Chapter 30

Project: Develop a Neural Machine Translation Model

Machine translation is a challenging task that traditionally involves large statistical models developed using highly sophisticated linguistic knowledge. Neural machine translation is the use of deep neural networks for the problem of machine translation. In this tutorial, you will discover how to develop a neural machine translation system for translating German phrases to English. After completing this tutorial, you will know:

- How to clean and prepare data ready to train a neural machine translation system.
- How to develop an encoder-decoder model for machine translation.
- How to use a trained model for inference on new input phrases and evaluate the model skill.

Let's get started.

30.1 Tutorial Overview

This tutorial is divided into the following parts:

1. German to English Translation Dataset
2. Preparing the Text Data
3. Train Neural Translation Model
4. Evaluate Neural Translation Model

30.2 German to English Translation Dataset

In this tutorial, we will use a dataset of German to English terms used as the basis for flashcards for language learning. The dataset is available from the ManyThings.org website, with examples drawn from the Tatoeba Project. The dataset is comprised of German phrases and their English counterparts and is intended to be used with the Anki flashcard software.

- Download the English-German pairs dataset.
  http://www.manythings.org/anki/deu-eng.zip

Download the dataset to your current working directory and decompress it; for example:

```
unzip deu-eng.zip
```

Listing 30.1: Unzip the dataset

You will have a file called `deu.txt` that contains 152,820 pairs of English to German phases, one pair per line with a tab separating the language. For example, the first 5 lines of the file look as follows:

```
Hi.	Hallo!
Hi.	GruB Gott!
Run!	Lauf!
Wow!	Potzdonner!
Wow!	Donnerwetter!
```

Listing 30.2: Sample of the raw dataset (with Unicode characters normalized).

We will frame the prediction problem as given a sequence of words in German as input, translate or predict the sequence of words in English. The model we will develop will be suitable for some beginner German phrases.

30.3 Preparing the Text Data

The next step is to prepare the text data ready for modeling. Take a look at the raw data and note what you see that we might need to handle in a data cleaning operation. For example, here are some observations I note from reviewing the raw data:

- There is punctuation.
- The text contains uppercase and lowercase.
- There are special characters in the German.
- There are duplicate phrases in English with different translations in German.
- The file is ordered by sentence length with very long sentences toward the end of the file.

A good text cleaning procedure may handle some or all of these observations. Data preparation is divided into two subsections:

1. Clean Text
2. Split Text

30.3.1 Clean Text

First, we must load the data in a way that preserves the Unicode German characters. The function below called `load_doc()` will load the file as a blob of text.

```
# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, mode='rt', encoding='utf-8')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text
```

Listing 30.3: Function to load a file into memory

Each line contains a single pair of phrases, first English and then German, separated by a tab character. We must split the loaded text by line and then by phrase. The function `to_pairs()` below will split the loaded text.

```
# split a loaded document into sentences
def to_pairs(doc):
  lines = doc.strip().split('\n')
  pairs = [line.split('\t') for line in lines]
  return pairs
```

Listing 30.4: Function to split lines into pairs

We are now ready to clean each sentence. The specific cleaning operations we will perform are as follows:

- Remove all non-printable characters.
- Remove all punctuation characters.
- Normalize all Unicode characters to ASCII (e.g. Latin characters).
- Normalize the case to lowercase.
- Remove any remaining tokens that are not alphabetic.

We will perform these operations on each phrase for each pair in the loaded dataset. The `clean_pairs()` function below implements these operations.

```
# clean a list of lines
def clean_pairs(lines):
  cleaned = list()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  for pair in lines:
    clean_pair = list()
    for line in pair:
      # normalize unicode characters
      line = normalize('NFD', line).encode('ascii', 'ignore')
```

```
      line = line.decode('UTF-8')
      # tokenize on white space
      line = line.split()
      # convert to lowercase
      line = [word.lower() for word in line]
      # remove punctuation from each token
      line = [re_punc.sub('', w) for w in line]
      # remove non-printable chars form each token
      line = [re_print.sub('', w) for w in line]
      # remove tokens with numbers in them
      line = [word for word in line if word.isalpha()]
      # store as string
      clean_pair.append(' '.join(line))
    cleaned.append(clean_pair)
  return array(cleaned)
```

Listing 30.5: Function to clean text

Finally, now that the data has been cleaned, we can save the list of phrase pairs to a file ready for use. The function `save_clean_data()` uses the pickle API to save the list of clean text to file. Pulling all of this together, the complete example is listed below.

```
import string
import re
from pickle import dump
from unicodedata import normalize
from numpy import array

# load doc into memory
def load_doc(filename):
  # open the file as read only
  file = open(filename, mode='rt', encoding='utf-8')
  # read all text
  text = file.read()
  # close the file
  file.close()
  return text

# split a loaded document into sentences
def to_pairs(doc):
  lines = doc.strip().split('\n')
  pairs = [line.split('\t') for line in lines]
  return pairs

# clean a list of lines
def clean_pairs(lines):
  cleaned = list()
  # prepare regex for char filtering
  re_punc = re.compile('[%s]' % re.escape(string.punctuation))
  re_print = re.compile('[^%s]' % re.escape(string.printable))
  for pair in lines:
    clean_pair = list()
    for line in pair:
      # normalize unicode characters
      line = normalize('NFD', line).encode('ascii', 'ignore')
      line = line.decode('UTF-8')
```

```
      # tokenize on white space
      line = line.split()
      # convert to lowercase
      line = [word.lower() for word in line]
      # remove punctuation from each token
      line = [re_punc.sub('', w) for w in line]
      # remove non-printable chars form each token
      line = [re_print.sub('', w) for w in line]
      # remove tokens with numbers in them
      line = [word for word in line if word.isalpha()]
      # store as string
      clean_pair.append(' '.join(line))
    cleaned.append(clean_pair)
  return array(cleaned)

# save a list of clean sentences to file
def save_clean_data(sentences, filename):
  dump(sentences, open(filename, 'wb'))
  print('Saved: %s' % filename)

# load dataset
filename = 'deu.txt'
doc = load_doc(filename)
# split into english-german pairs
pairs = to_pairs(doc)
# clean sentences
clean_pairs = clean_pairs(pairs)
# save clean pairs to file
save_clean_data(clean_pairs, 'english-german.pkl')
# spot check
for i in range(100):
  print('[%s] => [%s]' % (clean_pairs[i,0], clean_pairs[i,1]))
```

Listing 30.6: Complete example of text data preparation.

Running the example creates a new file in the current working directory with the cleaned text called `english-german.pkl`. Some examples of the clean text are printed for us to evaluate at the end of the run to confirm that the clean operations were performed as expected.

30.3.2 Split Text

The clean data contains a little over 150,000 phrase pairs and some of the pairs toward the end of the file are very long. This is a good number of examples for developing a small translation model. The complexity of the model increases with the number of examples, length of phrases, and size of the vocabulary. Although we have a good dataset for modeling translation, we will simplify the problem slightly to dramatically reduce the size of the model required, and in turn the training time required to fit the model.

You can explore developing a model on the fuller dataset as an extension; I would love to hear how you do. We will simplify the problem by reducing the dataset to the first 10,000 examples in the file; these will be the shortest phrases in the dataset. Further, we will then stake the first 9,000 of those as examples for training and the remaining 1,000 examples to test the fit model.

Below is the complete example of loading the clean data, splitting it, and saving the split portions of data to new files.

```
from pickle import load
from pickle import dump
from numpy.random import shuffle

# load a clean dataset
def load_clean_sentences(filename):
  return load(open(filename, 'rb'))

# save a list of clean sentences to file
def save_clean_data(sentences, filename):
  dump(sentences, open(filename, 'wb'))
  print('Saved: %s' % filename)

# load dataset
raw_dataset = load_clean_sentences('english-german.pkl')

# reduce dataset size
n_sentences = 10000
dataset = raw_dataset[:n_sentences, :]
# random shuffle
shuffle(dataset)
# split into train/test
train, test = dataset[:9000], dataset[9000:]
# save
save_clean_data(dataset, 'english-german-both.pkl')
save_clean_data(train, 'english-german-train.pkl')
save_clean_data(test, 'english-german-test.pkl')
```

Listing 30.7: Complete example of splitting text data.

Running the example creates three new files: the `english-german-both.pkl` that contains all of the train and test examples that we can use to define the parameters of the problem, such as max phrase lengths and the vocabulary, and the `english-german-train.pkl` and `english-german-test.pkl` files for the train and test dataset. We are now ready to start developing our translation model.

30.4 Train Neural Translation Model

In this section, we will develop the translation model. This involves both loading and preparing the clean text data ready for modeling and defining and training the model on the prepared data. Let's start off by loading the datasets so that we can prepare the data. The function below named `load_clean_sentences()` can be used to load the train, test, and both datasets in turn.

```
# load a clean dataset
def load_clean_sentences(filename):
  return load(open(filename, 'rb'))

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
```

```
test = load_clean_sentences('english-german-test.pkl')
```

Listing 30.8: Load cleaned data from file.

We will use the *both* or combination of the train and test datasets to define the maximum length and vocabulary of the problem. This is for simplicity. Alternately, we could define these properties from the training dataset alone and truncate examples in the test set that are too long or have words that are out of the vocabulary. We can use the Keras Tokenize class to map words to integers, as needed for modeling. We will use separate tokenizer for the English sequences and the German sequences. The function below-named `create_tokenizer()` will train a tokenizer on a list of phrases.

```
# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer
```

Listing 30.9: Fit a tokenizer on the clean text data.

Similarly, the function named `max_length()` below will find the length of the longest sequence in a list of phrases.

```
# max sentence length
def max_length(lines):
  return max(len(line.split()) for line in lines)
```

Listing 30.10: Calculate the maximum sequence length.

We can call these functions with the combined dataset to prepare tokenizers, vocabulary sizes, and maximum lengths for both the English and German phrases.

```
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
print('English Vocabulary Size: %d' % eng_vocab_size)
print('English Max Length: %d' % (eng_length))
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
print('German Vocabulary Size: %d' % ger_vocab_size)
print('German Max Length: %d' % (ger_length))
```

Listing 30.11: Prepare Tokenizers for source and target sequences.

We are now ready to prepare the training dataset. Each input and output sequence must be encoded to integers and padded to the maximum phrase length. This is because we will use a word embedding for the input sequences and one hot encode the output sequences The function below named `encode_sequences()` will perform these operations and return the result.

```
# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
  # integer encode sequences
  X = tokenizer.texts_to_sequences(lines)
  # pad sequences with 0 values
```

```
  X = pad_sequences(X, maxlen=length, padding='post')
  return X
```

Listing 30.12: Function to encode and pad sequences.

The output sequence needs to be one hot encoded. This is because the model will predict the probability of each word in the vocabulary as output. The function `encode_output()` below will one hot encode English output sequences.

```
# one hot encode target sequence
def encode_output(sequences, vocab_size):
  ylist = list()
  for sequence in sequences:
    encoded = to_categorical(sequence, num_classes=vocab_size)
    ylist.append(encoded)
  y = array(ylist)
  y = y.reshape(sequences.shape[0], sequences.shape[1], vocab_size)
  return y
```

Listing 30.13: One hot encode output sequences.

We can make use of these two functions and prepare both the train and test dataset ready for training the model.

```
# prepare training data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
trainY = encode_sequences(eng_tokenizer, eng_length, train[:, 0])
trainY = encode_output(trainY, eng_vocab_size)
# prepare validation data
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
testY = encode_sequences(eng_tokenizer, eng_length, test[:, 0])
testY = encode_output(testY, eng_vocab_size)
```

Listing 30.14: Prepare training and test data for modeling.

We are now ready to define the model. We will use an encoder-decoder LSTM model on this problem. In this architecture, the input sequence is encoded by a front-end model called the encoder then decoded word by word by a backend model called the decoder. The function `define_model()` below defines the model and takes a number of arguments used to configure the model, such as the size of the input and output vocabularies, the maximum length of input and output phrases, and the number of memory units used to configure the model.

The model is trained using the efficient Adam approach to stochastic gradient descent and minimizes the categorical loss function because we have framed the prediction problem as multiclass classification. The model configuration was not optimized for this problem, meaning that there is plenty of opportunity for you to tune it and lift the skill of the translations. I would love to see what you can come up with.

```
# define NMT model
def define_model(src_vocab, tar_vocab, src_timesteps, tar_timesteps, n_units):
  model = Sequential()
  model.add(Embedding(src_vocab, n_units, input_length=src_timesteps, mask_zero=True))
  model.add(LSTM(n_units))
  model.add(RepeatVector(tar_timesteps))
  model.add(LSTM(n_units, return_sequences=True))
  model.add(TimeDistributed(Dense(tar_vocab, activation='softmax')))
  # compile model
```

```
  model.compile(optimizer='adam', loss='categorical_crossentropy')
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model
```

Listing 30.15: Define and summarize the model.

Finally, we can train the model. We train the model for 30 epochs and a batch size of 64 examples. We use checkpointing to ensure that each time the model skill on the test set improves, the model is saved to file.

```
# fit model
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
    save_best_only=True, mode='min')
model.fit(trainX, trainY, epochs=30, batch_size=64, validation_data=(testX, testY),
    callbacks=[checkpoint], verbose=2)
```

Listing 30.16: Fit the defined model and save models using checkpointing.

We can tie all of this together and fit the neural translation model. The complete working example is listed below.

```
from pickle import load
from numpy import array
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Embedding
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.callbacks import ModelCheckpoint

# load a clean dataset
def load_clean_sentences(filename):
  return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# max sentence length
def max_length(lines):
  return max(len(line.split()) for line in lines)

# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
  # integer encode sequences
  X = tokenizer.texts_to_sequences(lines)
  # pad sequences with 0 values
  X = pad_sequences(X, maxlen=length, padding='post')
```

```
  return X

# one hot encode target sequence
def encode_output(sequences, vocab_size):
  ylist = list()
  for sequence in sequences:
    encoded = to_categorical(sequence, num_classes=vocab_size)
    ylist.append(encoded)
  y = array(ylist)
  y = y.reshape(sequences.shape[0], sequences.shape[1], vocab_size)
  return y

# define NMT model
def define_model(src_vocab, tar_vocab, src_timesteps, tar_timesteps, n_units):
  model = Sequential()
  model.add(Embedding(src_vocab, n_units, input_length=src_timesteps, mask_zero=True))
  model.add(LSTM(n_units))
  model.add(RepeatVector(tar_timesteps))
  model.add(LSTM(n_units, return_sequences=True))
  model.add(TimeDistributed(Dense(tar_vocab, activation='softmax')))
  # compile model
  model.compile(optimizer='adam', loss='categorical_crossentropy')
  # summarize defined model
  model.summary()
  plot_model(model, to_file='model.png', show_shapes=True)
  return model

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
print('English Vocabulary Size: %d' % eng_vocab_size)
print('English Max Length: %d' % (eng_length))
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
print('German Vocabulary Size: %d' % ger_vocab_size)
print('German Max Length: %d' % (ger_length))
# prepare training data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
trainY = encode_sequences(eng_tokenizer, eng_length, train[:, 0])
trainY = encode_output(trainY, eng_vocab_size)
# prepare validation data
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
testY = encode_sequences(eng_tokenizer, eng_length, test[:, 0])
testY = encode_output(testY, eng_vocab_size)
# define model
model = define_model(ger_vocab_size, eng_vocab_size, ger_length, eng_length, 256)
# fit model
checkpoint = ModelCheckpoint('model.h5', monitor='val_loss', verbose=1,
    save_best_only=True, mode='min')
```

```
model.fit(trainX, trainY, epochs=30, batch_size=64, validation_data=(testX, testY),
    callbacks=[checkpoint], verbose=2)
```

Listing 30.17: Complete example of training the neural machine translation model.

Running the example first prints a summary of the parameters of the dataset such as vocabulary size and maximum phrase lengths.

```
English Vocabulary Size: 2404
English Max Length: 5
German Vocabulary Size: 3856
German Max Length: 10
```

Listing 30.18: Summary of the loaded data

Next, a summary of the defined model is printed, allowing us to confirm the model configuration.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 10, 256)           987136
_________________________________________________________________
lstm_1 (LSTM)                (None, 256)               525312
_________________________________________________________________
repeat_vector_1 (RepeatVecto (None, 5, 256)            0
_________________________________________________________________
lstm_2 (LSTM)                (None, 5, 256)            525312
_________________________________________________________________
time_distributed_1 (TimeDist (None, 5, 2404)           617828
=================================================================
Total params: 2,655,588
Trainable params: 2,655,588
Non-trainable params: 0
_________________________________________________________________
```

Listing 30.19: Summary of the defined model

A plot of the model is also created providing another perspective on the model configuration.

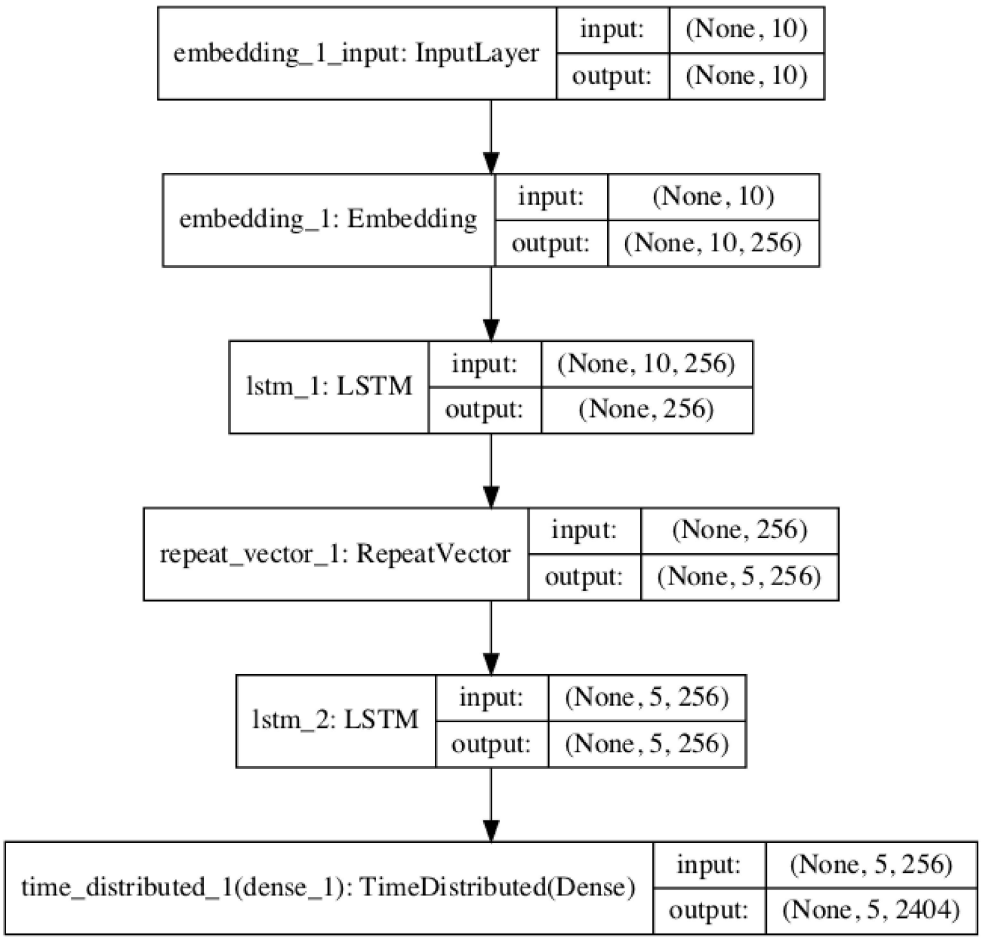

Figure 30.1: Plot of the defined neural machine translation model

Next, the model is trained. Each epoch takes about 30 seconds on modern CPU hardware; no GPU is required. During the run, the model will be saved to the file `model.h5`, ready for inference in the next step.

```
...
Epoch 26/30
Epoch 00025: val_loss improved from 2.20048 to 2.19976, saving model to model.h5
17s - loss: 0.7114 - val_loss: 2.1998
Epoch 27/30
Epoch 00026: val_loss improved from 2.19976 to 2.18255, saving model to model.h5
17s - loss: 0.6532 - val_loss: 2.1826
Epoch 28/30
Epoch 00027: val_loss did not improve
17s - loss: 0.5970 - val_loss: 2.1970
Epoch 29/30
Epoch 00028: val_loss improved from 2.18255 to 2.17872, saving model to model.h5
17s - loss: 0.5474 - val_loss: 2.1787
Epoch 30/30
Epoch 00029: val_loss did not improve
17s - loss: 0.5023 - val_loss: 2.1823
```

Listing 30.20: Summary output from training the neural machine translation model.

30.5 Evaluate Neural Translation Model

We will evaluate the model on the train and the test dataset. The model should perform very well on the train dataset and ideally have been generalized to perform well on the test dataset. Ideally, we would use a separate validation dataset to help with model selection during training instead of the test set. You can try this as an extension. The clean datasets must be loaded and prepared as before.

```
...
# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
# prepare data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
```

Listing 30.21: Load and prepare data.

Next, the best model saved during training must be loaded.

```
# load model
model = load_model('model.h5')
```

Listing 30.22: Load and the saved model.

Evaluation involves two steps: first generating a translated output sequence, and then repeating this process for many input examples and summarizing the skill of the model across multiple cases. Starting with inference, the model can predict the entire output sequence in a one-shot manner.

```
translation = model.predict(source, verbose=0)
```

Listing 30.23: Predict the target sequence given the source sequence.

This will be a sequence of integers that we can enumerate and lookup in the tokenizer to map back to words. The function below, named `word_for_id()`, will perform this reverse mapping.

```
# map an integer to a word
def word_for_id(integer, tokenizer):
  for word, index in tokenizer.word_index.items():
    if index == integer:
      return word
```

```
  return None
```

Listing 30.24: Map a predicted word index to the word in the vocabulary.

We can perform this mapping for each integer in the translation and return the result as a string of words. The function `predict_sequence()` below performs this operation for a single encoded source phrase.

```
# generate target given source sequence
def predict_sequence(model, tokenizer, source):
  prediction = model.predict(source, verbose=0)[0]
  integers = [argmax(vector) for vector in prediction]
  target = list()
  for i in integers:
    word = word_for_id(i, tokenizer)
    if word is None:
      break
    target.append(word)
  return ' '.join(target)
```

Listing 30.25: Predict and interpret the target sequence.

Next, we can repeat this for each source phrase in a dataset and compare the predicted result to the expected target phrase in English. We can print some of these comparisons to screen to get an idea of how the model performs in practice. We will also calculate the BLEU scores to get a quantitative idea of how well the model has performed. The `evaluate_model()` function below implements this, calling the above `predict_sequence()` function for each phrase in a provided dataset.

```
# evaluate the skill of the model
def evaluate_model(model, tokenizer, sources, raw_dataset):
  actual, predicted = list(), list()
  for i, source in enumerate(sources):
    # translate encoded source text
    source = source.reshape((1, source.shape[0]))
    translation = predict_sequence(model, eng_tokenizer, source)
    raw_target, raw_src = raw_dataset[i]
    if i < 10:
      print('src=[%s], target=[%s], predicted=[%s]' % (raw_src, raw_target, translation))
    actual.append(raw_target.split())
    predicted.append(translation.split())
  # calculate BLEU score
  print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
  print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
  print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
  print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))
```

Listing 30.26: Function to evaluate a fit model.

We can tie all of this together and evaluate the loaded model on both the training and test datasets. The complete code listing is provided below.

```
from pickle import load
from numpy import argmax
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
from keras.models import load_model
```

```
from nltk.translate.bleu_score import corpus_bleu

# load a clean dataset
def load_clean_sentences(filename):
  return load(open(filename, 'rb'))

# fit a tokenizer
def create_tokenizer(lines):
  tokenizer = Tokenizer()
  tokenizer.fit_on_texts(lines)
  return tokenizer

# max sentence length
def max_length(lines):
  return max(len(line.split()) for line in lines)

# encode and pad sequences
def encode_sequences(tokenizer, length, lines):
  # integer encode sequences
  X = tokenizer.texts_to_sequences(lines)
  # pad sequences with 0 values
  X = pad_sequences(X, maxlen=length, padding='post')
  return X

# map an integer to a word
def word_for_id(integer, tokenizer):
  for word, index in tokenizer.word_index.items():
    if index == integer:
      return word
  return None

# generate target given source sequence
def predict_sequence(model, tokenizer, source):
  prediction = model.predict(source, verbose=0)[0]
  integers = [argmax(vector) for vector in prediction]
  target = list()
  for i in integers:
    word = word_for_id(i, tokenizer)
    if word is None:
      break
    target.append(word)
  return ' '.join(target)

# evaluate the skill of the model
def evaluate_model(model, sources, raw_dataset):
  actual, predicted = list(), list()
  for i, source in enumerate(sources):
    # translate encoded source text
    source = source.reshape((1, source.shape[0]))
    translation = predict_sequence(model, eng_tokenizer, source)
    raw_target, raw_src = raw_dataset[i]
    if i < 10:
      print('src=[%s], target=[%s], predicted=[%s]' % (raw_src, raw_target, translation))
    actual.append(raw_target.split())
    predicted.append(translation.split())
  # calculate BLEU score
```

```
  print('BLEU-1: %f' % corpus_bleu(actual, predicted, weights=(1.0, 0, 0, 0)))
  print('BLEU-2: %f' % corpus_bleu(actual, predicted, weights=(0.5, 0.5, 0, 0)))
  print('BLEU-3: %f' % corpus_bleu(actual, predicted, weights=(0.3, 0.3, 0.3, 0)))
  print('BLEU-4: %f' % corpus_bleu(actual, predicted, weights=(0.25, 0.25, 0.25, 0.25)))

# load datasets
dataset = load_clean_sentences('english-german-both.pkl')
train = load_clean_sentences('english-german-train.pkl')
test = load_clean_sentences('english-german-test.pkl')
# prepare english tokenizer
eng_tokenizer = create_tokenizer(dataset[:, 0])
eng_vocab_size = len(eng_tokenizer.word_index) + 1
eng_length = max_length(dataset[:, 0])
# prepare german tokenizer
ger_tokenizer = create_tokenizer(dataset[:, 1])
ger_vocab_size = len(ger_tokenizer.word_index) + 1
ger_length = max_length(dataset[:, 1])
# prepare data
trainX = encode_sequences(ger_tokenizer, ger_length, train[:, 1])
testX = encode_sequences(ger_tokenizer, ger_length, test[:, 1])
# load model
model = load_model('model.h5')
# test on some training sequences
print('train')
evaluate_model(model, trainX, train)
# test on some test sequences
print('test')
evaluate_model(model, testX, test)
```

Listing 30.27: Complete example of translating text with a fit neural machine translation model.

Running the example first prints examples of source text, expected and predicted translations, as well as scores for the training dataset, followed by the test dataset. Your specific results will differ given the random shuffling of the dataset and the stochastic nature of neural networks. Looking at the results for the test dataset first, we can see that the translations are readable and mostly correct. For example: '*ich liebe dich*' was correctly translated to '*i love you*'.

We can also see that the translations were not perfect, with '*ich konnte nicht gehen*' translated to *i cant go* instead of the expected '*i couldnt walk*'. We can also see the BLEU-4 score of 0.51, which provides an upper bound on what we might expect from this model.

Note: Given the stochastic nature of neural networks, your specific results may vary. Consider running the example a few times.

```
src=[ich liebe dich], target=[i love you], predicted=[i love you]
src=[ich sagte du sollst den mund halten], target=[i said shut up], predicted=[i said stop
    up]
src=[wie geht es eurem vater], target=[hows your dad], predicted=[hows your dad]
src=[das gefallt mir], target=[i like that], predicted=[i like that]
src=[ich gehe immer zu fu], target=[i always walk], predicted=[i will to]
src=[ich konnte nicht gehen], target=[i couldnt walk], predicted=[i cant go]
src=[er ist sehr jung], target=[he is very young], predicted=[he is very young]
src=[versucht es doch einfach], target=[just try it], predicted=[just try it]
src=[sie sind jung], target=[youre young], predicted=[youre young]
src=[er ging surfen], target=[he went surfing], predicted=[he went surfing]
```

```
BLEU-1: 0.085682
BLEU-2: 0.284191
BLEU-3: 0.459090
BLEU-4: 0.517571
```

Listing 30.28: Sample output translation on the training dataset.

Looking at the results on the test set, do see readable translations, which is not an easy task. For example, we see '*ich mag dich nicht*' correctly translated to '*i dont like you*'. We also see some poor translations and a good case that the model could support from further tuning, such as '*ich bin etwas beschwipst*' translated as '*i a bit bit*' instead of the expected *im a bit tipsy* A BLEU-4 score of 0.076238 was achieved, providing a baseline skill to improve upon with further improvements to the model.

```
src=[tom erblasste], target=[tom turned pale], predicted=[tom went pale]
src=[bring mich nach hause], target=[take me home], predicted=[let us at]
src=[ich bin etwas beschwipst], target=[im a bit tipsy], predicted=[i a bit bit]
src=[das ist eine frucht], target=[its a fruit], predicted=[thats a a]
src=[ich bin pazifist], target=[im a pacifist], predicted=[im am]
src=[unser plan ist aufgegangen], target=[our plan worked], predicted=[who is a man]
src=[hallo tom], target=[hi tom], predicted=[hello tom]
src=[sei nicht nervos], target=[dont be nervous], predicted=[dont be crazy]
src=[ich mag dich nicht], target=[i dont like you], predicted=[i dont like you]
src=[tom stellte eine falle], target=[tom set a trap], predicted=[tom has a cough]

BLEU-1: 0.082088
BLEU-2: 0.006182
BLEU-3: 0.046129
BLEU-4: 0.076238
```

Listing 30.29: Sample output translation on the test dataset.

30.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Cleaning**. Different data cleaning operations could be performed on the data, such as not removing punctuation or normalizing case, or perhaps removing duplicate English phrases.
- **Vocabulary**. The vocabulary could be refined, perhaps removing words used less than 5 or 10 times in the dataset and replaced with *unk*.
- **More Data**. The dataset used to fit the model could be expanded to 50,000, 100,000 phrases, or more.
- **Input Order**. The order of input phrases could be reversed, which has been reported to lift skill, or a Bidirectional input layer could be used.
- **Layers**. The encoder and/or the decoder models could be expanded with additional layers and trained for more epochs, providing more representational capacity for the model.

- **Units**. The number of memory units in the encoder and decoder could be increased, providing more representational capacity for the model.
- **Regularization**. The model could use regularization, such as weight or activation regularization, or the use of dropout on the LSTM layers.
- **Pre-Trained Word Vectors**. Pre-trained word vectors could be used in the model.
- **Alternate Measure**. Explore alternate performance measures beside BLEU such as ROGUE. Compare scores for the same translations to develop an intuition for how the measures differ in practice.
- **Recursive Model**. A recursive formulation of the model could be used where the next word in the output sequence could be conditional on the input sequence and the output sequence generated so far.

If you explore any of these extensions, I'd love to know.

30.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

30.7.1 Dataset

- Tab-delimited Bilingual Sentence Pairs.
  http://www.manythings.org/anki/
- `German - English deu-eng.zip`.
  http://www.manythings.org/anki/deu-eng.zip

30.7.2 Neural Machine Translation

- *Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation*, 2016.
  https://arxiv.org/abs/1609.08144
- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215
- *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, 2014.
  https://arxiv.org/abs/1406.1078
- *Neural Machine Translation by Jointly Learning to Align and Translate*, 2014.
  https://arxiv.org/abs/1409.0473
- *On the Properties of Neural Machine Translation: Encoder-Decoder Approaches*, 2014.
  https://arxiv.org/abs/1409.1259
- *Massive Exploration of Neural Machine Translation Architectures*, 2017.
  https://arxiv.org/abs/1703.03906

30.8 Summary

In this tutorial, you discovered how to develop a neural machine translation system for translating German phrases to English. Specifically, you learned:

- How to clean and prepare data ready to train a neural machine translation system.
- How to develop an encoder-decoder model for machine translation.
- How to use a trained model for inference on new input phrases and evaluate the model skill.

30.8.1 Next

This is the final chapter for the machine translation part. In the next part you will discover helpful information in the appendix.