1. Imports and Setup

We import all necessary libraries:

- For mathematical operations (`math`, `numpy`),
- For data handling (`csv`, `os`, `random`),
- For plotting (`matplotlib`),
- For building deep learning models (`tensorflow`, `tensorflow_addons`),
- For working with Recurrent Neural Networks (RNNs), LSTM cells, attention mechanisms, and dense layers.

```
%matplotlib inline

import math

import numpy as np

import os

import random

import tensorflow as tf

from matplotlib import pylab

from collections import Counter

import csv



# Seq2Seq Items

from tensorflow.python.ops.rnn_cell import LSTMCell

from tensorflow.python.ops.rnn_cell import MultiRNNCell

import tensorflow_addons.seq2seq as seq2seq

import tensorflow_addons.seq2seq.attention_wrapper as attention_wrapper

from tensorflow.python.layers.core import Dense
```

2. Defining Model Hyperparameters

We set the main hyperparameters of our model:

- Vocabulary size for both source (German) and target (English) languages,
- Number of units (hidden size) in the LSTM cells,
- Input embedding size,
- Batch size for training,
- Sequence lengths for encoder and decoder,
- Decoder type (basic or attention-based),
- Number of sentence pairs to read for training.

```
vocab_size= 50000

num_units = 128

input_size = 128

batch_size = 16

source_sequence_length=40

target_sequence_length=60

decoder_type = 'basic' # could be basic or attention

sentences_to_read = 50000
```

3. Loading Vocabulary Files

We load the vocabularies for both German (source) and English (target) languages.

- Each word is mapped to a unique integer index.
- Reverse dictionaries (index to word) are also created for easy decoding and visualization later.

```
src_dictionary = dict()
with open('vocab.50K.de.txt', encoding='utf-8') as f:
   for line in f:
       #we are discarding last char as it is new line char
       src_dictionary[line[:-1]] = len(src_dictionary)


src_reverse_dictionary =
dict(zip(src_dictionary.values(),src_dictionary.keys()))


print('Source')
print('\t',list(src_dictionary.items())[:10])
print('\t',list(src_reverse_dictionary.items())[:10])
print('\t','Vocabulary size: ', len(src_dictionary))


tgt_dictionary = dict()
with open('vocab.50K.en.txt', encoding='utf-8') as f:
   for line in f:
       #we are discarding last char as it is new line char
       tgt_dictionary[line[:-1]] = len(tgt_dictionary)


tgt_reverse_dictionary =
dict(zip(tgt_dictionary.values(),tgt_dictionary.keys()))
```

```
print('Target')

print('\t',list(tgt_dictionary.items())[:10])

print('\t',list(tgt_reverse_dictionary.items())[:10])

print('\t','Vocabulary size: ', len(tgt_dictionary))
```

---

4. Loading Training Sentences

We load the German-English training sentence pairs from files.

- Skip the first 50 lines to avoid noisy translations.
- Load exactly 50,000 sentences for both source and target datasets.
- Ensure the source and target datasets are perfectly aligned (same number of sentences).

```
source_sent = []

target_sent = []


test_source_sent = []

test_target_sent = []




with open('train.de', encoding='utf-8') as f:

   for l_i, line in enumerate(f):

       # discarding first 20 translations as there was some

       # english to english translations found in the first few. which
are wrong

       if l_i<50:
```

```
            continue

        source_sent.append(line)

        if len(source_sent)>=sentences_to_read:

            break




with open('train.en', encoding='utf-8') as f:

    for l_i, line in enumerate(f):

        if l_i<50:

            continue


        target_sent.append(line)

        if len(target_sent)>=sentences_to_read:

            break




assert len(source_sent)==len(target_sent),'Source: %d, Target: 
%d'%(len(source_sent),len(target_sent))


print('Sample translations (%d)'%len(source_sent))

for i in range(0,sentences_to_read,10000):

    print('(',i,') DE: ', source_sent[i])

    print('(',i,') EN: ', target_sent[i])
```

5. Analyzing Sentence Statistics

We preprocess each sentence by:

- Separating punctuation from words,
- Replacing unknown words with a special `<unk>` token.

We also calculate:

- The average length and standard deviation of sentence lengths for both source and target datasets.
- This helps understand sentence distributions and set padding/truncation limits.

```
def split_to_tokens(sent,is_source):
    #sent = sent.replace('-',' ')
    sent = sent.replace(',',' ,')
    sent = sent.replace('.',' .')
    sent = sent.replace('\n',' ')

    sent_toks = sent.split(' ')
    for t_i, tok in enumerate(sent_toks):
        if is_source:
            if tok not in src_dictionary.keys():
                sent_toks[t_i] = '<unk>'
        else:
            if tok not in tgt_dictionary.keys():
                sent_toks[t_i] = '<unk>'
    return sent_toks
```

```python
# Let us first look at some statistics of the sentences

source_len = []

source_mean, source_std = 0,0

for sent in source_sent:

   source_len.append(len(split_to_tokens(sent,True)))


print('(Source) Sentence mean length: ', np.mean(source_len))

print('(Source) Sentence stddev length: ', np.std(source_len))


target_len = []

target_mean, target_std = 0,0

for sent in target_sent:

   target_len.append(len(split_to_tokens(sent,False)))


print('(Target) Sentence mean length: ', np.mean(target_len))

print('(Target) Sentence stddev length: ', np.std(target_len))
```

---

6. Preparing Training Inputs and Outputs

We prepare the final dataset for training:

- Add special tokens like `<s>` (start) and `</s>` (end).
- Reverse source sentences to help the model learn better (common trick in Seq2Seq).
- Pad all sentences to fixed maximum lengths to enable efficient batch processing.
- Create input length arrays to keep track of the actual (unpadded) lengths of sentences.

```
train_inputs = []

train_outputs = []

train_inp_lengths = []

train_out_lengths = []



max_tgt_sent_lengths = 0



src_max_sent_length = 41

tgt_max_sent_length = 61

for s_i, (src_sent, tgt_sent) in
enumerate(zip(source_sent,target_sent)):



   src_sent_tokens = split_to_tokens(src_sent,True)

   tgt_sent_tokens = split_to_tokens(tgt_sent,False)



   num_src_sent = []

   for tok in src_sent_tokens:

       num_src_sent.append(src_dictionary[tok])



   num_src_set = num_src_sent[::-1] # we reverse the source sentence.
This improves performance

   num_src_sent.insert(0,src_dictionary['<s>'])


train_inp_lengths.append(min(len(num_src_sent)+1,src_max_sent_length))



   # append until the sentence reaches max length
```

```
    if len(num_src_sent)<src_max_sent_length:

        num_src_sent.extend([src_dictionary['</s>'] for _ in
range(src_max_sent_length - len(num_src_sent))])

    # if more than max length, truncate the sentence

    elif len(num_src_sent)>src_max_sent_length:

        num_src_sent = num_src_sent[:src_max_sent_length]

    assert len(num_src_sent)==src_max_sent_length,len(num_src_sent)



    train_inputs.append(num_src_sent)



    num_tgt_sent = [tgt_dictionary['</s>']]

    for tok in tgt_sent_tokens:

        num_tgt_sent.append(tgt_dictionary[tok])




train_out_lengths.append(min(len(num_tgt_sent)+1,tgt_max_sent_length))



    if len(num_tgt_sent)<tgt_max_sent_length:

        num_tgt_sent.extend([tgt_dictionary['</s>'] for _ in
range(tgt_max_sent_length - len(num_tgt_sent))])

    elif len(num_tgt_sent)>tgt_max_sent_length:

        num_tgt_sent = num_tgt_sent[:tgt_max_sent_length]



    train_outputs.append(num_tgt_sent)

    assert len(train_outputs[s_i])==tgt_max_sent_length, 'Sent length
needs to be 60, but is %d'%len(binned_outputs[s_i])
```

```
assert len(train_inputs)  == len(source_sent),\
       'Size of total bin elements: %d, Total sentences: %d'\
               %(len(train_inputs),len(source_sent))


print('Max sent lengths: ', max_tgt_sent_lengths)




train_inputs = np.array(train_inputs, dtype=np.int32)

train_outputs = np.array(train_outputs, dtype=np.int32)

train_inp_lengths = np.array(train_inp_lengths, dtype=np.int32)

train_out_lengths = np.array(train_out_lengths, dtype=np.int32)

print('Samples from bin')

print('\t',[src_reverse_dictionary[w]  for w in
train_inputs[0,:].tolist()])

print('\t',[tgt_reverse_dictionary[w]  for w in
train_outputs[0,:].tolist()])

print('\t',[src_reverse_dictionary[w]  for w in
train_inputs[10,:].tolist()])

print('\t',[tgt_reverse_dictionary[w]  for w in
train_outputs[10,:].tolist()])

print()

print('\tSentences ',train_inputs.shape[0])
```

---

7. Data Batching Class

We define a custom batch generator class.

- This class handles how batches are created for training.
- It efficiently unrolls the source and target sentences into sequential batches.
- It manages sentence cursors and resets after reaching the end of sentences.
- Word embeddings for German and English are loaded from pre-trained `.npy` files.

```
input_size = 128


class DataGeneratorMT(object):


   def __init__(self,batch_size,num_unroll,is_source):

       self._batch_size = batch_size

       self._num_unroll = num_unroll

       self._cursor = [0 for offset in range(self._batch_size)]



       self._src_word_embeddings = np.load('de-embeddings.npy')


       self._tgt_word_embeddings = np.load('en-embeddings.npy')


       self._sent_ids = None


       self._is_source = is_source



   def next_batch(self, sent_ids, first_set):
```

```python
if self._is_source:
    max_sent_length = src_max_sent_length
else:
    max_sent_length = tgt_max_sent_length
batch_labels_ind = []
batch_data = np.zeros((self._batch_size),dtype=np.float32)
batch_labels = np.zeros((self._batch_size),dtype=np.float32)

for b in range(self._batch_size):

    sent_id = sent_ids[b]

    if self._is_source:
        sent_text = train_inputs[sent_id]

        batch_data[b] = sent_text[self._cursor[b]]
        batch_labels[b]=sent_text[self._cursor[b]+1]

    else:
        sent_text = train_outputs[sent_id]

        batch_data[b] = sent_text[self._cursor[b]]
        batch_labels[b] = sent_text[self._cursor[b]+1]

    self._cursor[b] = (self._cursor[b]+1)%(max_sent_length-1)
```

```python
    return batch_data,batch_labels

def unroll_batches(self,sent_ids):

    if sent_ids is not None:

        self._sent_ids = sent_ids

        self._cursor = [0 for _ in range(self._batch_size)]

    unroll_data,unroll_labels = [],[]
    inp_lengths = None
    for ui in range(self._num_unroll):

        data, labels = self.next_batch(self._sent_ids, False)

        unroll_data.append(data)
        unroll_labels.append(labels)
        inp_lengths = train_inp_lengths[sent_ids]
    return unroll_data, unroll_labels, self._sent_ids, inp_lengths

def reset_indices(self):
    self._cursor = [0 for offset in range(self._batch_size)]
```

```
# Running a tiny set to see if the implementation correct

dg = DataGeneratorMT(batch_size=5,num_unroll=40,is_source=True)

u_data, u_labels, _, _ = dg.unroll_batches([0,1,2,3,4])


print('Source data')

for _, lbl in zip(u_data,u_labels):

   print([src_reverse_dictionary[w] for w in lbl.tolist()])




# Running a tiny set to see if the implementation correct

dg = DataGeneratorMT(batch_size=5,num_unroll=60,is_source=False)

u_data, u_labels, _, _ = dg.unroll_batches([0,2,3,4,5])

print('\nTarget data batch (first time)')

for d_i,(_, lbl) in enumerate(zip(u_data,u_labels)):

   #if d_i>5 and d_i < 35:

   #    continue


   print([tgt_reverse_dictionary[w] for w in lbl.tolist()])


print('\nTarget data batch (non-first time)')

u_data, u_labels, _, _ = dg.unroll_batches(None)

for d_i,(_, lbl) in enumerate(zip(u_data,u_labels)):


   #if d_i>5 and d_i < 35:

   #    continue
```

```
print([tgt_reverse_dictionary[w] for w in lbl.tolist()])
```

---

8. Input Placeholders, Embeddings, and Masks

We set up TensorFlow placeholders for:

- Encoder and decoder inputs,
- Decoder output labels,
- Masks for padding during loss computation.

Pre-trained word embeddings are used instead of training embeddings from scratch, saving time and improving translation quality.

```
tf.reset_default_graph()


enc_train_inputs = []

dec_train_inputs = []


# Need to use pre-trained word embeddings

encoder_emb_layer = tf.convert_to_tensor(np.load('de-embeddings.npy'))

decoder_emb_layer = tf.convert_to_tensor(np.load('en-embeddings.npy'))


# Defining unrolled training inputs

for ui in range(source_sequence_length):

   enc_train_inputs.append(tf.placeholder(tf.int32,
shape=[batch_size],name='enc_train_inputs_%d'%ui))
```

```
dec_train_labels=[]

dec_label_masks = []

for ui in range(target_sequence_length):

   dec_train_inputs.append(tf.placeholder(tf.int32,
shape=[batch_size],name='dec_train_inputs_%d'%ui))

   dec_train_labels.append(tf.placeholder(tf.int32,
shape=[batch_size],name='dec-train_outputs_%d'%ui))

   dec_label_masks.append(tf.placeholder(tf.float32,
shape=[batch_size],name='dec-label_masks_%d'%ui))



encoder_emb_inp = [tf.nn.embedding_lookup(encoder_emb_layer, src) for
src in enc_train_inputs]

encoder_emb_inp = tf.stack(encoder_emb_inp)



decoder_emb_inp = [tf.nn.embedding_lookup(decoder_emb_layer, src) for
src in dec_train_inputs]

decoder_emb_inp = tf.stack(decoder_emb_inp)



enc_train_inp_lengths = tf.placeholder(tf.int32,
shape=[batch_size],name='train_input_lengths')

dec_train_inp_lengths = tf.placeholder(tf.int32,
shape=[batch_size],name='train_output_lengths')
```

---

9. Encoder Architecture

We define the encoder as a basic LSTM network:

- It processes the input German sentences,
- Produces hidden states (`encoder_outputs`) and the final context (`encoder_state`),
- The encoder is dynamic, meaning it can handle variable sentence lengths.

```
encoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

initial_state = encoder_cell.zero_state(batch_size, dtype=tf.float32)

encoder_outputs, encoder_state = tf.nn.dynamic_rnn(
   encoder_cell, encoder_emb_inp, initial_state=initial_state,
   sequence_length=enc_train_inp_lengths,
   time_major=True, swap_memory=True)
```

---

10. Decoder Architecture

We define the decoder as another LSTM network:

- It uses the final encoder state to start decoding,
- It can be either a basic decoder or attention-based depending on the setting,
- A dense projection layer maps LSTM outputs to vocabulary logits (prediction probabilities).

```
# Build RNN cell
decoder_cell = tf.nn.rnn_cell.BasicLSTMCell(num_units)

projection_layer = Dense(units=vocab_size, use_bias=True)
```

```python
# Helper
helper = tf.contrib.seq2seq.TrainingHelper(
    decoder_emb_inp, [tgt_max_sent_length-1 for _ in range(batch_size)],
time_major=True)


# Decoder
if decoder_type == 'basic':
    decoder = tf.contrib.seq2seq.BasicDecoder(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)


elif decoder_type == 'attention':
    decoder = tf.contrib.seq2seq.BahdanauAttention(
        decoder_cell, helper, encoder_state,
        output_layer=projection_layer)


# Dynamic decoding
outputs, _, _ = tf.contrib.seq2seq.dynamic_decode(
    decoder, output_time_major=True,
    swap_memory=True
)
```

11. Loss Function and Optimization

We define the loss:

- Use cross-entropy loss between predicted and actual words.
- Apply masking so that padding tokens don't contribute to loss.

We also define two optimizers:

- **Adam** optimizer is used for faster convergence in early training.
- **SGD** optimizer is used later for fine-tuning.
- Gradient clipping is applied to prevent exploding gradients.

```
logits = outputs.rnn_output



crossent = tf.nn.sparse_softmax_cross_entropy_with_logits(

   labels=dec_train_labels, logits=logits)

loss = (tf.reduce_sum(crossent*tf.stack(dec_label_masks)) /
(batch_size*target_sequence_length))



train_prediction = outputs.sample_id
```

12. Training Loop

We run the training for 10001 steps:

- In each step, sample random sentence pairs,
- Pass them through encoder and decoder,
- Compute loss and optimize the model,

- Switch optimizer after 10000 steps for better stability,
- Every 250 steps, print actual vs predicted translations to monitor progress,
- Every 500 steps, log and reset the running average of the loss.

```
print('Defining Optimizer')

# Adam Optimizer. And gradient clipping.

global_step = tf.Variable(0, trainable=False)

inc_gstep = tf.assign(global_step,global_step + 1)

learning_rate = tf.train.exponential_decay(

   0.01, global_step, decay_steps=10, decay_rate=0.9, staircase=True)


with tf.variable_scope('Adam'):

   adam_optimizer = tf.train.AdamOptimizer(learning_rate)


adam_gradients, v = zip(*adam_optimizer.compute_gradients(loss))

adam_gradients, _ = tf.clip_by_global_norm(adam_gradients, 25.0)

adam_optimize = adam_optimizer.apply_gradients(zip(adam_gradients, v))


with tf.variable_scope('SGD'):

   sgd_optimizer = tf.train.GradientDescentOptimizer(learning_rate)


sgd_gradients, v = zip(*sgd_optimizer.compute_gradients(loss))

sgd_gradients, _ = tf.clip_by_global_norm(sgd_gradients, 25.0)

sgd_optimize = sgd_optimizer.apply_gradients(zip(sgd_gradients, v))



sess = tf.InteractiveSession()
```

```
if not os.path.exists('logs'):

   os.mkdir('logs')

log_dir = 'logs'


bleu_scores_over_time = []

loss_over_time = []

tf.global_variables_initializer().run()


src_word_embeddings = np.load('de-embeddings.npy')

tgt_word_embeddings = np.load('en-embeddings.npy')


# Defining data generators

enc_data_generator =
DataGeneratorMT(batch_size=batch_size,num_unroll=source_sequence_length
,is_source=True)

dec_data_generator =
DataGeneratorMT(batch_size=batch_size,num_unroll=target_sequence_length
,is_source=False)


num_steps = 10001

avg_loss = 0


bleu_labels, bleu_preds = [],[]


print('Started Training')


for step in range(num_steps):
```

```
    # input_sizes for each bin: [40]

    # output_sizes for each bin: [60]

    print('.',end='')

    if (step+1)%100==0:

        print('')


    sent_ids =
np.random.randint(low=0,high=train_inputs.shape[0],size=(batch_size))

    # ====================== ENCODER DATA COLLECTION
================================================


    eu_data, eu_labels, _, eu_lengths =
enc_data_generator.unroll_batches(sent_ids=sent_ids)


    feed_dict = {}

    feed_dict[enc_train_inp_lengths] = eu_lengths

    for ui,(dat,lbl) in enumerate(zip(eu_data,eu_labels)):

        feed_dict[enc_train_inputs[ui]] = dat


    # ====================== DECODER DATA COLLECITON
===========================

    # First step we change the ids in a batch

    du_data, du_labels, _, du_lengths =
dec_data_generator.unroll_batches(sent_ids=sent_ids)


    feed_dict[dec_train_inp_lengths] = du_lengths
```

```
    for ui,(dat,lbl) in enumerate(zip(du_data,du_labels)):

        feed_dict[dec_train_inputs[ui]] = dat

        feed_dict[dec_train_labels[ui]] = lbl

        feed_dict[dec_label_masks[ui]] = (np.array([ui for _ in 
range(batch_size)])<du_lengths).astype(np.int32)



    # ====================== OPTIMIZATION ==========================

    if step < 10000:

        _,l,tr_pred = sess.run([adam_optimize,loss,train_prediction], 
feed_dict=feed_dict)

    else:

        _,l,tr_pred = sess.run([sgd_optimize,loss,train_prediction], 
feed_dict=feed_dict)

    tr_pred = tr_pred.flatten()




    if (step+1)%250==0:



        print('Step ',step+1)



        print_str = 'Actual: '

        for w in 
np.concatenate(du_labels,axis=0)[::batch_size].tolist():

            print_str += tgt_reverse_dictionary[w] + ' '

            if tgt_reverse_dictionary[w] == '</s>':

                break
```

```
        print(print_str)

        print()



        print_str = 'Predicted: '

        for w in tr_pred[::batch_size].tolist():

            print_str += tgt_reverse_dictionary[w] + ' '

            if tgt_reverse_dictionary[w] == '</s>':

                break

        print(print_str)



        print('\n')



        rand_idx = np.random.randint(low=1,high=batch_size)

        print_str = 'Actual: '

        for w in
np.concatenate(du_labels,axis=0)[rand_idx::batch_size].tolist():

            print_str += tgt_reverse_dictionary[w] + ' '

            if tgt_reverse_dictionary[w] == '</s>':

                break

        print(print_str)




        print()

        print_str = 'Predicted: '

        for w in tr_pred[rand_idx::batch_size].tolist():
```

```
            print_str += tgt_reverse_dictionary[w] + ' '

            if tgt_reverse_dictionary[w] == '</s>':

                break

        print(print_str)

        print()


    avg_loss += l


    #sess.run(reset_train_state) # resetting hidden state for each batch


    if (step+1)%500==0:

        print('============= Step ', str(step+1), ' =============')

        print('\t Loss: ',avg_loss/500.0)


        loss_over_time.append(avg_loss/500.0)


        avg_loss = 0.0

        sess.run(inc_gstep)
```

Conclusion

This project trains a basic **Neural Machine Translation (NMT)** model that translates sentences from German to English using:

- LSTM-based Encoder-Decoder architecture,
- Pre-trained word embeddings,
- Proper batching, masking, and optimization techniques,
- Dynamic handling of variable sentence lengths.