SASTRA DEEMED TO BE UNIVERSITY

## (A University under section 3 of the UGC Act, 1956)

**BCSCCS708 / BITCIT707 - MINI PROJECT**



**School of Computing**

# SASTRA Deemed to be University

**TIRUMALAISAMUDRAM THANJAVUR — 613 401 TAMIL NADU, INDIA**

## A Modern Approach For Sign Language Production From Text Using Neural Machine Translation and Generative Adversarial Networks

*Report submitted to the SASTRA Deemed to be University as the requirement for the course*

**BCSCCS708 / BITITC707: MINI PROJECT**

*Submitted by*

**KUNTHIPURAM AJAYKUMAR (Reg. No.: 121003152, CSE)**

**MOTURI LEELA PRASAD (Reg. No.: 121003172, CSE)**

**BEZAWADA VARUN (Reg. No.: 121015011, IT)**

**December 2021**



**SCHOOL OF COMPUTING**

**THANJAVUR, TAMIL NADU, INDIA – 613 401**



**SCHOOL OF COMPUTING THANJAVUR – 613 401**

**Bonafide Certificate**

This is to certify that the report titled “**A Modern Approach For Sign Language Production From Text Using Neural Machine Translation and Generative Adversarial Networks**” submitted as a requirement for the course, BCSCCS708 / BITCIT707**: MINI PROJECT** for B.Tech. is a bonafide record of the work done by **Mr. KUNTHIPURAM AJAYKUMAR (121003152, CSE**), **Mr. MOTURI LEELA PRASAD (121003172, CSE), Mr. BEZAWADA VARUN (121015011, IT)** during the academic year 2020-21, in the School of Computing, under my supervision.

**Signature of Project Supervisor :**

**Name with Affiliation :** Dr. R. Elakkiya

**Date :**

Mini Project *Viva voc*e held on

### Examiner1 Examiner2



**SCHOOL OF COMPUTING**

**THANJAVUR – 613 401**

**Acknowledgement**

First and foremost we thank the almighty for helping us to gain support in all forms to finish our Mini Project successfully. We express our sincere thanks to

**Dr.S.VAIDHYASUBRAMANIAM**, *Vice Chancellor* and **Dr R. CHANDRA MOULI**, *Registrar*, **SASTRA UNIVERSITY**, for permitting us to do this Mini Project as a part of our curriculum.

We express our profound gratitude to **Dr.UMA MAKESHWARI**, *Dean School of Computer Science Engineering*, **SASTRA UNIVERSITY**for her complete support throughout the Mini Project.

We are fortunate to have, **Dr R. ELAKKIYA**, as our project guide. Her valuable assistance and supervision guided us towards the successful completion of our Mini project. Lastly, I thank all the technical and non-technical staff of School of Computing and our friends for their constant support throughout the completion of our Mini Project Report.

**ABSTRACT**

Text2Sign is a production of sign language using Neural Machine Translation(NMT) along with Motion Graphs(MGs) and Generative Adversarial Networks(GAN). This implementation is capable of producing sign videos from spoken language sentences. The previous proposed systems are dependent on heavily annotated data and tackled using animated avatars, which is limited to pre-recorded phrases and the production of motion capture data is costly.

We proposed a new approach using NMT, Computer Graphics and Neural Network based image/video generation. We solved this by dividing the problem into two sub-problems. First, we translate the spoken language sentences into human pose sequence by using NMT along with MG. Secondly, the resulting human pose sequence is translated to sign language video using GAN. We evaluate the translation abilities of this implementation on the PHOENIX14T Sign Language Translation dataset.

**KEYWORDS:** Neural Machine Translation, Motion Graphs, Generative Adversarial Networks.

**TABLE OF CONTENTS**

|  |  |  |
| --- | --- | --- |
| **S.No** | **Title** | **Page No.** |
| i | Bonafide Certificate | 3 |
| ii | Acknowledgements | 4 |
| ii | Abstract | 5 |
| iii | List of Figures | 7 |
| iv | List of Tables | 7 |
| v | Abbreviations | 8 |
| vi | Notations | 8 |
| 1 | Summary of the base paper  1.1 Introduction  1.2 Objective  1.3 Literature Survey  1.4 Methodology  1.4.1 NMT  1.4.2 Motion Graphs  1.4.3 GANs  1.5 Dataset  1.5.1 Preprocessing | 9  9  10  11  12  12  13  14  15  15 |
| 2 | Merits and Demerits | 16 |
| 3 | Source Code | 16 |
| 4 | Results and Snapshots | 43 |
| 5 | Conclusions and Future work | 46 |
| 6 | References | 47 |

## LIST OF FIGURES

|  |  |  |
| --- | --- | --- |
| **Figure**  **No** | **Figure Name** | **Page No** |
| 1 | Avatar-based sign language representations | 9 |
| 2 | Overview of the Spoken Language Sentence to  Sign Language Video conversion | 9 |
| 3 | NMT-based encoder-decoder architecture | 11 |
| 4 | Illustration of Text2Gloss2Pose | 12 |
| 5 | Basic architecture of Generator and  Discriminator in GANs | 13 |
| 6 | Image Generation in GAN | 14 |
| 7 | NMT output executed in kaggle.com | 42 |
| 8 | Motion Graphs on image | 42 |
| 9 | Output of GAN | 43 |

**LIST OF TABLES**

|  |  |  |
| --- | --- | --- |
| **Table No** | **Table Name** | **Page No** |
| 1 | Literature Survey | 10 |

**ABBREVIATIONS**

NMT Neural Machine Translation

MG Motion Graphs

GAN Generative Adversarial Networks

ASL American Sign Language

WHO World Health Organization

SLP Sign Language Production

RNN Recurrent Neural Network

GRU Gated Recurrent Units

CNN Convolutional neural network

LTSM Long Short Term Memor

**NOTATIONS**

## English Symbols (in alphabetical order)

hn Hypothesis

## Greek Symbols (in alphabetical order)

** Learning Rate

∑ Summation

## Miscellaneous Symbols (in alphabetical order)

x absolute value ofx

**CHAPTER 1**

**SUMMARY OF THE BASE PAPER**

**Title** : Text2Sign: Towards Sign Language Production Using Neural Machine Translation and Generative Adversarial Networks

**Journal Name** : International Journal of Computer Vision

**Publisher** : Springer

**Year** : Jan 2020

**Indexed** : SCIE

**1.1 INTRODUCTION :**

According to the WHO (2018) there are around 0.466 billion people around the world are suffering from deaf and hearing loss problems. Widely, these people primarily communicate in sign languages which have their grammar and linguistic syntax. In this work, we have used Neural Machine Translation along with Motion Graphs and Generative Adversarial Networks and translated spoken sentence to sign video. We solved this problem by diving into sub-problems. Firstly, we translate the spoken sentences into sign language pose sequences using Neural Machine translation (NMT) along with Motion Graphs. Finally, output of motion graphs is given as input to Generative Adversarial Network(GAN) to generate realistic sign language image sequence.



**Fig 1:** Avatar-based sign language representations

## Furthermore, producing sign language image sequences from normal spoken language sentence is a difficult task that can’t be done simply by mapping.

**1.2 OBJECTIVE**

The ultimate objective of the project is to convert spoken text to sign language video. The spoken sentence is converted to a sign gloss using Neural Machine Translation (NMT). The sign language gloss i.e., the output from the Neural Machine Translation is converted to sign language pose using Motion Graphs (MG). This sign language pose is then given as input to Generative Adversarial Networks (GAN). Generative Adversarial Networks generates a sequence of images representing the sign language for the input spoken language sentence that is given as input to the Neural Machine Translation initially.

## 

**Fig 2 :** Overview of the Spoken Sentence to Sign Language Video conversion

**1.3 LITERATURE SURVEY**

|  |  |
| --- | --- |
| Paper | Description |
| Luong, T., Pham, H., & Manning, C. D. (2015).Effective approaches to attention-based neural machine translation. | Using a seq2seq model ,This mechanism translates text to gloss. In this architecture , every cell gives the relevant context to the other cell in the network. |
| Camgoz, N. C., Hadfield, S., Koller, O., Ney, H., & Bowden, R. (2018).Neural sign language translation. In IEEE Conference on computer vision and pattern recognition (CVPR). | Combined a seq2seq model with a Convolutional Neural Network (CNN) for Sign2text translations. |
| Kovar, L., Gleicher, M., & Pighin, F. (2002). Motion graphs.In Proceedings of the 29th annual conference on computer graphics and interactive techniques. | Motion Graphs (MGs) is a mechanism related to computer graphics , used in animations to dynamically configure a character. It is used for generating a sequence of 2D skeletal poses from a given gloss Sequence |
| Wang, T. C., Liu, M. Y., Zhu, J. Y., Tao, A., Kautz, J., & Catanzaro, B. (2018b).High-resolution image synthesis and semantic manipulation with conditional GANs. | This system uses style transfer like pix2pixHD and also used for HD quality of video output. GANs are capable of producing images based on conditions. |

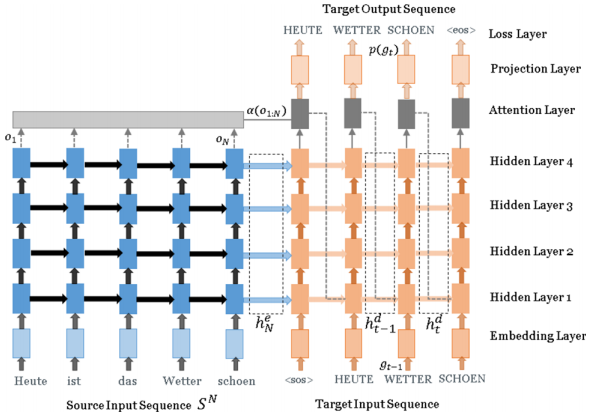
Table 1 : Literature Survey

## 1.4 METHODOLOGY

**1.4.1 Attention based NMT (Nerual Machine Translation):**

NMT is a Recurrent Neural Network (RNN) architecture used for translating a sentence of a language to other languages. We have used an encoder and decoder based architecture to translate the spoken sentence to gloss. NMT architecture mainly has an encoder neural network and decoder neural network. To handle long term dependencies in the spoken language sentence we use Long Short Term Memory(LSTM) or Gated Recurrent Units(GRUs). In this architecture , every cell gives the relevant context to the other cell in the network. This will increase the performance of NMT translation.

We used attention based approach which gives additional information to the decoder network by observing encoder’s hidden states. Luong et al., based attention mechanism is used to improve the long term dependencies.



**Fig 3 :** NMT-based encoder-decoder architecture

## 1.4.1.1 Encoder:

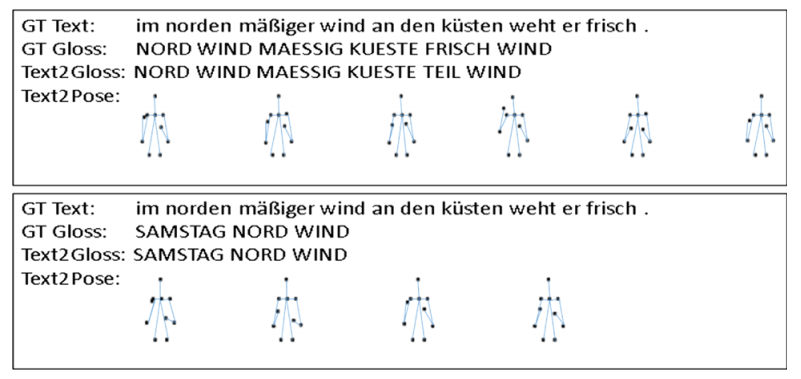
Consider a sentence S(N) = {w1, w2, w3,……, wn} where N is number of words. Here encoder encodes the input sentence to learn the relationship between the words. Encoder maps the output to the latent space.



## 1.4.1.2 Decoder:

The encoder outputs and hidden representations of the encoded sentence is given as input to the decoder which uses the attention mechanism to produce the probability of the word in other language.





**Fig 4 :** Illustration of Text2Gloss2pose

For translating text to pose, CNN plays major role in first step. But in divergence, our model used probabilities of NMT’s decoder network which is used for the Motion Graph for pose image. Thus we obtain text to pose translation.

**1.4.2 MOTION GRAPHS**

Motion Graphs (MGs) is a mechanism related to computer graphics , used in animations to dynamically configure a character. It is used for generating a sequence of 2D skeletal poses from a given gloss Sequence. It is done by detecting and locating the position of objects in the image.

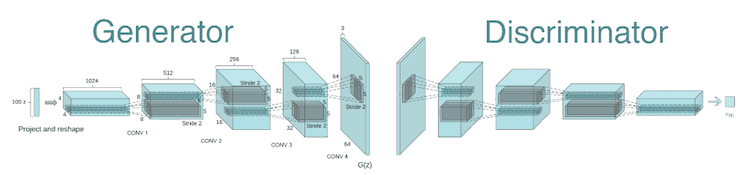
We can build the Motion Graph for pose data by dividing the sign text into separate gloss and combining every motion text by glosses. The sequences of motion are taken out from huge set of motion capture data. We have to identify the transition points in the motion data those are main frames in the motion. We used probabilities obtained from NMT’s decoder at each cell for transition points. The transition points are must lie inside the permissible level. To avoid visual imbalance, threshold is set to low value

**1.4.3 Generative Adversarial Networks (GAN)**

GAN is an advanced neural network in Deep Learning. It is mainly used for generating fake images which doesn’t exist before. The GAN technique trains two Neural Networks simultaneously: A Generator neural network and a Discriminator neural network.

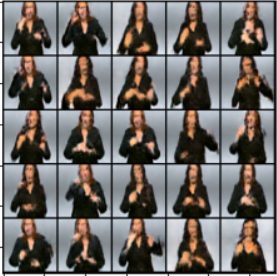
Generator creates new instances of data. The Generator network is a De-convolutional Neural Network which takes noise from Motion Graphs as an input and returns a generated image as an output. The goal of the Generator network is to deceive the Discriminator network by generating realistic fake images.

Discriminator D evaluates the generated image. The Discriminator network is Convolutional Neural Network which is trained on fake images from the Generator network and real images from a given dataset. The goal of the Discriminator neural network is to find the difference between real and fake images.



**Fig 5 :** Basic architecture of Generator and Discriminator in GANs

ZUSCHAUER BEGRUESSEN



**Fig 6 :** Image Generation in GAN

## 1.5 DATASET

Text2Sign Model is trained using using PHOENIX 14 T dataset . Dataset has 8257 sequence has been done by 9 signers. It consists of spoken language and sign language gloss vocabulary of 2887 and 1066, respectively. Every spoken language sentence has its own sign gloss and their respective poses.

The dataset consist of sequence of images which are extracted from the video i.e., extracting frames from the video. These frames are preprocessed and are used for training.

**1.5.1 Dataset pre-processing**

While training NMT we consider spoken language sentences and their respective sign glosses. In dataset the files containing these spoken languages sentences and sign gloss are parsed using Java and the sentences are extracted be omitting the unwanted tags and extra spaces.

Spoken language sentences are preprocessed using tensorflow tokenizer. And the sentence is converted to sequence and padded null vectors to tackle dimensional issues.

**CHAPTER 2**

**Merits and Demerits**

## The previous works related to the topic of generating sign language video the implementation was carried out using avatar based video generating approach which is costly. Instead of using traditional avatar based approach for generating sign language video we have used realistic human images and conditioned them using Motion Graphs(MG) and Generative Adversarial Networks(GAN).

Since we are using Generative Adversarial Network(GAN) for sign language video generation we need a large dataset to train the GAN model. Also we need huge computing resources for training the GAN model to produce better results with good accuracy.

**CHAPTER 3**

**Source Code**

**Neural Machine Translation**

```python

from \_\_future\_\_ import absolute\_import, division, print\_function, unicode\_literals

try:

# %tensorflow\_version only exists in Colab.

%tensorflow\_version 2.x

except Exception:

pass

import tensorflow as tf

import matplotlib.pyplot as plt

import matplotlib.ticker as ticker

from sklearn.model\_selection import train\_test\_split

import pandas as pd

import unicodedata

import re

import numpy as np

import os

import io

import time

from string import digits

# Pre process and create dataset

def preprocess\_sentence(sentence):

#sentence = unicode\_to\_ascii(sentence.strip())

num\_digits= str.maketrans('','', digits)

sentence= re.sub(" +", " ", sentence)

sentence= re.sub("'", '', sentence)

sentence= sentence.translate(num\_digits)

sentence= sentence.strip()

sentence= re.sub(r"([?.!,¿])", r" \1 ", sentence)

sentence = sentence.rstrip().strip()

sentence= 'start\_ ' + sentence + ' \_end'

return sentence

en\_sentence = u"May I borrow this book?"

sp\_sentence = u"¿Puedo tomar prestado este libro?"

print(preprocess\_sentence(en\_sentence))

print(preprocess\_sentence(sp\_sentence).encode('utf-8'))

def create\_dataset(path):

lines = io.open(path, encoding='UTF-8').read().strip().split('\n')

#print(lines)

words = [preprocess\_sentence(l) for l in lines]

return words

sample\_size=7096

source = create\_dataset(r"../input/germansign-language/phoenix2014T.train.de")+create\_dataset(r"../input/germansign-language/phoenix2014T.test.de")

print(source[0])

target =create\_dataset(r"../input/germansign-language/phoenix2014T.train.gloss")+create\_dataset(r"../input/germansign-language/phoenix2014T.test.gloss")

print(target[0])

def max\_length(tensor):

return max(len(t) for t in tensor)

source\_sentence\_tokenizer= tf.keras.preprocessing.text.Tokenizer(filters='')

source\_sentence\_tokenizer.fit\_on\_texts(source)

source\_tensor = source\_sentence\_tokenizer.texts\_to\_sequences(source)

source\_tensor= tf.keras.preprocessing.sequence.pad\_sequences(source\_tensor,padding='post' )

print(len(source\_tensor[0]))

target\_sentence\_tokenizer= tf.keras.preprocessing.text.Tokenizer(filters='')

target\_sentence\_tokenizer.fit\_on\_texts(target)

target\_tensor = target\_sentence\_tokenizer.texts\_to\_sequences(target)

target\_tensor= tf.keras.preprocessing.sequence.pad\_sequences(target\_tensor,padding='post' )

print(len(target\_tensor[0]))

max\_target\_length= max(len(t) for t in target\_tensor)

print(max\_target\_length)

max\_source\_length= max(len(t) for t in source\_tensor)

print(max\_source\_length)

## Creating Train and Test dataset

source\_train\_tensor, source\_test\_tensor, target\_train\_tensor, target\_test\_tensor= train\_test\_split(source\_tensor, target\_tensor,test\_size=0.2)

# Creating training and validation sets using an 80-20 split

input\_tensor\_train, input\_tensor\_val, target\_tensor\_train, target\_tensor\_val = train\_test\_split(source\_tensor, target\_tensor, test\_size=0.1)

# Show length

print(len(input\_tensor\_train), len(target\_tensor\_train), len(input\_tensor\_val), len(target\_tensor\_val))

```

### Create a tf.data dataset

BUFFER\_SIZE = len(source\_train\_tensor)

BATCH\_SIZE = 64

steps\_per\_epoch = len(source\_train\_tensor)//BATCH\_SIZE

embedding\_dim = 256

units = 1024

vocab\_inp\_size = len(source\_sentence\_tokenizer.word\_index)+1

vocab\_tar\_size = len(target\_sentence\_tokenizer.word\_index)+1

dataset = tf.data.Dataset.from\_tensor\_slices((source\_train\_tensor, target\_train\_tensor)).shuffle(BUFFER\_SIZE)

dataset = dataset.batch(BATCH\_SIZE, drop\_remainder=True)

type(dataset)

example\_input\_batch, example\_target\_batch = next(iter(dataset))

example\_input\_batch.shape, example\_target\_batch.shape

```

```python

class Encoder(tf.keras.Model):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, enc\_units, batch\_sz):

super(Encoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.enc\_units = enc\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = tf.keras.layers.GRU(self.enc\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.gru2 = tf.keras.layers.GRU(self.enc\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.gru3 = tf.keras.layers.GRU(self.enc\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

def call(self, x, hidden):

x = self.embedding(x)

output, state = self.gru(x, initial\_state = hidden)

output,state = self.gru2(output,initial\_state=state)

output,state= self.gru3(output,initial\_state=state)

return output, state

def initialize\_hidden\_state(self):

return tf.zeros((self.batch\_sz, self.enc\_units))

```

```python

encoder = Encoder(vocab\_inp\_size, embedding\_dim, units, BATCH\_SIZE)

# sample input

sample\_hidden = encoder.initialize\_hidden\_state()

sample\_output, sample\_hidden = encoder(example\_input\_batch, sample\_hidden)

print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sample\_output.shape))

print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample\_hidden.shape))

```

```python

class BahdanauAttention(tf.keras.layers.Layer):

def \_\_init\_\_(self, units):

super(BahdanauAttention, self).\_\_init\_\_()

self.W1 = tf.keras.layers.Dense(units)

self.W2 = tf.keras.layers.Dense(units)

self.V = tf.keras.layers.Dense(1)

def call(self, query, values):

# hidden shape == (batch\_size, hidden size)

# hidden\_with\_time\_axis shape == (batch\_size, 1, hidden size)

# we are doing this to perform addition to calculate the score

hidden\_with\_time\_axis = tf.expand\_dims(query, 1)

# score shape == (batch\_size, max\_length, 1)

# we get 1 at the last axis because we are applying score to self.V

# the shape of the tensor before applying self.V is (batch\_size, max\_length, units)

score = self.V(tf.nn.tanh(

self.W1(values) + self.W2(hidden\_with\_time\_axis)))

# attention\_weights shape == (batch\_size, max\_length, 1)

attention\_weights = tf.nn.softmax(score, axis=1)

# context\_vector shape after sum == (batch\_size, hidden\_size)

context\_vector = attention\_weights \* values

context\_vector = tf.reduce\_sum(context\_vector, axis=1)

return context\_vector, attention\_weights

```

```python

attention\_layer = BahdanauAttention(10)

attention\_result, attention\_weights = attention\_layer(sample\_hidden, sample\_output)

print("Attention result shape: (batch size, units) {}".format(attention\_result.shape))

print("Attention weights shape: (batch\_size, sequence\_length, 1) {}".format(attention\_weights.shape))

```

```python

class Decoder(tf.keras.Model):

def \_\_init\_\_(self, vocab\_size, embedding\_dim, dec\_units, batch\_sz):

super(Decoder, self).\_\_init\_\_()

self.batch\_sz = batch\_sz

self.dec\_units = dec\_units

self.embedding = tf.keras.layers.Embedding(vocab\_size, embedding\_dim)

self.gru = tf.keras.layers.GRU(self.dec\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.gru2 = tf.keras.layers.GRU(self.dec\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.gru3 = tf.keras.layers.GRU(self.dec\_units,

return\_sequences=True,

return\_state=True,

recurrent\_initializer='glorot\_uniform')

self.fc = tf.keras.layers.Dense(vocab\_size)

# used for attention

self.attention = BahdanauAttention(self.dec\_units)

def call(self, x, hidden, enc\_output):

# enc\_output shape == (batch\_size, max\_length, hidden\_size)

context\_vector, attention\_weights = self.attention(hidden, enc\_output)

# x shape after passing through embedding == (batch\_size, 1, embedding\_dim)

x = self.embedding(x)

# x shape after concatenation == (batch\_size, 1, embedding\_dim + hidden\_size)

x = tf.concat([tf.expand\_dims(context\_vector, 1), x], axis=-1)

# passing the concatenated vector to the GRU

output, state = self.gru(x)

output,state = self.gru2(output)

output,state = self.gru3(output)

# output shape == (batch\_size \* 1, hidden\_size)

output = tf.reshape(output, (-1, output.shape[2]))

# output shape == (batch\_size, vocab)

x = self.fc(output)

return x, state, attention\_weights

```

```python

decoder = Decoder(vocab\_tar\_size, embedding\_dim, units, BATCH\_SIZE)

sample\_decoder\_output, \_, \_ = decoder(tf.random.uniform((BATCH\_SIZE, 1)),

sample\_hidden, sample\_output)

print ('Decoder output shape: (batch\_size, vocab size) {}'.format(sample\_decoder\_output.shape))

```

## Define the optimizer and the loss function

```python

optimizer = tf.keras.optimizers.Adam()

loss\_object = tf.keras.losses.SparseCategoricalCrossentropy(

from\_logits=True, reduction='none')

def loss\_function(real, pred):

mask = tf.math.logical\_not(tf.math.equal(real, 0))

loss\_ = loss\_object(real, pred)

mask = tf.cast(mask, dtype=loss\_.dtype)

loss\_ \*= mask

return tf.reduce\_mean(loss\_)

```

```python

checkpoint\_dir = 'training\_checkpoints'

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

checkpoint = tf.train.Checkpoint(optimizer=optimizer,

encoder=encoder,

decoder=decoder)

```

```python

@tf.function

def train\_step(inp, targ, enc\_hidden):

loss = 0

with tf.GradientTape() as tape:

enc\_output, enc\_hidden = encoder(inp, enc\_hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([target\_sentence\_tokenizer.word\_index['start\_']] \* BATCH\_SIZE, 1)

# Teacher forcing - feeding the target as the next input

for t in range(1, targ.shape[1]):

# passing enc\_output to the decoder

predictions, dec\_hidden, \_ = decoder(dec\_input, dec\_hidden, enc\_output)

loss += loss\_function(targ[:, t], predictions)

# using teacher forcing

dec\_input = tf.expand\_dims(targ[:, t], 1)

batch\_loss = (loss / int(targ.shape[1]))

variables = encoder.trainable\_variables + decoder.trainable\_variables

gradients = tape.gradient(loss, variables)

optimizer.apply\_gradients(zip(gradients, variables))

return batch\_loss

```

```python

steps\_per\_epoch

EPOCHS = 20

history={}

history['loss']=[];

for epoch in range(EPOCHS):

start = time.time()

enc\_hidden = encoder.initialize\_hidden\_state()

total\_loss = 0

for (batch, (inp, targ)) in enumerate(dataset.take(steps\_per\_epoch)):

batch\_loss = train\_step(inp, targ, enc\_hidden)

total\_loss += batch\_loss

if batch % 100 == 0:

print('Epoch {} Batch {} loss {}'.format(epoch + 1,batch, batch\_loss.numpy()))

# saving (checkpoint) the model every 2 epochs

if (epoch + 1) % 2 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)

print('Epoch {} Loss {:.4f}'.format(epoch + 1,

total\_loss / steps\_per\_epoch))

history['loss'].append(total\_loss / steps\_per\_epoch);

print('Time taken for 1 epoch {} sec\n'.format(time.time() - start))

```

```python

# summarize history for loss

plt.plot(history['loss'])

plt.title('model loss')

plt.ylabel('loss')

plt.xlabel('epoch')

#plt.legend(['train', 'test'], loc='upper left')

plt.show()

```

# \_\*Evaluating\*\_ #

```python

def evaluate(sentence):

attention\_plot = np.zeros((max\_target\_length, max\_source\_length))

sentence = preprocess\_sentence(sentence)

#print(sentence)

#print(source\_sentence\_tokenizer.word\_index)

inputs = [source\_sentence\_tokenizer.word\_index[i] for i in sentence.split(' ')]

inputs = tf.keras.preprocessing.sequence.pad\_sequences([inputs],

maxlen=max\_source\_length,

padding='post')

inputs = tf.convert\_to\_tensor(inputs)

result = ''

hidden = [tf.zeros((1, units))]

enc\_out, enc\_hidden = encoder(inputs, hidden)

dec\_hidden = enc\_hidden

dec\_input = tf.expand\_dims([target\_sentence\_tokenizer.word\_index['start\_']], 0)

for t in range(max\_target\_length):

predictions, dec\_hidden, attention\_weights = decoder(dec\_input,

dec\_hidden,

enc\_out)

predicted\_id = tf.argmax(predictions[0]).numpy()

result += target\_sentence\_tokenizer.index\_word[predicted\_id] + ' '

if target\_sentence\_tokenizer.index\_word[predicted\_id] == '\_end':

return result, sentence, attention\_plot

# the predicted ID is fed back into the model

dec\_input = tf.expand\_dims([predicted\_id], 0)

return result, sentence, attention\_plot

```

```python

# function for plotting the attention weights

def plot\_attention(attention, sentence, predicted\_sentence):

fig = plt.figure(figsize=(10,10))

ax = fig.add\_subplot(1, 1, 1)

ax.matshow(attention, cmap='viridis')

fontdict = {'fontsize': 14}

ax.set\_xticklabels([''] + sentence, fontdict=fontdict, rotation=90)

ax.set\_yticklabels([''] + predicted\_sentence, fontdict=fontdict)

ax.xaxis.set\_major\_locator(ticker.MultipleLocator(1))

ax.yaxis.set\_major\_locator(ticker.MultipleLocator(1))

plt.show()

```

```python

def translate(sentence):

result, sentence, attention\_plot = evaluate(sentence)

result=result.upper()

print('Input: %s' % (sentence))

print('Predicted translation: {}'.format(result))

attention\_plot = attention\_plot[:len(result.split(' ')), :len(sentence.split(' '))]

#plot\_attention(attention\_plot, sentence.split(' '), result.split(' '))

return result

```

# Evaluate and Performance Metrics

```python

translate(u'liebe zuschauer guten abend')

```

```python

translate(u'da auch wieder mit schnee vermischt')

```

```python

from nltk.translate.bleu\_score import sentence\_bleu

def find\_bleu(sentence,answer):

predected= translate(sentence)

reference = [answer.split()]

candidate = predected.split()

return sentence\_bleu(reference, candidate, weights=(1.0,0,0,0))

```

```python

sentence = u"liebe zuschauer guten abend";

answer = u"\_\_ON\_\_ LIEB ZUSCHAUER ABEND \_END"

score = find\_bleu(sentence,answer)

print("Blue Score :",score,sep="\t")

```

```python

predicted=translate(u"aber erfreuliche nachricht.")

```

**Motion Graphs Code**

def pose\_estimate(frame):

frameWidth = frame.shape[1]

frameHeight = frame.shape[0]

net.setInput(cv.dnn.blobFromImage(frame, 1.0, (inWidth, inHeight), (127.5, 127.5, 127.5), swapRB=True, crop=False))

out = net.forward()

out = out[:, :19, :, :] # MobileNet output [1, 57, -1, -1], we only need the first 19 elements

assert(len(BODY\_PARTS) == out.shape[1])

#print(out.shape)

points = []

for i in range(len(BODY\_PARTS)):

# Slice heatmap of corresponging body's part.

heatMap = out[0, i, :, :]

# Originally, we try to find all the local maximums. To simplify a sample

# we just find a global one. However only a single pose at the same time

# could be detected this way.

\_, conf, \_, point = cv.minMaxLoc(heatMap)

x = (frameWidth \* point[0]) / out.shape[3]

y = (frameHeight \* point[1]) / out.shape[2]

# Add a point if it's confidence is higher than threshold.

points.append((int(x), int(y)) if conf > thr else None)

#print(points)

#frame = np.zeros((frameWidth,frameHeight,3), np.uint8)

for pair in POSE\_PAIRS:

partFrom = pair[0]

partTo = pair[1]

assert(partFrom in BODY\_PARTS)

assert(partTo in BODY\_PARTS)

idFrom = BODY\_PARTS[partFrom]

idTo = BODY\_PARTS[partTo]

if points[idFrom] and points[idTo]:

cv.line(frame, points[idFrom], points[idTo], (0, 255, 0), 3)

cv.ellipse(frame, points[idFrom], (3, 3), 0, 0, 360, (0, 0, 255), cv.FILLED)

cv.ellipse(frame, points[idTo], (3, 3), 0, 0, 360, (0, 0, 255), cv.FILLED)

t, \_ = net.getPerfProfile()

freq = cv.getTickFrequency() / 1000

cv.putText(frame, '%.2fms' % (t / freq), (10, 20), cv.FONT\_HERSHEY\_SIMPLEX, 0.5, (0, 0, 0))

x=cv.cvtColor(frame,cv.COLOR\_RGB2GRAY)

t = cv.resize(x, (100, 128),

interpolation = cv.INTER\_NEAREST)

return points , frame,t

points,ans,t = pose\_estimate(img);

import pandas as pd

column\_names=['id','diri','signer','gloss']

df= pd.DataFrame(columns = column\_names);

for line in lines:

words= line.split("|")

df = df.append({'id' :words[0], 'diri' :words[1], 'signer' :words[2],'gloss':words[3]},

ignore\_index = True)

vocab = open(vocab,'r')

glosses={};

gloss2={}

for word in vocab.readlines():

glosses[word.strip()]=np.zeros((128,100,1,1))

gloss2[word.strip()]=0;

count=0;

for id,row in df.iterrows():

words = row['gloss'].split(" ");

images = glob.glob(os.path.join(diri,row['diri']))

n=len(images)

count+=n;

if n==0:

continue;

tt=n//len(words)

for id,word in enumerate(words):

if word not in glosses:

continue;

for i in range(0,tt):

img = cv.imread(images[id\*tt+i])

points,ans,t = pose\_estimate(img)

t = t.reshape(128,100,1,1)

glosses[word]+=t

gloss2[word]+=1

if i>3:

break;

print(count)

for k,v in glosses.items():

if k in gloss2 and gloss2[k]>0:

glosses[k]=glosses[k]/gloss2[k]

**Generative Adversarial Networks Code:**

import torch

import torch.nn as nn

import torch.nn.functional as F

def weights\_init(w):

"""

Initializes the weights of the layer, w.

"""

classname = w.\_\_class\_\_.\_\_name\_\_

if classname.find('conv') != -1:

nn.init.normal\_(w.weight.data, 0.0, 0.02)

elif classname.find('bn') != -1:

nn.init.normal\_(w.weight.data, 1.0, 0.02)

nn.init.constant\_(w.bias.data, 0)

# Generator Network

class Generator(nn.Module):

def \_\_init\_\_(self, params):

super().\_\_init\_\_()

self.tconv1 = nn.ConvTranspose2d(params['nz'], params['ngf']\*8,

kernel\_size=4, stride=1, padding=0, bias=False)

self.bn1 = nn.BatchNorm2d(params['ngf']\*8)

self.tconv2 = nn.ConvTranspose2d(params['ngf']\*8, params['ngf']\*4,

4, 2, 1, bias=False)

self.bn2 = nn.BatchNorm2d(params['ngf']\*4)

self.tconv3 = nn.ConvTranspose2d(params['ngf']\*4, params['ngf']\*2,

4, 2, 1, bias=False)

self.bn3 = nn.BatchNorm2d(params['ngf']\*2)

self.tconv4 = nn.ConvTranspose2d(params['ngf']\*2, params['ngf'],

4, 2, 1, bias=False)

self.bn4 = nn.BatchNorm2d(params['ngf'])

self.tconv5 = nn.ConvTranspose2d(params['ngf'], params['nc'],

4, 2, 1, bias=False)

def forward(self, x):

x = F.relu(self.bn1(self.tconv1(x)))

x = F.relu(self.bn2(self.tconv2(x)))

x = F.relu(self.bn3(self.tconv3(x)))

x = F.relu(self.bn4(self.tconv4(x)))

x = F.tanh(self.tconv5(x))

return x

# Discriminator Network

class Discriminator(nn.Module):

def \_\_init\_\_(self, params):

super().\_\_init\_\_()

self.conv1 = nn.Conv2d(params['nc'], params['ndf'],

4, 2, 1, bias=False)

self.conv2 = nn.Conv2d(params['ndf'], params['ndf']\*2,

4, 2, 1, bias=False)

self.bn2 = nn.BatchNorm2d(params['ndf']\*2)

self.conv3 = nn.Conv2d(params['ndf']\*2, params['ndf']\*4,

4, 2, 1, bias=False)

self.bn3 = nn.BatchNorm2d(params['ndf']\*4)

self.conv4 = nn.Conv2d(params['ndf']\*4, params['ndf']\*8,

4, 2, 1, bias=False)

self.bn4 = nn.BatchNorm2d(params['ndf']\*8)

self.conv5 = nn.Conv2d(params['ndf']\*8, 1, 4, 1, 0, bias=False)

def forward(self, x):

x = F.leaky\_relu(self.conv1(x), 0.2, True)

x = F.leaky\_relu(self.bn2(self.conv2(x)), 0.2, True)

x = F.leaky\_relu(self.bn3(self.conv3(x)), 0.2, True)

x = F.leaky\_relu(self.bn4(self.conv4(x)), 0.2, True)

x = F.sigmoid(self.conv5(x))

return x

import torch

import torch.nn as nn

import torch.optim as optim

import torchvision.utils as vutils

import numpy as np

import matplotlib.pyplot as plt

import matplotlib.animation as animation

import random

# Set seed

seed = 369

random.seed(seed)

torch.manual\_seed(seed)

print("Random Seed: ", seed)

# Parameters

params = {

"bsize" : 128,

'imsize' : 64,

'nc' : 3,

'nz' : 100,

'ngf' : 64,

'ndf' : 64,

'nepochs' : 50,

'lr' : 0.0002,

'beta1' : 0.5,

'save\_epoch' : 2}

device = torch.device("cuda:0" if(torch.cuda.is\_available()) else "cpu")

print(device, " will be used.\n")

dataloader = get\_celeba(params)

# Plot

sample\_batch = next(iter(dataloader))

plt.figure(figsize=(8, 8))

plt.axis("off")

plt.title("Training Images")

plt.imshow(np.transpose(vutils.make\_grid(

sample\_batch[0].to(device)[ : 64], padding=2, normalize=True).cpu(), (1, 2, 0)))

plt.show()

# generator.

netG = Generator(params).to(device)

netG.apply(weights\_init)

print(netG)

# discriminator.

netD = Discriminator(params).to(device)

netD.apply(weights\_init)

print(netD)

criterion = nn.BCELoss()

fixed\_noise = torch.randn(64, params['nz'], 1, 1, device=device)

real\_label = 1

fake\_label = 0

# Optimizer

optimizerD = optim.Adam(netD.parameters(), lr=params['lr'], betas=(params['beta1'], 0.999))

optimizerG = optim.Adam(netG.parameters(), lr=params['lr'], betas=(params['beta1'], 0.999))

# Stores generated images as training progresses.

img\_list = []

# Stores generator losses during training.

G\_losses = []

# Stores discriminator losses during training.

D\_losses = []

iters = 0

print("Starting Training Loop...")

print("-"\*25)

for epoch in range(params['nepochs']):

for i, data in enumerate(dataloader, 0):

real\_data = data[0].to(device)

b\_size = real\_data.size(0)

netD.zero\_grad()

label = torch.full((b\_size, ), int(real\_label)\*1.0, device=device)

output = netD(real\_data).view(-1)

errD\_real = criterion(output, label)

errD\_real.backward()

D\_x = output.mean().item()

noise3 = torch.randn(b\_size, params['nz'], 1, 1, device=device)

t2=random.choice(select1);

noise=torch.from\_numpy(gloss[t2])

if b\_size!=128:

noise = noise3;

else:

noise = noise3+noise.to(torch.device(device))

fake\_data = netG(noise)

label.fill\_(fake\_label )

output = netD(fake\_data.detach()).view(-1)

errD\_fake = criterion(output, label)

errD\_fake.backward()

D\_G\_z1 = output.mean().item()

errD = errD\_real + errD\_fake

optimizerD.step()

netG.zero\_grad()

label.fill\_(real\_label)

output = netD(fake\_data).view(-1)

errG = criterion(output, label)

errG.backward()

D\_G\_z2 = output.mean().item()

optimizerG.step()

if i%50 == 0:

print(torch.cuda.is\_available())

print('[%d/%d][%d/%d]\tLoss\_D: %.4f\tLoss\_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'

% (epoch, params['nepochs'], i, len(dataloader),

errD.item(), errG.item(), D\_x, D\_G\_z1, D\_G\_z2))

show\_tensor\_images(fake\_data)

show\_tensor\_images(real\_data)

G\_losses.append(errG.item())

D\_losses.append(errD.item())

if (iters % 100 == 0) or ((epoch == params['nepochs']-1) and (i == len(dataloader)-1)):

with torch.no\_grad():

fake\_data = netG(fixed\_noise).detach().cpu()

img\_list.append(vutils.make\_grid(fake\_data, padding=2, normalize=True))

iters += 1

if epoch % params['save\_epoch'] == 0:

torch.save({

'generator' : netG.state\_dict(),

'discriminator' : netD.state\_dict(),

'optimizerG' : optimizerG.state\_dict(),

'optimizerD' : optimizerD.state\_dict(),

'params' : params

}, 'model\_epoch\_{}.pth'.format(epoch))

print("n\_ephocs completed...")

torch.save({

'generator' : netG.state\_dict(),

'discriminator' : netD.state\_dict(),

'optimizerG' : optimizerG.state\_dict(),

'optimizerD' : optimizerD.state\_dict(),

'params' : params

}, 'model\_final.pth')

# Plot

plt.figure(figsize=(10,5))

plt.title("Generator and Discriminator Loss During Training")

plt.plot(G\_losses,label="G")

plt.plot(D\_losses,label="D")

plt.xlabel("iterations")

plt.ylabel("Loss")

plt.legend()

plt.show()

print("plot-2")

# Animation showing the improvements of the generator.

fig = plt.figure(figsize=(8,8))

plt.axis("off")

ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img\_list]

anim = animation.ArtistAnimation(fig, ims, interval=1000, repeat\_delay=1000, blit=True)

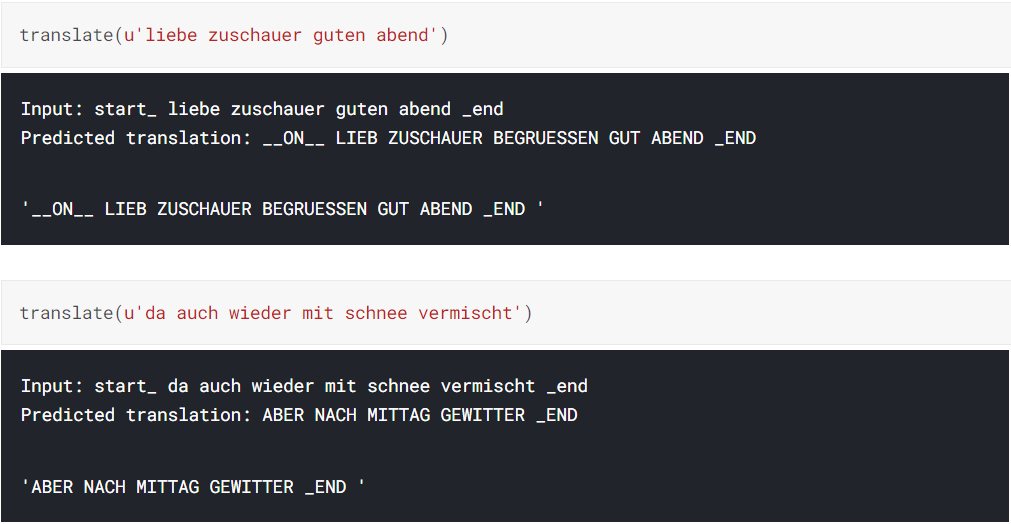
plt.show()

**CHAPTER 4**

**RESULTS & SNAPSHOTS**

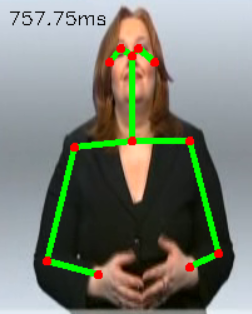
**4.1 Inference**

In our approach we trained our Neural Machine Translation model with learning rate 0.0001 using Adam optimizer and got some decent results. We used four layer Gated Recurrent Units(GRUs) which gives fine results in converting spoken language sentences to sign glosses.



## Fig 7 : NMT Output executed in kaggle.com

These sign glosses are mapped with respective sign language image sequences and corresponding gloss-image pairs are obtained. For every image we get motion primitives using open pose estimation model. And as the result we get corresponding gloss-pose pairs.



**Fig 8:** Motion Graphs on image

(Applying Open-pose estimation on images)

These gloss-pose pairs are pickled and given as training data noise for GAN. The GAN is trained on the original dataset images and are trained till the Generator fools the Discriminator more than 50% of the time. Finally, the pose noise is given as input to the trained Generator model and respective images are obtained for every gloss.



**Fig 9 :** Output of GAN

**CHAPTER 5**

**CONCLUSIONS AND FUTURE WORK**

## In our work we successfully translated spoken sentence to sign language image sequences. And the model is capable of producing output images sequence based on the input pose information given.

There is no proposed method till now to measure the accuracy of the generated sign language video. But, we can measure its accuracy using BLEU score. For that, we first have to give the video generated by GAN to a model which converts a sign language video to spoken sentence (this model is proposed by Camgoz et al in their one of their publications). The result will be a spoken language sentence. This resulted sentence and the sentence given to the NMT(the sentence for which gloss has been generated) will be compared and BLEU score will be calculated.

**CHAPTER 6**

**REFERENCES**

## Luong, T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation.

## Camgoz, N. C., Hadfield, S., Koller, O., Ney, H., & Bowden, R. (2018).

## Neural sign language translation. In IEEE Conference on computer vision and pattern recognition (CVPR).

## Wang, T. C., Liu, M. Y., Zhu, J. Y., Tao, A., Kautz, J., & Catanzaro, B. (2018b). High-resolution image synthesis and semantic manipulation with conditional GANs

## Kovar, L., Gleicher, M., & Pighin, F. (2002). Motion graphs. In Proceedings of the 29th annual conference on computer graphics and interactive techniques.