Machine-Translation

April 25, 2020

1 HW5 Machine translation with Encoder-Decoder model

1.1 Due April 24th, 23:59

In this homework, you are first shown an example of encoder-decoder machine translation model for a dummy problem. Make sure you understand how it works. Then you will need to build a similar model for a real machine translation data set. The data set provided in this homework is an italiano-english dataset (perché italiano è mia lingua preferita), but feel free to download your preferred language pari here (http://www.manythings.org/anki/).

You are given the following files: - `Machine-Translation.ipynb`: This notebook file - `ita.txt`: Training dataset (see http://www.manythings.org/anki/ to understand the structure) - `utils/`: folder containing all utility code for the series of homeworks

1.1.1 Deliverables (zip them all)

- pdf or html version of your final notebook
- Show some translation examples in your notebook
- writeup.pdf: Add a short essay discussing the biggest challenges you encounter during this assignment and what you have learnt.

(**You are encouraged to add the writeup doc into your notebook using markdown/html langauge, just like how this notes is prepared**)

HW6 Write up

The dummy task

The date conversion task can get very high accracy, because it is very simple, even it can be described by some rules, so a NN can easily study its pattern, same as the own dummy task, But it is very useful, it let us understand the encoder-decoder stucture. It is suitable to solve the sequence generating problem with labeled data.(seqtoseq)

BLEU score

$BLEUscore = BP * exp(\frac{1}{N}\sum_i^4 log(p_i))$

$BP = min(1, e^{1-r/c})$

$p_i = \frac{\#of\ common\ ngram}{\#\ of\ total\ ngrams}$

Biggest challenge

First is the time

Second is the seed selection, for some seed, even the dummy task turns out to predict to be some weird value, I figured this out for quite some time. My own dummy "Adding number" does not converge as the date conversion example. I need more time to debug.

Third is the BLEU score in the translation problem. The vocab size and the training sample is large so it will take very long time to train. I did not implement it in this homework, I will definitely re-evaluate it.

2 Set up

In [1]:
```
#%load_ext autoreload
#%autoreload 2
%matplotlib inline

import os, sys
# add utils folder to path
p = os.path.dirname(os.getcwd())
if p not in sys.path:
    sys.path = [p] + sys.path

from utils.general import show_keras_model

from keras.models import Model
```

```
Using TensorFlow backend.
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:516:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:517:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:518:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:519:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:520:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/Library/Python/3.7/site-packages/tensorflow/python/framework/dtypes.py:525:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:541:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:542:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:543:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
```

```
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:544:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:545:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/Library/Python/3.7/site-packages/tensorboard/compat/tensorflow_stub/dtypes.py:550:
FutureWarning: Passing (type, 1) or '1type' as a synonym of type is deprecated; in a
future version of numpy, it will be understood as (type, (1,)) / '(1,)type'.
  np_resource = np.dtype([("resource", np.ubyte, 1)])
```

3 Dummy Translation Problem

We are not doing anything real here, rather, we create a dummy problem to demonstrate how easy or hard to use a S2S model for machine translation.

The dummy prblem I choose here is to translate datestr like "Aug-30-1989" to another format "1989/08/30". Sounds easy, isn't it? But think about it, you feel this simple because you have so much prior knowledge. You know the English meaning of "Aug", you know the different ways of representing dates, MM-DD-YYYY vs YYYY/MM/DD. But our model starts from absolute ignorance. Imagine you show this problem to a 2-year-old child, how much time does it make for him to figure out the rule?

3.1 Generate Training Data

```
In [2]: import numpy as np

        choice = np.random.choice
        def source_generation(batch=100):
            months = choice(['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep',
        'Oct', 'Nov', 'Dec'], batch)
            days = choice(range(1, 28), batch)
            years = choice(range(1990, 2050), batch)

            return [ f"{m}-{d}-{y}" for m, d, y in zip(months, days, years)]

        def translate(src):
            if type(src) == str: src = [src]
            mmap = {'Jan': '01', 'Feb': '02', 'Mar': '03', 'Apr': '04', 'May': '05', 'Jun':
        "06", 'Jul': "07",
                  'Aug': '08', 'Sep': '09', 'Oct': '10', 'Nov': '11', 'Dec': '12'}
            result = []
            for d in src:
                m, d, y = d.split('-')
                result.append(f"{y}/{mmap[m]}/{str(d).rjust(2, '0')}")

            return result
```

```
In [3]: # Let's generate some data
        train_X_raw = source_generation(10000)
        train_Y_raw = translate(train_X_raw)

        # Verify the translation
        print(train_X_raw[:5])
        print(train_Y_raw[:5])
```

```
['Feb-15-1999', 'Jul-12-1997', 'May-1-1996', 'Jul-7-2004', 'Jan-12-2003']
['1999/02/15', '1997/07/12', '1996/05/01', '2004/07/07', '2003/01/12']
```

3.2 Other dummy tasks

You are encouraged to generate your own dummy tasks, for example, what about a simple calculator, can you train your model to understand "186+95" equal to "281"?

4 Encoder-Decoder Model

In [4]:
```
encoder_input_len = 11
decoder_input_len = 10
latent_dim = 256
```

4.1 Raw data transformer

As of today, I guess you should be quite familar with what we are doing here.

In [5]:
```
from keras.preprocessing.sequence import pad_sequences

char_vocab = list('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-/0123456789$^')

reverse_vocab = {k:v for v, k in enumerate(char_vocab)}
def char_to_num(X_raw, is_encoder=True):
    """
    Translate the raw input to the numerical encoding. We take different treatments for
the
    encoder inputs and decoder inputs. This is because we need a starter character "^"
for the
    decoder inputs.
    """
    result = [[reverse_vocab[c] for c in sent] for sent in X_raw]

    if(is_encoder):
        assert all([len(row) <= encoder_input_len for row in X_raw])
        return pad_sequences(sequences=result, maxlen=encoder_input_len,
                             padding='post', truncating='post',
                             value=reverse_vocab['$'])
    else:
        assert all([len(row) == decoder_input_len for row in X_raw])
        return pad_sequences(sequences=result, maxlen=decoder_input_len+1,
                             padding='pre', truncating='post',
                             value=reverse_vocab['^'])

    return pad_sequences(result)

def num_to_char(X):
    return [''.join([char_vocab[c] for c in row]) for row in X]
```

4.2 Training model

In [7]:
```
# from keras.models import Model
from keras.layers import (Input, LSTM, Dense, Bidirectional, Embedding,
                          TimeDistributed, Concatenate)

"""
Define an input Layer. We use one-hot encoding instead of embedding layer here. Since
we are using character based model, embedding may not be necessary, and may not be very
helpful neither. Do you know why?
"""
encoder_inputs = Input(shape=(encoder_input_len, len(char_vocab)), name="Encoder_Input")
```

```
# For encoder, we can see the entire sentence at once, so we can use Bidirectional LSTM
encoder_lstm = Bidirectional(LSTM(latent_dim, return_state=True, name="Encoder_LSTM"))
# Bidrectional LSTM has 4 states instead of 2, we concatenate them to be comparable
# with the decoder LSTM
_, forward_h, forward_c, backward_h, backward_c = encoder_lstm(encoder_inputs)
state_h = Concatenate()([forward_h, backward_h])
state_c = Concatenate()([forward_c, backward_c])

# Set up the decoder, using `encoder_states` as initial state
encoder_states = [state_h, state_c]
decoder_inputs = Input(shape=(decoder_input_len, len(char_vocab)), name="Decoder_Input")
decoder_lstm = LSTM(latent_dim*2, return_sequences=True, name="Decoder_LSTM")
decoder_lstm_outputs = decoder_lstm(decoder_inputs,
                                    initial_state=encoder_states)
decoder_dense = Dense(len(char_vocab), activation='softmax')
decoder_outputs = TimeDistributed(decoder_dense)(decoder_lstm_outputs)

# Define the model that will turn
# `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

#show_keras_model(model)
```

4.3 Train training model

In [8]:
```
# Run training
from keras.utils import to_categorical
"""
Don't be suprized that this model actually needs quite quite a lot of epochs to train,
so please be patient.
After the model is trained, you can use the history.history object to plot the metrics
improvment process.

While you are waiting for the model to train, feel free to read the next cell.
"""
batch_size = 1000
epochs = 75

# Here it's just some data transformation to translate the raw data to matrix inputs
encoder_input_data = to_categorical(char_to_num(train_X_raw, True),
num_classes=len(char_vocab))
train_Y = to_categorical(char_to_num(train_Y_raw, False), num_classes=len(char_vocab))
# for decoder, the target lags input by 1 time step
decoder_input_data = train_Y[:, :-1, :]
decoder_target_data = train_Y[:, 1:, :]

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_split=0.2)
```

```
WARNING:tensorflow:From /Library/Python/3.7/site-
packages/tensorflow/python/ops/math_grad.py:1250:
add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is
deprecated and will be removed in a future version.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
WARNING:tensorflow:From /Library/Python/3.7/site-
packages/keras/backend/tensorflow_backend.py:422: The name tf.global_variables is
deprecated. Please use tf.compat.v1.global_variables instead.

Train on 8000 samples, validate on 2000 samples
Epoch 1/75
8000/8000 [==============================] - 13s 2ms/step - loss: 3.6607 - accuracy:
```

```
0.2740 - val_loss: 2.5431 - val_accuracy: 0.2531
Epoch 2/75
8000/8000 [==============================] - 12s 2ms/step - loss: 2.1796 - accuracy:
0.2801 - val_loss: 1.9859 - val_accuracy: 0.3609
Epoch 3/75
8000/8000 [==============================] - 12s 2ms/step - loss: 1.9008 - accuracy:
0.3690 - val_loss: 1.8111 - val_accuracy: 0.3898
Epoch 4/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.7766 - accuracy:
0.3910 - val_loss: 1.7258 - val_accuracy: 0.3862
Epoch 5/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.6913 - accuracy:
0.4035 - val_loss: 1.6522 - val_accuracy: 0.4229
Epoch 6/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.6111 - accuracy:
0.4383 - val_loss: 1.5579 - val_accuracy: 0.4516
Epoch 7/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.5021 - accuracy:
0.4871 - val_loss: 1.4221 - val_accuracy: 0.5482
Epoch 8/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.3444 - accuracy:
0.5765 - val_loss: 1.2367 - val_accuracy: 0.5903
Epoch 9/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.1462 - accuracy:
0.6006 - val_loss: 1.0399 - val_accuracy: 0.6077
Epoch 10/75
8000/8000 [==============================] - 11s 1ms/step - loss: 1.0108 - accuracy:
0.6065 - val_loss: 0.9779 - val_accuracy: 0.6130
Epoch 11/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.9700 - accuracy:
0.6113 - val_loss: 0.9573 - val_accuracy: 0.6117
Epoch 12/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.9458 - accuracy:
0.6162 - val_loss: 0.9306 - val_accuracy: 0.6253
Epoch 13/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.9204 - accuracy:
0.6305 - val_loss: 0.9197 - val_accuracy: 0.6289
Epoch 14/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.9003 - accuracy:
0.6389 - val_loss: 0.8852 - val_accuracy: 0.6529
Epoch 15/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.8723 - accuracy:
0.6553 - val_loss: 0.8564 - val_accuracy: 0.6632
Epoch 16/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.8476 - accuracy:
0.6733 - val_loss: 0.8331 - val_accuracy: 0.6835
Epoch 17/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.8236 - accuracy:
0.6858 - val_loss: 0.8085 - val_accuracy: 0.6939
Epoch 18/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.8032 - accuracy:
0.6945 - val_loss: 0.7835 - val_accuracy: 0.7058
Epoch 19/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.7767 - accuracy:
0.7060 - val_loss: 0.7599 - val_accuracy: 0.7207
Epoch 20/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.7499 - accuracy:
0.7207 - val_loss: 0.7434 - val_accuracy: 0.7213
Epoch 21/75
```

```
8000/8000 [==============================] - 11s 1ms/step - loss: 0.7275 - accuracy:
0.7254 - val_loss: 0.7092 - val_accuracy: 0.7337
Epoch 22/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.6965 - accuracy:
0.7339 - val_loss: 0.6784 - val_accuracy: 0.7437
Epoch 23/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.6616 - accuracy:
0.7480 - val_loss: 0.6438 - val_accuracy: 0.7527
Epoch 24/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.6344 - accuracy:
0.7547 - val_loss: 0.6243 - val_accuracy: 0.7526
Epoch 25/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.6112 - accuracy:
0.7644 - val_loss: 0.5971 - val_accuracy: 0.7681
Epoch 26/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.6082 - accuracy:
0.7600 - val_loss: 0.5787 - val_accuracy: 0.7763
Epoch 27/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.5756 - accuracy:
0.7787 - val_loss: 0.5582 - val_accuracy: 0.7890
Epoch 28/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.5529 - accuracy:
0.7931 - val_loss: 0.5345 - val_accuracy: 0.8036
Epoch 29/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.5364 - accuracy:
0.8023 - val_loss: 0.5313 - val_accuracy: 0.8005
Epoch 30/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.5154 - accuracy:
0.8112 - val_loss: 0.4965 - val_accuracy: 0.8227
Epoch 31/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4868 - accuracy:
0.8243 - val_loss: 0.4723 - val_accuracy: 0.8319
Epoch 32/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4610 - accuracy:
0.8361 - val_loss: 0.4526 - val_accuracy: 0.8368
Epoch 33/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4416 - accuracy:
0.8407 - val_loss: 0.4297 - val_accuracy: 0.8493
Epoch 34/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4213 - accuracy:
0.8503 - val_loss: 0.4105 - val_accuracy: 0.8561
Epoch 35/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.3969 - accuracy:
0.8619 - val_loss: 0.3892 - val_accuracy: 0.8669
Epoch 36/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.3737 - accuracy:
0.8727 - val_loss: 0.3650 - val_accuracy: 0.8738
Epoch 37/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4156 - accuracy:
0.8524 - val_loss: 0.5014 - val_accuracy: 0.8054
Epoch 38/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.4232 - accuracy:
0.8418 - val_loss: 0.3749 - val_accuracy: 0.8673
Epoch 39/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.3606 - accuracy:
0.8749 - val_loss: 0.3414 - val_accuracy: 0.8842
Epoch 40/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.3248 - accuracy:
0.8924 - val_loss: 0.3141 - val_accuracy: 0.8967
```

```
Epoch 41/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2977 - accuracy:
0.9050 - val_loss: 0.2900 - val_accuracy: 0.9050
Epoch 42/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2744 - accuracy:
0.9140 - val_loss: 0.2658 - val_accuracy: 0.9170
Epoch 43/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2518 - accuracy:
0.9214 - val_loss: 0.2513 - val_accuracy: 0.9191
Epoch 44/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2313 - accuracy:
0.9301 - val_loss: 0.2212 - val_accuracy: 0.9344
Epoch 45/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2219 - accuracy:
0.9320 - val_loss: 0.2288 - val_accuracy: 0.9263
Epoch 46/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.2269 - accuracy:
0.9252 - val_loss: 0.1975 - val_accuracy: 0.9424
Epoch 47/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.1946 - accuracy:
0.9422 - val_loss: 0.1870 - val_accuracy: 0.9409
Epoch 48/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.1737 - accuracy:
0.9515 - val_loss: 0.1717 - val_accuracy: 0.9526
Epoch 49/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.1526 - accuracy:
0.9624 - val_loss: 0.1471 - val_accuracy: 0.9603
Epoch 50/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.1349 - accuracy:
0.9686 - val_loss: 0.1225 - val_accuracy: 0.9754
Epoch 51/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.1147 - accuracy:
0.9774 - val_loss: 0.1068 - val_accuracy: 0.9812
Epoch 52/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0971 - accuracy:
0.9840 - val_loss: 0.0916 - val_accuracy: 0.9851
Epoch 53/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.1058 - accuracy:
0.9751 - val_loss: 0.0910 - val_accuracy: 0.9826
Epoch 54/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0893 - accuracy:
0.9835 - val_loss: 0.0785 - val_accuracy: 0.9877
Epoch 55/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0711 - accuracy:
0.9903 - val_loss: 0.0660 - val_accuracy: 0.9891
Epoch 56/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0592 - accuracy:
0.9936 - val_loss: 0.0550 - val_accuracy: 0.9937
Epoch 57/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0504 - accuracy:
0.9958 - val_loss: 0.0478 - val_accuracy: 0.9958
Epoch 58/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0437 - accuracy:
0.9974 - val_loss: 0.0420 - val_accuracy: 0.9972
Epoch 59/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0385 - accuracy:
0.9984 - val_loss: 0.0372 - val_accuracy: 0.9980
Epoch 60/75
8000/8000 [==============================] - 11s 1ms/step - loss: 0.0340 - accuracy:
```

```
0.9990 - val_loss: 0.0328 - val_accuracy: 0.9988
Epoch 61/75
8000/8000 [==============================] - 1884s 235ms/step - loss: 0.0301 -
accuracy: 0.9994 - val_loss: 0.0294 - val_accuracy: 0.9993
Epoch 62/75
8000/8000 [==============================] - 23s 3ms/step - loss: 0.0268 - accuracy:
0.9998 - val_loss: 0.0263 - val_accuracy: 0.9996
Epoch 63/75
8000/8000 [==============================] - 14s 2ms/step - loss: 0.0240 - accuracy:
0.9999 - val_loss: 0.0237 - val_accuracy: 0.9998
Epoch 64/75
8000/8000 [==============================] - 12s 2ms/step - loss: 0.0218 - accuracy:
0.9999 - val_loss: 0.0215 - val_accuracy: 0.9999
Epoch 65/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.0198 - accuracy:
0.9999 - val_loss: 0.0198 - val_accuracy: 1.0000
Epoch 66/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.0181 - accuracy:
1.0000 - val_loss: 0.0181 - val_accuracy: 0.9999
Epoch 67/75
8000/8000 [==============================] - 12s 2ms/step - loss: 0.0166 - accuracy:
1.0000 - val_loss: 0.0167 - val_accuracy: 0.9999
Epoch 68/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.0153 - accuracy:
1.0000 - val_loss: 0.0154 - val_accuracy: 1.0000
Epoch 69/75
8000/8000 [==============================] - 12s 1ms/step - loss: 0.0141 - accuracy:
1.0000 - val_loss: 0.0142 - val_accuracy: 1.0000
Epoch 70/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0130 - accuracy:
1.0000 - val_loss: 0.0133 - val_accuracy: 1.0000
Epoch 71/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0123 - accuracy:
1.0000 - val_loss: 0.0126 - val_accuracy: 1.0000
Epoch 72/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0115 - accuracy:
1.0000 - val_loss: 0.0115 - val_accuracy: 1.0000
Epoch 73/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0107 - accuracy:
1.0000 - val_loss: 0.0108 - val_accuracy: 1.0000
Epoch 74/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0099 - accuracy:
1.0000 - val_loss: 0.0102 - val_accuracy: 1.0000
Epoch 75/75
8000/8000 [==============================] - 13s 2ms/step - loss: 0.0093 - accuracy:
1.0000 - val_loss: 0.0095 - val_accuracy: 1.0000
```

4.4 Inference model

Similar to HW04, we need a different model structure for the inference model. The inference model should copy exactly the same weights from the training model, but it predicts only 1 time step at a time.

```
In [10]: # Trucate the encoder part of the training model as encoder model
         encoder_model = Model(encoder_inputs, encoder_states)
         #show_keras_model(encoder_model)
```

```
In [11]: # Build the inference model
```

```
inference_inputs = Input(batch_shape=(1,1, len(char_vocab)), name="Inference_Input")
inference_lstm = LSTM(latent_dim*2, stateful=True,
                      name="Inference_LSTM",)
inference_lstm_outputs = inference_lstm(inference_inputs)

inference_dense = Dense(len(char_vocab), activation='softmax')
inference_outputs = inference_dense(inference_lstm_outputs)

# Assign the weights of decoder to inference model
inference_lstm.set_weights(decoder_lstm.get_weights())
inference_dense.set_weights(decoder_dense.get_weights())

inference_model = Model(inference_inputs, inference_outputs)
#show_keras_model(inference_model)
```

In [12]:
```
def inference(encoder_input_data):
    """
    A utility function to generate the model prediction
    """
    states_h, states_c = encoder_model.predict(encoder_input_data)
    results = []
    inference_model.reset_states()
    for h, c in zip(states_h, states_c):
        sent, seed = [], reverse_vocab['^']
        inference_lstm.states[0].assign(h[None, :])
        inference_lstm.states[1].assign(c[None, :])
        for i in range(decoder_input_len):
            seed = to_categorical(np.array([seed]), num_classes=len(char_vocab))[None,
:, :]
            seed = inference_model.predict(seed)[0].argmax()
            sent.append(seed)

        results.append(sent)

    return num_to_char(results)
```

In [14]:
```
# Let's look at some output
print(num_to_char(encoder_input_data[:10].argmax(axis=2)))
print(translate(num_to_char(encoder_input_data[:10].argmax(axis=2))))
print(inference(encoder_input_data[:10]))
print(num_to_char(decoder_input_data[:10].argmax(axis=2)))
print(num_to_char(decoder_target_data[:10].argmax(axis=2)))
```

```
['Feb-15-1999', 'Jul-12-1997', 'May-1-1996$', 'Jul-7-2004$', 'Jan-12-2003',
'Oct-1-2017$', 'Jun-20-2030', 'Jan-5-1991$', 'May-23-2010', 'Jul-18-2036']
['1999/02/15', '1997/07/12', '1996$/05/01', '2004$/07/07', '2003/01/12',
'2017$/10/01', '2030/06/20', '1991$/01/05', '2010/05/23', '2036/07/18']
['201//18111', '111/11/11/', '1181181181', '1811811811', '8118118118', '1181181181',
'1811811811', '8118118118', '1181181181', '1811811811']
['^1999/02/1', '^1997/07/1', '^1996/05/0', '^2004/07/0', '^2003/01/1', '^2017/10/0',
'^2030/06/2', '^1991/01/0', '^2010/05/2', '^2036/07/1']
['1999/02/15', '1997/07/12', '1996/05/01', '2004/07/07', '2003/01/12', '2017/10/01',
'2030/06/20', '1991/01/05', '2010/05/23', '2036/07/18']
```

5 Own dummy Model

In [15]:
```
choice = np.random.choice
def source_generation(batch=100):
    a = choice(range(1,1000), batch)
    b = choice(range(1,1000), batch)

    return [f"{m}+{n}" for m,n in zip(a,b)]

def translate(src):
```

```
    if type(src) == str: src = [src]
    result = []
    for d in src:
        a,b = d.split('+')
        result.append(f"{int(a)+int(b)}".rjust(5,'0'))
    return result
```

In [16]:
```
# Let's generate some data
train_X_raw = source_generation(10000)
train_Y_raw = translate(train_X_raw)

# Verify the translation
print(train_X_raw[:5])
print(train_Y_raw[:5])
```

```
['513+83', '24+814', '978+758', '346+625', '466+166']
['00596', '00838', '01736', '00971', '00632']
```

In [17]:
```
encoder_input_len = 8
decoder_input_len = 5
latent_dim = 256
```

In [18]:
```
from keras.preprocessing.sequence import pad_sequences

char_vocab = list('0123456789$^+')

reverse_vocab = {k:v for v, k in enumerate(char_vocab)}
def char_to_num(X_raw, is_encoder=True):
    """
    Translate the raw input to the numerical encoding. We take different treatments for
the
    encoder inputs and decoder inputs. This is because we need a starter character "^"
for the
    decoder inputs.
    """
    result = [[reverse_vocab[c] for c in sent] for sent in X_raw]

    if(is_encoder):
        assert all([len(row) <= encoder_input_len for row in X_raw])
        return pad_sequences(sequences=result, maxlen=encoder_input_len,
                             padding='post', truncating='post',
                             value=reverse_vocab['$'])
    else:
        assert all([len(row) == decoder_input_len for row in X_raw])
        return pad_sequences(sequences=result, maxlen=decoder_input_len+1,
                             padding='pre', truncating='post',
                             value=reverse_vocab['^'])

    return pad_sequences(result)

def num_to_char(X):
    return [''.join([char_vocab[c] for c in row]) for row in X]
```

In [19]:
```
from keras.layers import (Input, LSTM, Dense, Bidirectional, Embedding,
                          TimeDistributed, Concatenate)

encoder_inputs = Input(shape=(encoder_input_len, len(char_vocab)), name="Encoder_Input")
# For encoder, we can see the entire sentence at once, so we can use Bidirectional LSTM
encoder_lstm = Bidirectional(LSTM(latent_dim, return_state=True, name="Encoder_LSTM"))
# Bidrectional LSTM has 4 states instead of 2, we concatenate them to be comparable
# with the decoder LSTM
_, forward_h, forward_c, backward_h, backward_c = encoder_lstm(encoder_inputs)
state_h = Concatenate()([forward_h, backward_h])
state_c = Concatenate()([forward_c, backward_c])

# Set up the decoder, using `encoder_states` as initial state
encoder_states = [state_h, state_c]
decoder_inputs = Input(shape=(decoder_input_len, len(char_vocab)), name="Decoder_Input")
```

```python
    decoder_lstm = LSTM(latent_dim*2, return_sequences=True, name="Decoder_LSTM")
    decoder_lstm_outputs = decoder_lstm(decoder_inputs,
                                        initial_state=encoder_states)
    decoder_dense = Dense(len(char_vocab), activation='softmax')
    decoder_outputs = TimeDistributed(decoder_dense)(decoder_lstm_outputs)

    # Define the model that will turn
    # `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
    model = Model([encoder_inputs, decoder_inputs], decoder_outputs)

    #show_keras_model(model)
```

In [20]:
```python
# Run training
"""
Don't be suprized that this model actually needs quite quite a lot of epochs to train,
so please be patient.
After the model is trained, you can use the history.history object to plot the metrics
improvment process.

While you are waiting for the model to train, feel free to read the next cell.
"""
batch_size = 1000
epochs = 75

# Here it's just some data transformation to translate the raw data to matrix inputs
encoder_input_data = to_categorical(char_to_num(train_X_raw, True),
num_classes=len(char_vocab))
train_Y = to_categorical(char_to_num(train_Y_raw, False), num_classes=len(char_vocab))
# for decoder, the target lags input by 1 time step
decoder_input_data = train_Y[:, :-1, :]
decoder_target_data = train_Y[:, 1:, :]

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_split=0.2)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/75
8000/8000 [==============================] - 10s 1ms/step - loss: 2.3666 - accuracy:
0.3223 - val_loss: 2.0043 - val_accuracy: 0.3602
Epoch 2/75
8000/8000 [==============================] - 9s 1ms/step - loss: 1.7434 - accuracy:
0.3616 - val_loss: 1.5760 - val_accuracy: 0.3549
Epoch 3/75
8000/8000 [==============================] - 8s 942us/step - loss: 1.5453 - accuracy:
0.3611 - val_loss: 1.5327 - val_accuracy: 0.3881
Epoch 4/75
8000/8000 [==============================] - 7s 822us/step - loss: 1.5267 - accuracy:
0.3696 - val_loss: 1.5220 - val_accuracy: 0.3652
Epoch 5/75
8000/8000 [==============================] - 6s 802us/step - loss: 1.5165 - accuracy:
0.3792 - val_loss: 1.5115 - val_accuracy: 0.3890
Epoch 6/75
8000/8000 [==============================] - 6s 797us/step - loss: 1.5040 - accuracy:
0.3963 - val_loss: 1.4940 - val_accuracy: 0.4240
Epoch 7/75
8000/8000 [==============================] - 6s 793us/step - loss: 1.4882 - accuracy:
0.4174 - val_loss: 1.4880 - val_accuracy: 0.4014
Epoch 8/75
8000/8000 [==============================] - 6s 790us/step - loss: 1.4781 - accuracy:
0.4184 - val_loss: 1.4658 - val_accuracy: 0.4305
Epoch 9/75
```

```
8000/8000 [==============================] - 7s 821us/step - loss: 1.4674 - accuracy:
0.4240 - val_loss: 1.4719 - val_accuracy: 0.4161
Epoch 10/75
8000/8000 [==============================] - 7s 821us/step - loss: 1.4596 - accuracy:
0.4292 - val_loss: 1.4544 - val_accuracy: 0.4365
Epoch 11/75
8000/8000 [==============================] - 7s 814us/step - loss: 1.4526 - accuracy:
0.4360 - val_loss: 1.4488 - val_accuracy: 0.4350
Epoch 12/75
8000/8000 [==============================] - 6s 802us/step - loss: 1.4445 - accuracy:
0.4387 - val_loss: 1.4429 - val_accuracy: 0.4388
Epoch 13/75
8000/8000 [==============================] - 7s 837us/step - loss: 1.4379 - accuracy:
0.4415 - val_loss: 1.4403 - val_accuracy: 0.4355
Epoch 14/75
8000/8000 [==============================] - 6s 798us/step - loss: 1.4353 - accuracy:
0.4408 - val_loss: 1.4377 - val_accuracy: 0.4333
Epoch 15/75
8000/8000 [==============================] - 7s 855us/step - loss: 1.4319 - accuracy:
0.4409 - val_loss: 1.4419 - val_accuracy: 0.4342
Epoch 16/75
8000/8000 [==============================] - 7s 831us/step - loss: 1.4285 - accuracy:
0.4417 - val_loss: 1.4229 - val_accuracy: 0.4439
Epoch 17/75
8000/8000 [==============================] - 7s 831us/step - loss: 1.4193 - accuracy:
0.4450 - val_loss: 1.4164 - val_accuracy: 0.4449
Epoch 18/75
8000/8000 [==============================] - 6s 811us/step - loss: 1.4108 - accuracy:
0.4484 - val_loss: 1.4094 - val_accuracy: 0.4450
Epoch 19/75
8000/8000 [==============================] - 7s 826us/step - loss: 1.4064 - accuracy:
0.4502 - val_loss: 1.4061 - val_accuracy: 0.4501
Epoch 20/75
8000/8000 [==============================] - 7s 819us/step - loss: 1.3972 - accuracy:
0.4543 - val_loss: 1.3980 - val_accuracy: 0.4484
Epoch 21/75
8000/8000 [==============================] - 6s 788us/step - loss: 1.3922 - accuracy:
0.4584 - val_loss: 1.3946 - val_accuracy: 0.4549
Epoch 22/75
8000/8000 [==============================] - 6s 779us/step - loss: 1.3871 - accuracy:
0.4602 - val_loss: 1.3810 - val_accuracy: 0.4631
Epoch 23/75
8000/8000 [==============================] - 6s 783us/step - loss: 1.3794 - accuracy:
0.4640 - val_loss: 1.3836 - val_accuracy: 0.4567
Epoch 24/75
8000/8000 [==============================] - 6s 802us/step - loss: 1.3760 - accuracy:
0.4614 - val_loss: 1.3675 - val_accuracy: 0.4656
Epoch 25/75
8000/8000 [==============================] - 6s 780us/step - loss: 1.3664 - accuracy:
0.4672 - val_loss: 1.3626 - val_accuracy: 0.4640
Epoch 26/75
8000/8000 [==============================] - 6s 781us/step - loss: 1.3617 - accuracy:
0.4683 - val_loss: 1.3547 - val_accuracy: 0.4702
Epoch 27/75
8000/8000 [==============================] - 6s 784us/step - loss: 1.3541 - accuracy:
0.4739 - val_loss: 1.3743 - val_accuracy: 0.4633
Epoch 28/75
8000/8000 [==============================] - 8s 979us/step - loss: 1.3572 - accuracy:
0.4685 - val_loss: 1.3680 - val_accuracy: 0.4645
```

```
Epoch 29/75
8000/8000 [==============================] - 7s 853us/step - loss: 1.3525 - accuracy:
0.4726 - val_loss: 1.3371 - val_accuracy: 0.4759
Epoch 30/75
8000/8000 [==============================] - 7s 823us/step - loss: 1.3288 - accuracy:
0.4830 - val_loss: 1.3257 - val_accuracy: 0.4776
Epoch 31/75
8000/8000 [==============================] - 7s 827us/step - loss: 1.3201 - accuracy:
0.4853 - val_loss: 1.3195 - val_accuracy: 0.4825
Epoch 32/75
8000/8000 [==============================] - 8s 994us/step - loss: 1.3352 - accuracy:
0.4776 - val_loss: 1.3371 - val_accuracy: 0.4720
Epoch 33/75
8000/8000 [==============================] - 7s 871us/step - loss: 1.3220 - accuracy:
0.4841 - val_loss: 1.3195 - val_accuracy: 0.4846
Epoch 34/75
8000/8000 [==============================] - 7s 874us/step - loss: 1.3108 - accuracy:
0.4884 - val_loss: 1.3188 - val_accuracy: 0.4795
Epoch 35/75
8000/8000 [==============================] - 6s 800us/step - loss: 1.2934 - accuracy:
0.4961 - val_loss: 1.2927 - val_accuracy: 0.4903
Epoch 36/75
8000/8000 [==============================] - 6s 808us/step - loss: 1.2820 - accuracy:
0.4988 - val_loss: 1.2854 - val_accuracy: 0.4954
Epoch 37/75
8000/8000 [==============================] - 6s 809us/step - loss: 1.2759 - accuracy:
0.5017 - val_loss: 1.4194 - val_accuracy: 0.4503
Epoch 38/75
8000/8000 [==============================] - 6s 809us/step - loss: 1.3217 - accuracy:
0.4832 - val_loss: 1.3108 - val_accuracy: 0.4802
Epoch 39/75
8000/8000 [==============================] - 7s 823us/step - loss: 1.2800 - accuracy:
0.4972 - val_loss: 1.3158 - val_accuracy: 0.4832
Epoch 40/75
8000/8000 [==============================] - 7s 840us/step - loss: 1.2866 - accuracy:
0.4943 - val_loss: 1.2922 - val_accuracy: 0.4901
Epoch 41/75
8000/8000 [==============================] - 8s 991us/step - loss: 1.2594 - accuracy:
0.5077 - val_loss: 1.2721 - val_accuracy: 0.5021
Epoch 42/75
8000/8000 [==============================] - 7s 875us/step - loss: 1.2576 - accuracy:
0.5092 - val_loss: 1.2560 - val_accuracy: 0.5026
Epoch 43/75
8000/8000 [==============================] - 7s 820us/step - loss: 1.2431 - accuracy:
0.5143 - val_loss: 1.2345 - val_accuracy: 0.5127
Epoch 44/75
8000/8000 [==============================] - 8s 942us/step - loss: 1.2297 - accuracy:
0.5210 - val_loss: 1.2322 - val_accuracy: 0.5138
Epoch 45/75
8000/8000 [==============================] - 7s 921us/step - loss: 1.2169 - accuracy:
0.5277 - val_loss: 1.2535 - val_accuracy: 0.4958
Epoch 46/75
8000/8000 [==============================] - 10s 1ms/step - loss: 1.2668 - accuracy:
0.4979 - val_loss: 1.2120 - val_accuracy: 0.5234
Epoch 47/75
8000/8000 [==============================] - 8s 1ms/step - loss: 1.2439 - accuracy:
0.5113 - val_loss: 1.2448 - val_accuracy: 0.4980
Epoch 48/75
8000/8000 [==============================] - 9s 1ms/step - loss: 1.2167 - accuracy:
```

```
0.5242 - val_loss: 1.2276 - val_accuracy: 0.5133
Epoch 49/75
8000/8000 [==============================] - 7s 893us/step - loss: 1.1965 - accuracy:
0.5396 - val_loss: 1.1867 - val_accuracy: 0.5403
Epoch 50/75
8000/8000 [==============================] - 7s 900us/step - loss: 1.1828 - accuracy:
0.5446 - val_loss: 1.1809 - val_accuracy: 0.5433
Epoch 51/75
8000/8000 [==============================] - 7s 821us/step - loss: 1.2063 - accuracy:
0.5254 - val_loss: 1.2070 - val_accuracy: 0.5111
Epoch 52/75
8000/8000 [==============================] - 7s 817us/step - loss: 1.1867 - accuracy:
0.5351 - val_loss: 1.1674 - val_accuracy: 0.5455
Epoch 53/75
8000/8000 [==============================] - 7s 813us/step - loss: 1.1622 - accuracy:
0.5545 - val_loss: 1.1903 - val_accuracy: 0.5211
Epoch 54/75
8000/8000 [==============================] - 7s 818us/step - loss: 1.1496 - accuracy:
0.5651 - val_loss: 1.1537 - val_accuracy: 0.5555
Epoch 55/75
8000/8000 [==============================] - 7s 837us/step - loss: 1.1932 - accuracy:
0.5242 - val_loss: 1.2351 - val_accuracy: 0.4908
Epoch 56/75
8000/8000 [==============================] - 7s 908us/step - loss: 1.1614 - accuracy:
0.5472 - val_loss: 1.1627 - val_accuracy: 0.5395
Epoch 57/75
8000/8000 [==============================] - 6s 801us/step - loss: 1.1339 - accuracy:
0.5730 - val_loss: 1.1595 - val_accuracy: 0.5325
Epoch 58/75
8000/8000 [==============================] - 7s 840us/step - loss: 1.1361 - accuracy:
0.5648 - val_loss: 1.1256 - val_accuracy: 0.5741
Epoch 59/75
8000/8000 [==============================] - 7s 828us/step - loss: 1.1284 - accuracy:
0.5706 - val_loss: 1.1924 - val_accuracy: 0.5144
Epoch 60/75
8000/8000 [==============================] - 7s 844us/step - loss: 1.1394 - accuracy:
0.5592 - val_loss: 1.1955 - val_accuracy: 0.5073
Epoch 61/75
8000/8000 [==============================] - 6s 808us/step - loss: 1.1530 - accuracy:
0.5448 - val_loss: 1.1087 - val_accuracy: 0.5864
Epoch 62/75
8000/8000 [==============================] - 6s 787us/step - loss: 1.1224 - accuracy:
0.5700 - val_loss: 1.1180 - val_accuracy: 0.5722
Epoch 63/75
8000/8000 [==============================] - 7s 816us/step - loss: 1.1045 - accuracy:
0.5842 - val_loss: 1.0978 - val_accuracy: 0.5930
Epoch 64/75
8000/8000 [==============================] - 7s 843us/step - loss: 1.1169 - accuracy:
0.5713 - val_loss: 1.1121 - val_accuracy: 0.5717
Epoch 65/75
8000/8000 [==============================] - 6s 796us/step - loss: 1.1123 - accuracy:
0.5759 - val_loss: 1.1161 - val_accuracy: 0.5662
Epoch 66/75
8000/8000 [==============================] - 7s 828us/step - loss: 1.1153 - accuracy:
0.5677 - val_loss: 1.1231 - val_accuracy: 0.5566
Epoch 67/75
8000/8000 [==============================] - 6s 809us/step - loss: 1.0950 - accuracy:
0.5869 - val_loss: 1.1237 - val_accuracy: 0.5475
Epoch 68/75
```

```
8000/8000 [==============================] - 6s 809us/step - loss: 1.0975 - accuracy:
0.5805 - val_loss: 1.1133 - val_accuracy: 0.5610
Epoch 69/75
8000/8000 [==============================] - 7s 839us/step - loss: 1.1020 - accuracy:
0.5769 - val_loss: 1.1176 - val_accuracy: 0.5536
Epoch 70/75
8000/8000 [==============================] - 6s 811us/step - loss: 1.0915 - accuracy:
0.5842 - val_loss: 1.0982 - val_accuracy: 0.5689
Epoch 71/75
8000/8000 [==============================] - 7s 821us/step - loss: 1.0828 - accuracy:
0.5897 - val_loss: 1.1022 - val_accuracy: 0.5712
Epoch 72/75
8000/8000 [==============================] - 7s 852us/step - loss: 1.0700 - accuracy:
0.6003 - val_loss: 1.0948 - val_accuracy: 0.5692
Epoch 73/75
8000/8000 [==============================] - 7s 853us/step - loss: 1.0883 - accuracy:
0.5804 - val_loss: 1.0653 - val_accuracy: 0.6031
Epoch 74/75
8000/8000 [==============================] - 7s 897us/step - loss: 1.0752 - accuracy:
0.5978 - val_loss: 1.1565 - val_accuracy: 0.5340
Epoch 75/75
8000/8000 [==============================] - 7s 828us/step - loss: 1.0911 - accuracy:
0.5791 - val_loss: 1.0670 - val_accuracy: 0.5951
```

In [21]:
```
# Trucate the encoder part of the training model as encoder model
encoder_model = Model(encoder_inputs, encoder_states)
#show_keras_model(encoder_model)
```

In [22]:
```
from keras.models import Model
# Build the inference model
inference_inputs = Input(batch_shape=(1,1, len(char_vocab)), name="Inference_Input")
inference_lstm = LSTM(latent_dim*2, stateful=True,
                      name="Inference_LSTM",)
inference_lstm_outputs = inference_lstm(inference_inputs)

inference_dense = Dense(len(char_vocab), activation='softmax')
inference_outputs = inference_dense(inference_lstm_outputs)

# Assign the weights of decoder to inference model
inference_lstm.set_weights(decoder_lstm.get_weights())
inference_dense.set_weights(decoder_dense.get_weights())

inference_model = Model(inference_inputs, inference_outputs)
#show_keras_model(inference_model)
```

In [23]:
```
def inference(encoder_input_data):
    """
    A utility function to generate the model prediction
    """
    states_h, states_c = encoder_model.predict(encoder_input_data)
    results = []
    inference_model.reset_states()
    for h, c in zip(states_h, states_c):
        sent, seed = [], reverse_vocab['^']
        inference_lstm.states[0].assign(h[None, :])
        inference_lstm.states[1].assign(c[None, :])
        for i in range(decoder_input_len):
            seed = to_categorical(np.array([seed]), num_classes=len(char_vocab))[None,
:, :]
            seed = inference_model.predict(seed)[0].argmax()
            sent.append(seed)

        results.append(sent)
```

```
        return num_to_char(results)
```

```
In [24]: # Let's look at some output
         print(num_to_char(encoder_input_data[:10].argmax(axis=2)))
         print(inference(encoder_input_data[:10]))
```

```
['513+83$$', '24+814$$', '978+758$', '346+625$', '466+166$', '743+379$', '16+586$$',
'161+899$', '81+799$$', '914+32$$']
['08664', '01642', '24242', '24242', '24242', '24242', '24242', '24242', '24242',
'24242']
```

6 Real Machine translation

```
In [25]: """
         Now are you ready for the real challenge? You can use the ita.txt file as training data.
         But feel free to download different language from http://www.manythings.org/anki/. If
         you
         happen to speak French or Japanese, it's time to show off!

         1. Implement a Bidrectional LSTM Encoder-Decoder model, or other viable models to
         translate
            the language dataset you choose.

         2. Write the function to calculate the BLEU score of your model
         """
         import os, sys
         # add utils folder to path
         p = os.path.dirname(os.getcwd())
         if p not in sys.path:
             sys.path = [p] + sys.path

         from utils.general import show_keras_model

         from keras.models import Model
```

```
In [26]: train_X_raw = []
         train_Y_raw = []
         f = open("ita.txt")
         for r in f:
             s = r.split('\t')
             train_X_raw.append(s[0])
             train_Y_raw.append(s[1])
         f.close()
         encoder_input_len = max([len(X) for X in train_X_raw])
         decoder_input_len = max([len(y) for y in train_Y_raw])
         print(encoder_input_len,decoder_input_len)
         latent_dim = 256
```

```
262 303
```

```
In [27]: from collections import Counter
         from keras.preprocessing.sequence import pad_sequences

         total_chars = ''.join(train_X_raw)+''.join(train_Y_raw)
         total_chars = Counter(total_chars)
         char_vocab = sorted([c for c in total_chars])+['^','<END>']
         reverse_vocab = {k:v for v, k in enumerate(char_vocab)}
         def char_to_num(X_raw, is_encoder=True):
             """
             Translate the raw input to the numerical encoding. We take different treatments for
         the
             encoder inputs and decoder inputs. This is because we need a starter character "^"
         for the
             decoder inputs.
```

```
        """
        result = [[reverse_vocab[c] for c in sent] for sent in X_raw]

        if is_encoder :
            assert all([len(row) <= encoder_input_len for row in X_raw])
            return pad_sequences(sequences=result, maxlen=encoder_input_len,
                                 padding='post', truncating='post',
                                 value=reverse_vocab['$'])
        else:
            assert all([len(row) <= decoder_input_len for row in X_raw])
            postpend = pad_sequences(sequences=result, maxlen=decoder_input_len,
                                 padding='post', truncating='post',
                                 value=reverse_vocab['<END>'])
            return pad_sequences(sequences=postpend, maxlen=decoder_input_len+1,
                                 padding='pre', truncating='post',
                                 value=reverse_vocab['^'])

        return pad_sequences(result)

    def num_to_char(X):
        return [''.join([char_vocab[c] for c in row]) for row in X]
```

In [28]:
```
print(char_vocab)
print(reverse_vocab)
```

```
[' ', '!', '"', '$', '%', "'", ',', '-', '.', '/', '0', '1', '2', '3', '4', '5', '6',
'7', '8', '9', ':', ';', '?', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
't', 'u', 'v', 'w', 'x', 'y', 'z', '\xa0', '\xad', '°', 'º', 'È', 'à', 'á', 'ã', 'è',
'é', 'ê', 'ì', 'î', 'ï', 'ñ', 'ò', 'ö', 'ù', 'ú', 'ü', 'ō', '\u200b', '’', '€', '^',
'<END>']
{' ': 0, '!': 1, '"': 2, '$': 3, '%': 4, "'": 5, ',': 6, '-': 7, '.': 8, '/': 9, '0':
10, '1': 11, '2': 12, '3': 13, '4': 14, '5': 15, '6': 16, '7': 17, '8': 18, '9': 19,
':': 20, ';': 21, '?': 22, 'A': 23, 'B': 24, 'C': 25, 'D': 26, 'E': 27, 'F': 28, 'G':
29, 'H': 30, 'I': 31, 'J': 32, 'K': 33, 'L': 34, 'M': 35, 'N': 36, 'O': 37, 'P': 38,
'Q': 39, 'R': 40, 'S': 41, 'T': 42, 'U': 43, 'V': 44, 'W': 45, 'X': 46, 'Y': 47, 'Z':
48, 'a': 49, 'b': 50, 'c': 51, 'd': 52, 'e': 53, 'f': 54, 'g': 55, 'h': 56, 'i': 57,
'j': 58, 'k': 59, 'l': 60, 'm': 61, 'n': 62, 'o': 63, 'p': 64, 'q': 65, 'r': 66, 's':
67, 't': 68, 'u': 69, 'v': 70, 'w': 71, 'x': 72, 'y': 73, 'z': 74, '\xa0': 75, '\xad':
76, '°': 77, 'º': 78, 'È': 79, 'à': 80, 'á': 81, 'ã': 82, 'è': 83, 'é': 84, 'ê': 85,
'ì': 86, 'î': 87, 'ï': 88, 'ñ': 89, 'ò': 90, 'ö': 91, 'ù': 92, 'ú': 93, 'ü': 94, 'ō':
95, '\u200b': 96, '’': 97, '€': 98, '^': 99, '<END>': 100}
```

In [29]:
```
from keras.layers import (Input, LSTM, Dense, Bidirectional, Embedding,
                          TimeDistributed, Concatenate)

encoder_inputs = Input(shape=(encoder_input_len, len(char_vocab)), name="Encoder_Input")
# For encoder, we can see the entire sentence at once, so we can use Bidirectional LSTM
encoder_lstm = Bidirectional(LSTM(latent_dim, return_state=True, name="Encoder_LSTM"))
# Bidrectional LSTM has 4 states instead of 2, we concatenate them to be comparable
# with the decoder LSTM
_, forward_h, forward_c, backward_h, backward_c = encoder_lstm(encoder_inputs)
state_h = Concatenate()([forward_h, backward_h])
state_c = Concatenate()([forward_c, backward_c])

# Set up the decoder, using `encoder_states` as initial state
encoder_states = [state_h, state_c]
decoder_inputs = Input(shape=(decoder_input_len, len(char_vocab)), name="Decoder_Input")
decoder_lstm = LSTM(latent_dim*2, return_sequences=True, name="Decoder_LSTM")
decoder_lstm_outputs = decoder_lstm(decoder_inputs,
                                    initial_state=encoder_states)
decoder_dense = Dense(len(char_vocab), activation='softmax')
decoder_outputs = TimeDistributed(decoder_dense)(decoder_lstm_outputs)
```

```
# Define the model that will turn
# `encoder_input_data` & `decoder_input_data` into `decoder_target_data`
model = Model([encoder_inputs, decoder_inputs], decoder_outputs)
```

In [32]:
```
# Run training
from keras.utils import to_categorical

"""
Don't be suprized that this model actually needs quite quite a lot of epochs to train,
so please be patient.
After the model is trained, you can use the history.history object to plot the metrics
improvment process.

While you are waiting for the model to train, feel free to read the next cell.
"""
batch_size = 1000
epochs = 1

# Here it's just some data transformation to translate the raw data to matrix inputs
encoder_input_data = to_categorical(char_to_num(train_X_raw[:10000], True),
num_classes=len(char_vocab))
train_Y = to_categorical(char_to_num(train_Y_raw[:10000], False),
num_classes=len(char_vocab))
# for decoder, the target lags input by 1 time step
decoder_input_data = train_Y[:, :-1, :]
decoder_target_data = train_Y[:, 1:, :]

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
history = model.fit([encoder_input_data, decoder_input_data], decoder_target_data,
                    batch_size=batch_size,
                    epochs=epochs,
                    validation_split=0.2)
```

```
Train on 8000 samples, validate on 2000 samples
Epoch 1/1
8000/8000 [==============================] - 513s 64ms/step - loss: 0.8874 - accuracy:
0.8644 - val_loss: 0.2618 - val_accuracy: 0.9467
```

In [ ]: