding visemes: improving machine lip-reading (PhDthesis)". In: (2016). p.g 1-5.

[9] *BiasVariance Trade-off*. URL: https://en.wikipedia.org/wiki/Bias%E2%80%93variance_tradeoff (visited on 02/09/2019).

[10] David S. Bolme et al. "Visual Object Tracking using Adaptive Correlation Filters". In: ().

[11] Abel Brown. *Introduction to Object Detection & Image Segmentation*. 2017. URL: http://on-demand.gputechconf.com/gtcdc/2017/presentation/dc7217-abel-brown-deep-learning-object-detection-and-segmentation.pdf (visited on 02/02/2018).

[12] Jason Brownlee. *Deep Learning with Python*. Chap. 17.

[13] *C*. URL: https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#sphx-glr-auto-examples-model-selection-plot-confusion-matrix-py (visited on 02/02/2019).

[14] François Chollet. Ed. by Introduction to Keras. 2018. URL: https://web.stanford.edu/class/cs20si/lectures/march9guestlecture.pdf (visited on 02/02/2019).

[15] François Chollet. *Deep Learning with Python*. Manning, 2018.

[16] François Chollet. *Deep Learning with Python*. p.g 20-22. Manning, 2018.

[17] Joon Son Chung and Andrew Zisserman. "Lip Reading in the Wild". In: (). p.g 3.

[18] *Comparison of OpenCV Interpolation Algorithms*. URL: http://tanbakuchi.com/posts/comparison-of-openv-interpolation-algorithms/ (visited on 02/02/2019).

[19] *Die häufigsten Wörter in deutschsprachigen Texten*. URL: https://www.duden.de/sprachwissen/sprachratgeber/Die-haufigsten-Worter-deutschsprachigen-Texten (visited on 02/01/2019).

[20] Jeff Donahue et al. "Long-term Recurrent Convolutional Networks for Visual Recognition and Description". In: (Nov. 17, 2014). arXiv: http://arxiv.org/abs/1411.4389v4 [cs.CV].

[21] Muhammad Faisal and Sanaullah Manzoor. "Deep Learning for Lip Reading using Audio-Visual Information for Urdu Language". In: (2018).

[22] Dr. Robert Frischholz. 2018. URL: https://facedetection.com/ (visited on 12/11/2018).

[23] Yarin Gal and Zoubin Ghahramani. "A Theoretically Grounded Application of Dropout in Recurrent Neural Networks". In: (Dec. 16, 2015). arXiv: http://arxiv.org/abs/1512.05287v5 [stat.ML].

[24] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: ().

[25] Abiel Gutierrez and Zoe-Alanah Robert. "Lip Reading Word Classification". In: (2017).

[26] Kaiming He et al. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification". In: ().

[27] Sepp Hochreiter and Jürger Schmidhuber. "Long Short-Term Memory". In: (1997).

[28] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: (Feb. 11, 2015). arXiv: http://arxiv.org/abs/1502.03167v3 [cs.LG].

[29] Shuiwang Ji et al. "3D Convolutional Neural Networks for Human Action Recognition". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (2010), pp. 221–231.

[30] Jan Kukačka, Vladimir Golkov, and Daniel Cremers. "Regularization for Deep Learning: A Taxonomy". In: (Oct. 29, 2017). arXiv: http://arxiv.org/abs/1710.10686v1 [cs.LG].

[31] Yan LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: (1998).

[32] Dr. Hanhe Lin. *D*. 2018. URL: https://www.mmsp.uni-konstanz.de/typo3temp/secure_downloads/97871/0/36f431c3dcb9d494a8cd5dd4c483d81883c3d2dc/dlp-lec6-upload.pdf (visited on 02/02/2019).

[33] *Lippenablesen*. URL: http://www.typolis.de/hear/lippenablesen.htm (visited on 02/02/2019).

[34] *Lippenlesen*. URL: https://de.wikihow.com/Lippenlesen (visited on 02/02/2019).

[35] Alan Lukezic et al. "Discriminative Correlation Filter Tracker with Channel and Spatial Reliability". In: ().

[36] Satya Mallick. *Object Tracking using OpenCV (C++/Python)*. 2017. URL: https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/ (visited on 02/02/2019).

[37] Prof. Dr.-Ing. Andreas Meisel. *HAW Deep Learning Project.*

[38] Prof. Dr.-Ing. Andreas Meisel. *HAW Robot Vision course.*

[39] Julien Meynet. *Fast Face Detection Using AdaBoost.* 2013.

[40] Andrew Ng, Younes Bensouda Mourri, and Kian Katanforoosh. *Deep Learning Specialization.* URL: https://www.coursera.org/specializations/deep-learning (visited on 02/02/2019).

[41] *OpenCV Doc: Face Detection using Haar Cascades.* URL: https://docs.opencv.org/3.1.0/d7/d8b/tutorial_py_face_detection.html (visited on 12/11/2018).

[42] Arthur Ouaknine. *Review of Deep Learning Algorithms for Object Detection.* URL: https://medium.com/comet-app/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852 (visited on 02/02/2019).

[43] Adrian Rosebrock. *OpenCV Object Tracking.* 2018. URL: https://www.pyimagesearch.com/2018/07/30/opencv-object-tracking/ (visited on 02/02/2019).

[44] Dr. Adrian Rosebrock. *Practical Python and OpenCV: Case Studies.*

[45] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: (2017).

[46] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A modern approach.* p.g 1-5. Pearson, 2010.

[47] Rajalingappaa Shanmugamani. *Deep learning for computer vision.* Packt Publishing, 2018.

[48] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[49] *Sumed-Area Table.* URL: https://en.wikipedia.org/wiki/Summed-area_table (visited on 12/11/2018).

[50] Du Tran et al. "Learning Spatiotemporal Features with 3D Convolutional Networks". In: (2015).

[51] Paul Viola and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features". In: (2001).

[52] Paul Viola and Michael J. Jones. "Robust Real-Time Face Detection". In: (2004).

[53] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Advances in Neural Information Processing Systems 27, pages 3320-3328. Dec. 2014* (Nov. 6, 2014). arXiv: http://arxiv.org/abs/1411.1792v1 [cs.LG].

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, den _____ _____