National University of Computer and Emerging Sciences - FAST Computer Science Department

Natural Language Processing

Usman Ali 16i-0164
M.Afaq 17i-0217
Section: B
Subject: Natural Language Processing
Date of submission: 15th June 2021
Submitted To: M.Arif

STUDENT SIGNATURES:

MARKS:

REMARKS:

TEACHERS SIGNATURE

Table of Contents

Contents

Introduction:

A language translation deep learning model has been developed by the team. The model is capable of translating **English** text to **Urdu** text to a high degree but not completely. This project was chosen to contribute to a resource-poor language known as **Urdu.** The translation is also somewhat dependent on whether the dataset had the input words or not. This is a very sought-after project.

Motivation:

There is a lot of work done in other languages in Natural Language Processing. Languages such as English, Hindi and Telugu enjoy the status of resource rich languages. Hence a lot of research work can be easily done in these languages whereas Urdu lags behind.
The developed model can easily translate English text to Urdu text however it cannot cater for large paragraphs and English words that are not part of the training dataset. Hence sometimes the output is composed of broken Urdu text yet readable by a native person.

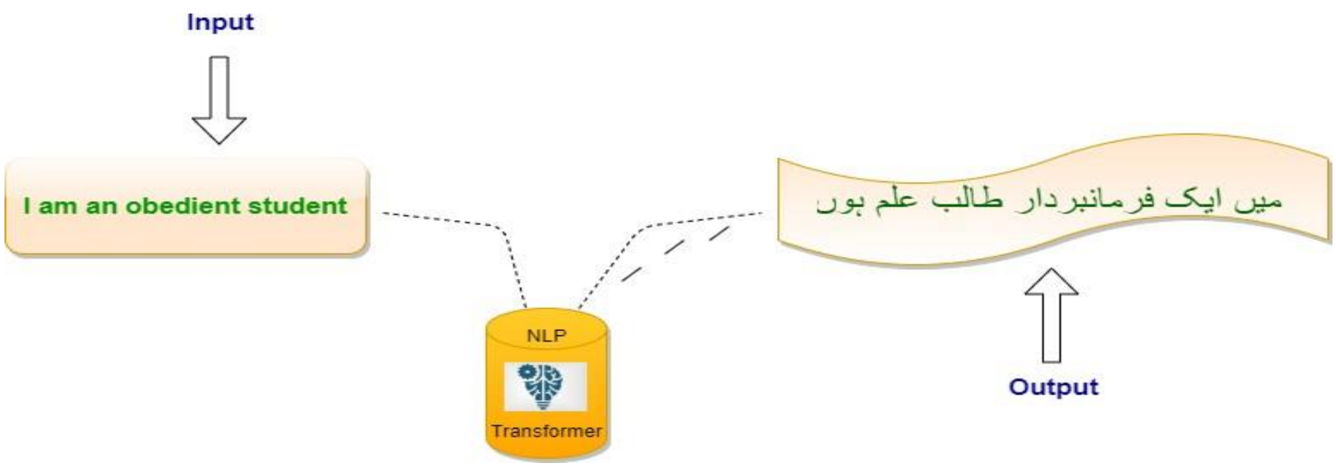


Figure 1: Working of our Model

Background and Research:

Since there are not many resources available online with regards to any such effort as undertaken in this project, many of the resources need to be obtained from sources that may not allow any disclosure of information regarding them.

There are no major translation models available as of 13th June 2021 in the domain of English to Urdu languages online. There are also no datasets available or any related previous work that is shared online.

Only a tokenizer (an object that breaks sentences into individual words) in **SpaCy** (a library that supports many languages) has been developed. This may help us in reading any datasets that are obtained. URDUNLP,[Urdu Tokenization using SpaCy (urdunlp.com),6th June 2021].

Project Specification:

The initial project under discussion was to develop a Deep Learning model that would be capable of differentiating whether a sentence has positive sentiment or negative sentiment (Sentiment means the view or opinion).
For example:
"I feel great this morning" is a sentence with positive sentiment.
"I feel tired this morning" is a sentence with negative sentiment.

The proposed model was to be able to analyze the sentiments of the languages of "English, Roman Urdu and Urdu".

Reasoning:

This topic was chosen because there seems to be very little work done on the language of **Urdu** and **Roman Urdu** in the field of Natural Language Processing. **English** was added so that more and more people would get to know about the language of **Urdu** and **Roman Urdu.** Even today, there are very few resources (specifically datasets and other relevant materials) in NLP (Natural Language Processing) available in these two languages.
One might say that this project was attempted to promote more work in these languages.

Revised Aims:

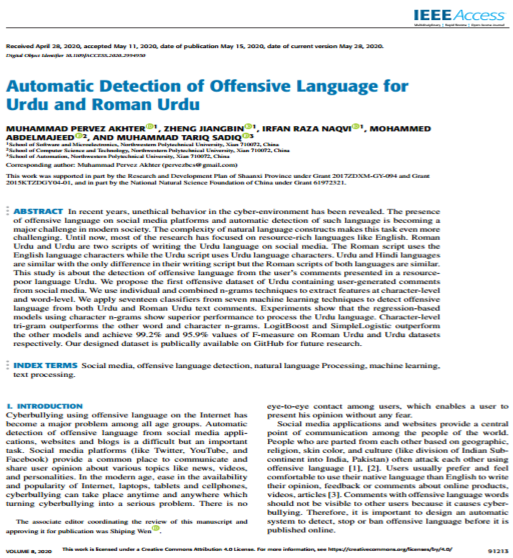
The above-mentioned details were the initial aims of this project but after careful consideration and review of available resources for the proposed project--the following changes were made

A Deep Learning model will be developed that will be able to convert English language text input to Urdu language text output. This project was chosen because **significant (not complete)** resources were available to make it a success and some contribution could also be made to the "Urdu language" in this regard.

There was also the problem of space constraints that we faced when coding our project on Google Colab. The initial project guidance was taken from a research paper. The paper is attached below.

The paper followed the working of "fastText" model developed by Facebook. Hence it uses n-grams to work. Colab only provides 12 GB space and when we developed this n-gram matrix. It was a sparse matrix. Whenever we wanted to convert it to a dense matrix, Colab crashed due to complete use of space.

https://ieeexplore.ieee.org/document/9094176/



IEEE Access

Received April 28, 2020, accepted May 11, 2020, date of publication May 15, 2020, date of current version May 28, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2994950

Automatic Detection of Offensive Language for Urdu and Roman Urdu

MUHAMMAD PERVEZ AKHTER[1], ZHENG JIANGBIN[1], IRFAN RAZA NAQVI[1], MOHAMMED ABDELMAJEED[2], AND MUHAMMAD TARIQ SADIQ[3]

[1]School of Software and Microelectronics, Northwestern Polytechnical University, Xian 710072, China
[2]School of Computer Science and Technology, Northwestern Polytechnical University, Xian 710072, China
[3]School of Automation, Northwestern Polytechnical University, Xian 710072, China

Corresponding author: Muhammad Pervez Akhter (pervezbcs@gmail.com)

This work was supported in part by the Research and Development Plan of Shaanxi Province under Grant 2017ZDXM-GY-094 and Grant 2015KTZDGY04-01, and in part by the National Natural Science Foundation of China under Grant 61972321.

ABSTRACT In recent years, unethical behavior in the cyber-environment has been revealed. The presence of offensive language on social media platforms and automatic detection of such language is becoming a major challenge in modern society. The complexity of natural language constructs makes this task even more challenging. Until now, most of the research has focused on resource-rich languages like English. Roman Urdu and Urdu are two scripts of writing the Urdu language on social media. The Roman script uses the English language characters while the Urdu script uses Urdu language characters. Urdu and Hindi languages are similar with the only difference in their writing script but the Roman scripts of both languages are similar. This study is about the detection of offensive language from the user's comments presented in a resource-poor language Urdu. We propose the first offensive dataset of Urdu containing user-generated comments from social media. We use individual and combined n-grams techniques to extract features at character-level and word-level. We apply seventeen classifiers from seven machine learning techniques to detect offensive language from both Urdu and Roman Urdu text comments. Experiments show that the regression-based models using character n-grams show superior performance to process the Urdu language. Character-level tri-gram outperforms the other word and character n-grams. LogitBoost and SimpleLogistic outperform the other models and achieve 99.2% and 95.9% values of F-measure on Roman Urdu and Urdu datasets respectively. Our designed dataset is publically available on GitHub for future research.

INDEX TERMS Social media, offensive language detection, natural language Processing, machine learning, text processing.

I. INTRODUCTION

Cyberbullying using offensive language on the Internet has become a major problem among all age groups. Automatic detection of offensive language from social media applications, websites and blogs is a difficult but an important task. Social media platforms (like Twitter, YouTube, and Facebook) provide a common place to communicate and share user opinion about various topics like news, videos, and personalities. In the modern age, ease in the availability and popularity of Internet, laptops, tablets and cellphones, cyberbullying can take place anytime and anywhere which turning cyberbullying into a serious problem. There is no eye-to-eye contact among users, which enables a user to present his opinion without any fear.

Social media applications and websites provide a central point of communication among the people of the world. People who are parted from each other based on geographic, religion, skin color, and culture (like division of Indian Sub-continent into India, Pakistan) often attack each other using offensive language [1], [2]. Users usually prefer and feel comfortable to use their native language than English to write their opinion, feedback or comments about online products, videos, articles [3]. Comments with offensive language words should not be visible to other users because it causes cyber-bullying. Therefore, it is important to design an automatic system to detect, stop or ban offensive language before it is published online.

The associate editor coordinating the review of this manuscript and approving it for publication was Shiping Wen.

 This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

A summary of what was to be produced and in what way.

Table 1: Table of summery

	Best Methods from research papers	Explanation
English	Word Bi-grams for classification using J48 graft classifier	Our proposed approach reaches an accuracy equal to 87.4% for the binary classification of tweets into offensive and non-offensive
Roman Urdu	Character Tri-grams for classification using regression algorithm LogitBoost	LogitBoost shows superior performance on Roman Urdu using character tri-gram and achieved 99.2% score of F-measure.
Urdu	Word Tri-Grams/Unigrams for classification using regression algorithm SimpleLogistic. *IT MAY BE NOTED THAT WE HAVE SPECIFIED WORD N-GRAMS AS OPPOSED TO CHARACTER N-GRAMS DUE TO UNAVAILIBITY OF PARSERS IN NASTALEEQ FONT OF URDU.*	SimpleLogistic outperforms the others classifiers using character tri-gram on Urdu dataset and achieved 95.8% F-measure value.

Problem Analysis:

The intended model to be developed needs some datasets that it may be trained on. However, there are no datasets available online.

To read that dataset is the second problem, we need a tokenizer (an object that is capable of reading text and breaking a sentence into words) of English and Urdu.

The words then need to be encoded in numerical form so that a deep learning model can understand it and train (learn it) on it and then give an encoded output.

This encoded output will then be decoded to produce human readable text in Urdu language.

6.1. Choosing a Model:

A deep learning model is to be developed that is capable of converting English input text to Urdu output text. There are a number of methods that can be employed to achieve this. They are

1. RNNs (Recurrent Neural Network)
2. LSTM (Long Short Term Memory)

6.2. Solution Design:

Datasets were obtained from sources that are not available for public disclosure.
There are a number of problems which a deep learning engineer may face when training a RNN for language learning. We will not dive deep into the technicalities of it. One of the major reasons to develop a LSTM model was to cover the drawbacks of a RNN model.
There are also some drawbacks of training on the LSTM model. The LSTM model cannot cater for long sentences, a problem that the RNN also faced. It does however cater for the vanishing and exploding gradients or “weights” of the RNN. Hence the name, Long Short-Term Memory.

An LSTM model has been chosen as the dataset is much more suited for it rather than an RNN or Transformer.

Now that a model has been finalized, we have to go into the details of how exactly we will be designing a solution for this problem. But first, we shall take a brief overview of both RNN and LSTM working

6.3. RNN

The information in an RNN cycles via a loop. When it makes a judgement, it takes into account the current input as well as what it has learnt from prior inputs. Consider a regular feed-forward neural network that receives the word "neuron" as an input and analyses it character by character. By the time it gets to the letter "r," it has already forgotten about the letters "n," "e," and "u," making it nearly difficult for this sort of neural network to anticipate which character will appear next.

A recurrent neural network, however, is able to remember those characters because of its internal memory. It produces output, copies that output and loops it back into the network **(Donges, 2019).**

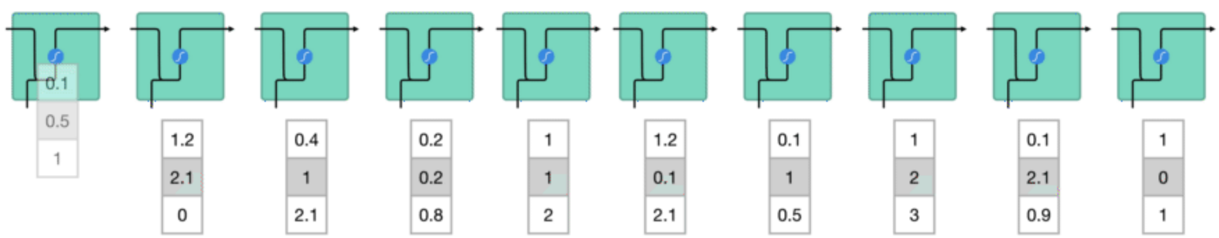
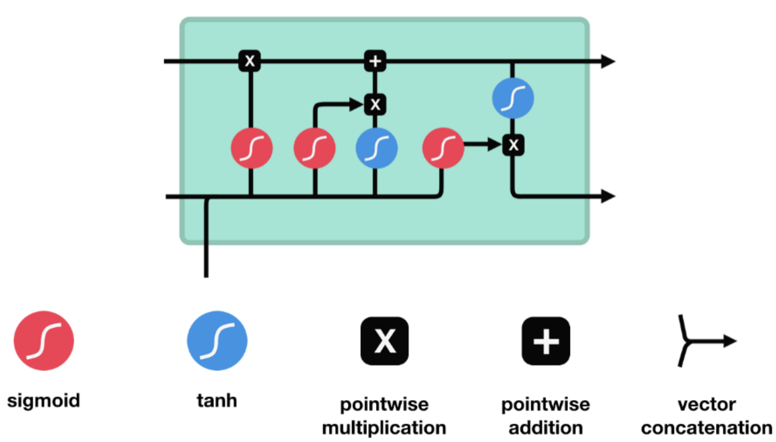


Figure 2: RNN

6.4. LSTM

The Extended Short Term Memory architecture was inspired by a study of error flow in current RNNs, which revealed that previous designs couldn't handle long time delays because backpropagated error either explodes or decays exponentially.

Figure 3: LSTM

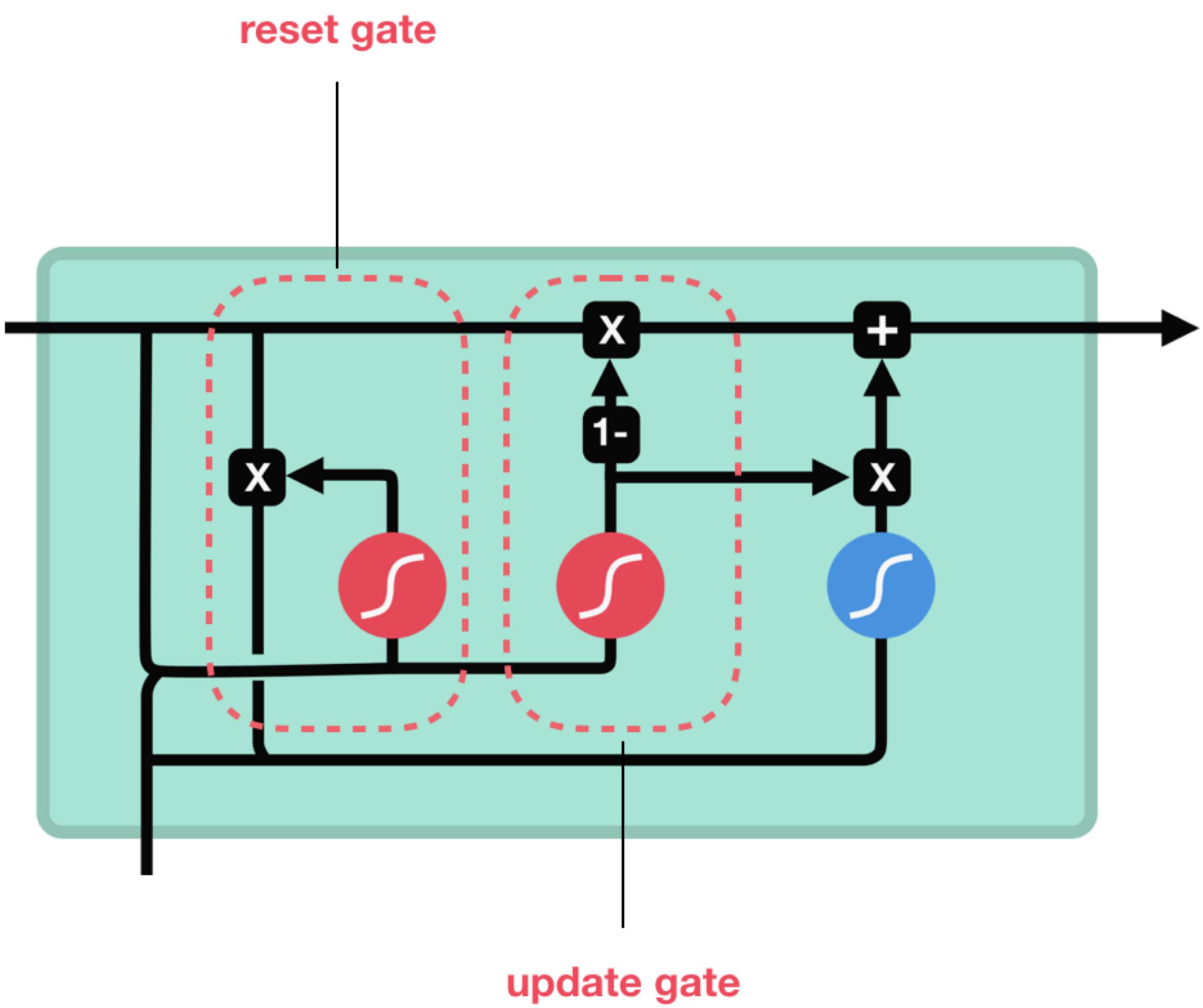


An LSTM layer is made up of memory blocks, which are recurrently linked blocks. These blocks can be thought of as a differentiable version of a digital computer's memory chips. Each one has one or more recurrently linked memory cells as well as three multiplicative units – the input, output, and forget gates – that offer continuous analogues of write, read, and reset operations for the input, output, and forget gates **(Brownlee, 2017).**

6.5. GRU

The GRU is the newer generation of Recurrent Neural networks and is pretty similar to an LSTM. GRU's got rid of the cell state and used the hidden state to transfer information. It also only has two gates, a reset gate and update gate.

Figure 4: GRU



6.6. Attention:

The attention mechanism to overcome the limitation that allows the network to learn where to pay attention in the input sequence for each item in the output sequence.
Each time the proposed model generates a word in a translation, it searches for a set of positions in a source sentence where the most relevant information is concentrated. The model then predicts a target word based on the context vectors associated with these source positions and all the previous generated target words

Loss per epoch:

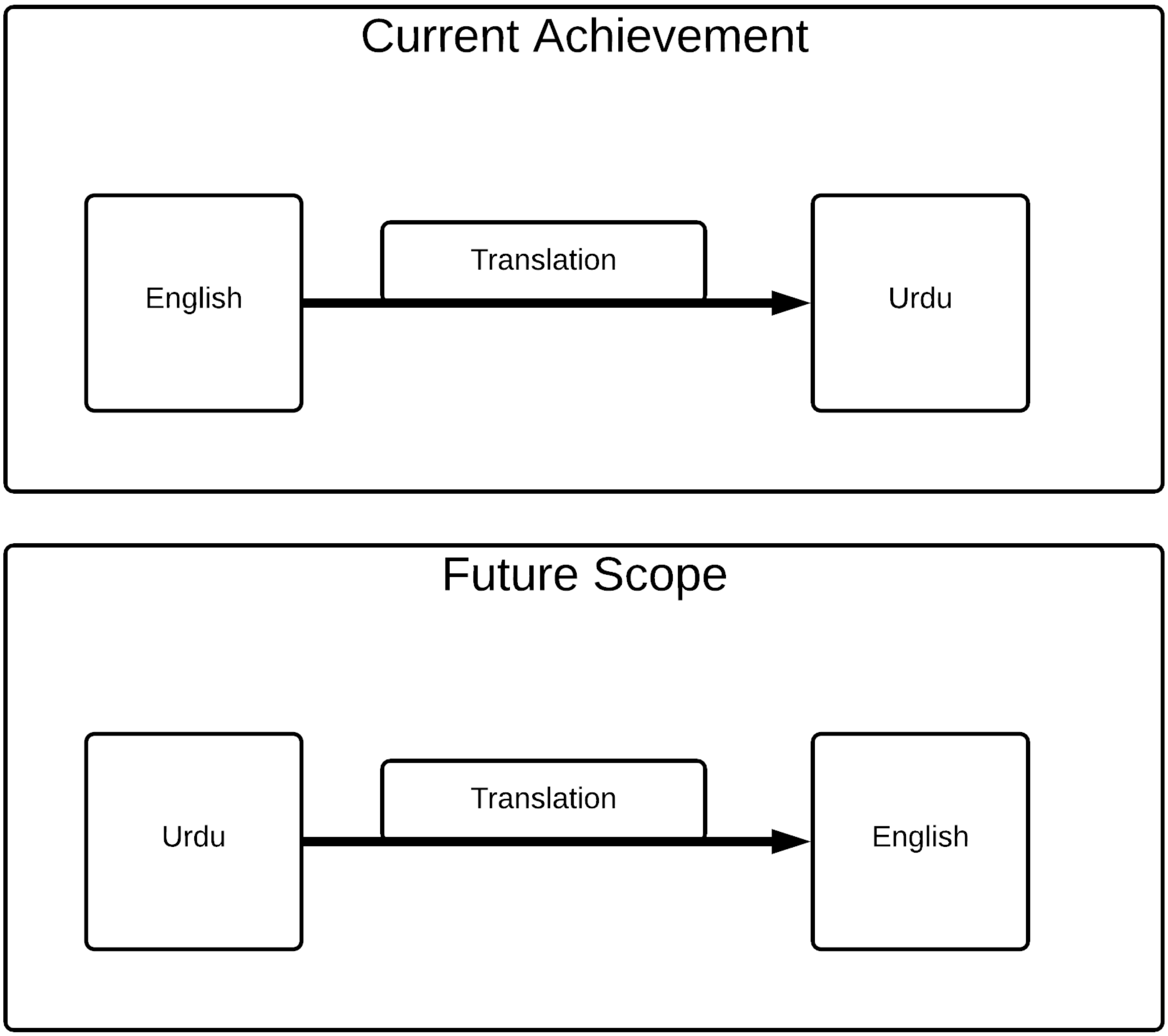
Figure 5: Loss per epoch

```
Epoch 1 Batch 0 loss 2.6736228466033936
Epoch 1 Loss 2.2056
Time taken for 1 epoch 25.624637842178345 sec

Epoch 2 Batch 0 loss 2.1453299522399902
Epoch 2 Loss 1.8550
Time taken for 1 epoch 25.578235626220703 sec

Epoch 3 Batch 0 loss 1.8335893154144287
Epoch 3 Loss 1.7099
Time taken for 1 epoch 25.127986907958984 sec

Epoch 4 Batch 0 loss 1.3724244832992554
Epoch 4 Loss 1.5726
Time taken for 1 epoch 25.6191086769104 sec

Epoch 5 Batch 0 loss 1.2512270212173462
Epoch 5 Loss 1.4426
Time taken for 1 epoch 25.247379541397095 sec

Epoch 6 Batch 0 loss 1.089025616645813
Epoch 6 Loss 1.2969
Time taken for 1 epoch 25.64974546432495 sec

Epoch 7 Batch 0 loss 1.1577202081680298
Epoch 7 Loss 1.1641
Time taken for 1 epoch 25.068135499954224 sec

Epoch 8 Batch 0 loss 1.1985244750976562
Epoch 8 Loss 1.0187
Time taken for 1 epoch 25.795782804489136 sec
```

Achievements

Our goal was to make English to Urdu translator. The target we achieved can best described by the figure below:

Figure 6: Our work



Data Collection

The data we collected is not available to the general public, its link comes back with the 404 error (page not found)

Solution

The solution to our problem is our trained python program using the Natural Language Processing techniques with the help of a dataset. We actually, provide the input as a text in English and it produce the Urdu translation of that text as output.

Journey to Solution

10.1. Programming Language

- Python

10.2. Machine Learning Library

- KERAS (coded on TensorFlow framework)

10.3. Neural Network

- LSTM + GRU

10.4. Dataset

English – Urdu from https://www.manythings.org/anki/urd-eng.zip/ (now removed)

10.5. Compiler or Tool

Google Colab

The Implementation

```
import pandas as pd
import numpy as np
import string
from string import digits
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
import matplotlib.ticker as ticker
from sklearn.model_selection import train_test_split
import re
import os
import io
import time
# Load the Drive helper and mount
from google.colab import drive

# This will prompt for authorization.
#drive.mount('/content/drive/Assignment4DL/urd.txt')


data_path = "/content/urd.txt"#Read the data

#Read the data
lines_raw= pd.read_table(data_path,names=['source', 'target'])
lines_raw.sample(5)


def preprocess_sentence(sentence):
    #sentence = unicode_to_ascii(sentence.lower().strip())
    num_digits= str.maketrans('','', digits)

    sentence= sentence.lower()
    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?"
print(preprocess_sentence(en_sentence))
```

Raw

```
import pandas as pd
import numpy as np
import string
from string import digits
import matplotlib.pyplot as plt
%matplotlib inline
import tensorflow as tf
import matplotlib.ticker as ticker
from sklearn.model_selection import train_test_split
import re
import os
import io
import time
# Load the Drive helper and mount
from google.colab import drive

# This will prompt for authorization.
#drive.mount('/content/drive/Assignment4DL/urd.txt')


data_path = "/content/urd.txt"#Read the data

#Read the data
lines_raw= pd.read_table(data_path,names=['source', 'target'])
lines_raw.sample(5)


def preprocess_sentence(sentence):
    #sentence = unicode_to_ascii(sentence.lower().strip())
    num_digits= str.maketrans('','', digits)

    sentence= sentence.lower()
    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?"
print(preprocess_sentence(en_sentence))
```

```
print(preprocess_sentence('Can you do it in thirty minutes?'))

# 1. Remove the accents
# 2. Clean the sentences
# 3. Return word pairs in the format: [ENGLISH, Urdu]
def create_dataset(path, num_examples):

  lines = io.open(path, encoding='UTF-8').read().strip().split('\n')
  #print(lines)
  word_pairs = [[preprocess_sentence(w) for w in l.split('\t')]  for l in lines[
:num_examples]]
  print(path)
  return zip(*word_pairs)

sample_size=1176
source, target = create_dataset(data_path, sample_size)
print("Making word pairs here")
print(source[-1])
print(target[-1])
type(target)

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,paddi
ng='post' )

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,paddi
ng='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)

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 = t
rain_test_split(source_tensor, target_tensor, test_size=0.2)

# Show length
print(len(input_tensor_train), len(target_tensor_train), len(input_tensor_val),
len(target_tensor_val))

type(input_tensor_train)

def convert(lang, tensor):
  for t in tensor:
    if t!=0:
      print ("%d ----> %s" % (t, lang.index_word[t]))

print ("Input Language; index to word mapping")
convert(source_sentence_tokenizer, source_train_tensor[0])
print ()
print ("Target Language; index to word mapping")
convert( target_sentence_tokenizer, target_train_tensor[0])

BUFFER_SIZE = len(source_train_tensor)
BATCH_SIZE = 12
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=False)
type(dataset)

source_batch, target_batch = next(iter(dataset))
source_batch.shape, target_batch.shape

class Encoder(tf.keras.Model):
    def __init__(self, vocab_size, embedding_dim, encoder_units, batch_size):
        super(Encoder, self).__init__()
        self.batch_size= batch_size
        self.encoder_units=encoder_units
        self.embedding=tf.keras.layers.Embedding(vocab_size, embedding_dim)
        self.gru= tf.keras.layers.GRU(encoder_units, return_sequences=True, retu
rn_state=True, recurrent_initializer='glorot_uniform')

    def call(self, x, hidden):
```

```
        #pass the input x to the embedding layer
        x= self.embedding(x)
        # pass the embedding and the hidden state to GRU
        output, state = self.gru(x, initial_state=hidden)
        return output, state

    def initialize_hidden_state(self):
        return tf.zeros((self.batch_size, self.encoder_units))

encoder = Encoder(vocab_inp_size, embedding_dim, units, BATCH_SIZE)

# sample input
sample_hidden = encoder.initialize_hidden_state()
sample_output, sample_hidden = encoder(source_batch, sample_hidden)
print ('Encoder output shape: (batch size, sequence length, units) {}'.format(sa
mple_output.shape))
print ('Encoder Hidden state shape: (batch size, units) {}'.format(sample_hidden
.shape))

class LSTM(tf.keras.layers.Layer):
  def __init__(self, units):
    super(LSTM, 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



attention_layer = LSTM(10)
attention_result, attention_weights = attention_layer(sample_hidden, sample_outp
ut)

print("Attention result shape: (batch size, units) {}".format(attention_result.s
hape))
print("Attention weights shape: (batch_size, sequence_length, 1) {}".format(atte
ntion_weights.shape))

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.fc = tf.keras.layers.Dense(vocab_size)

    # used for attention
    self.attention = LSTM(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 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

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))

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_)

checkpoint_dir = 'training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "ckpt")
checkpoint = tf.train.Checkpoint(optimizer=optimizer,
                                 encoder=encoder,
                                 decoder=decoder)

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

EPOCHS = 20

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

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_lengt
h,
                                                         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)

    # storing the attention weights to plot later on
    attention_weights = tf.reshape(attention_weights, (-1, ))
    attention_plot[t] = attention_weights.numpy()

    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

# 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()

def translate(sentence):
  result, sentence, attention_plot = evaluate(sentence)

  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(' '))

# restoring the latest checkpoint in checkpoint_dir
checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))



translate(u'Stay thin.')
```

References

- *Python Best for Machine Learning*. (2019). Towards datascience. https://towardsdatascience.com/8-reasons-why-python-is-good-for-artificial-intelligence-and-machine-learning-4a23f6bed2e6
- *RNN*. (2019). Builtin.Com. https://builtin.com/data-science/recurrent-neural-networks-and-lstm
- *LSTM*. (2017). Machinelearningmastery.Com. https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/
- Phi, M., 2021. *Illustrated Guide to LSTM's and GRU's: A step by step explanation*. [online] Medium. Available at: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-

gru-s-a-step-by-step-explanation-44e9eb85bf21> [Accessed 15 June 2021].

- Britz, D., 2021. *Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano*. [online] WildML. Available at: <http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano/> [Accessed 15 June 2021].