UDACITY

Return to Classroom

Machine Translation

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Dear Student,

I am really impressed with the amount of effort you've put into the project. You deserve applaud for your hardwork!

🎉 Finally, Congratulations on completing this project. You are one step closer to finishing your Nanodegree.

Wishing you good luck for all future projects 🎉

Some general suggestions

Use of assertions and Logging:

- Consider using Python assertions for sanity testing - assertions are great for catching bugs. This is especially true of a dynamically type-checked language like Python where a wrong variable type or shape can cause errors at runtime
- Logging is important for long-running applications. Logging done right produces a report that can be analyzed to debug errors and find crucial information. There could be different levels of logging or logging tags that can be used to filter messages most relevant to someone. Messages can be written to the terminal using print() or saved to file, for example using the Logger module. Sometimes it's worthwhile to catch and log exceptions during a long-running operation so that the operation itself is not aborted.

Debugging:

- Check out this guide on debugging in python

Reproducibility:

- Reproducibility is perhaps the biggest issue in machine learning right now. With so many moving parts present in the code (data, hyperparameters, etc) it is imperative that the instructions and code make it easy for anyone to get exactly the same results (just imagine debugging an ML pipeline where the data changes every time and so you cannot get the same result twice).
- Also consider using random seeds to make your data more reproducible.

Optimization and Profiling:

- Monitoring progress and debugging with Tensorboard: This tool can log detailed information about the model, data, hyperparameters, and more. Tensorboard can be used with Pytorch as well.

Submitted Files

The following files have been submitted: `helper.py`, `machine_translation.ipynb`, `machine_translation.html`

All required files are included in the submission zip.

✅ `Jupyter Notebook`
✅ `helper.py`

Suggestion:
You can export your conda environment into `environment.yaml` file so that you can recreate your conda environment later while practicing on your own system. Use the following command -

```
conda env export -f environment.yaml
```

Preprocess

The function `tokenize` returns tokenized input and the tokenized class.

```
def tokenize(x):
    """
    Tokenize x
    :param x: List of sentences/strings to be tokenized
    :return: Tuple of (tokenized x data, tokenizer used to tokenize x)
    """
    # TODO: Implement
    tokenizer = Tokenizer()
```

```
    tokenizer.fit_on_texts(x)
    tokenizer_txt2seq_x = tokenizer.texts_to_sequences(x)

    return tokenizer_txt2seq_x, tokenizer
```

`Tokenize` function takes in the input sequence, tokenizes it and then returns the tokenized sequence and the tokenizer class.

Why do we need to preprocess the input data before passing it into a Neural network?

- Text data is represented on computers using an encoding scheme such as ASCII or UNICODE, that maps every character to a number. Computers store and transmit these values as binary. So a string such as "UDACITY" is internally stored just as an array of binary values. The neural network won't be able to extract any meaningful information either from the binary values or from the encoding scheme values.
- This is why pre-processing is extremely important. During the pre-processing phase we might remove source specific markers (such as HTML tags from website data), punctuations, stopwords, etc.
- While some preprocessing steps are language agnostic, others are heavily dependent on the language we are working with.
  e.g. Languages like French have punctuations as part of the words. As such we need to carefully evaluate our data before we perform pre-processing.

The function `pad` returns padded input to the correct length.

The `pad` function takes in a tokenized input sequence and pads it or truncates to the required length based on the `length` argument.

```
def pad(x, length=None):
    """
    Pad x
    :param x: List of sequences.
    :param length: Length to pad the sequence to.  If None, use length of longest
sequence in x.
    :return: Padded numpy array of sequences
    """
    # TODO: Implement
    if length is None:
        list_len_sentence = [len(sentence) for sentence in x]
        length = max(list_len_sentence)

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

Padding is important to ensure all input sequences are of the same size when being passed to the model.

Models

The function `simple_model` builds a basic RNN model.

✅ Nicely done! You've designed a basic RNN pipeline and your model achieves reasonable validation accuracy.

```
                                  _                  _
Epoch 7/10
110288/110288 [==============================] - 10s 94us/step - loss: 1.
1135 - acc: 0.6651 - val_loss: nan - val_acc: 0.6714
Epoch 8/10
110288/110288 [==============================] - 10s 93us/step - loss: 1.
0722 - acc: 0.6716 - val_loss: nan - val_acc: 0.6733
Epoch 9/10
110288/110288 [==============================] - 10s 94us/step - loss: 1.
0356 - acc: 0.6766 - val_loss: nan - val_acc: 0.6766
Epoch 10/10
110288/110288 [==============================] - 10s 93us/step - loss: 1.
0066 - acc: 0.6821 - val_loss: nan - val_acc: 0.6756
new jersey est parfois calme en mois de mai et il est en en <PAD> <PAD>
<PAD> <PAD> <PAD> <PAD> <PAD>
```

⚠️ The validation loss throughout the training loop is `nan`. This can be fixed by adding 1 to vocab sizes in the function call

```
simple_rnn_model = simple_model(
    tmp_x.shape,
    max_french_sequence_length,
    english_vocab_size+1,
    french_vocab_size+1)
```

⚠️ Suggestion - While you've done a great job building a model, the network's performance isn't remarkable. You could try tuning the hyperparameters, especially the layer sizes. That should help improve the model's accuracy.

The function `embed_model` builds a RNN model using word embedding.

```
: def embed_model(input_shape, output_sequence_length, english_vocab_size, fr
  ench_vocab_size):
      """
      Build and train a RNN model using word embedding on x and y
      :param input_shape: Tuple of input shape
      :param output_sequence_length: Length of output sequence
      :param english_vocab_size: Number of unique English words in the datase
  t
      :param french_vocab_size: Number of unique French words in the dataset
      :return: Keras model built, but not trained
      """
      # TODO: Implement
      input_sequence = Input(input_shape[1:])
      embedding = Embedding(input_dim=english_vocab_size, output_dim=64)(inpu
  t_sequence)
      rnn = GRU(256, return_sequences=True)(embedding)
      dense = Dense(french_vocab_size, activation='softmax')
      logits = TimeDistributed(dense)(rnn)
      model = Model(input_sequence, Activation('tanh')(logits))

      model.compile(loss=sparse_categorical_crossentropy,
                    optimizer=Adam(lr=0.001),
                    metrics=['accuracy'])

      return model
```

Keras' Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset.

It is a flexible layer that can be used in a following ways:

- As part of a deep learning model where the embedding is learned along with the model itself.
- It can be used alone to learn a word embedding that can be saved and used in another model later.
- To load a pre-trained word embedding model, a type of transfer learning.

Additional Reading:
1) Check out this detailed guide on Word2Vec
2) A visual guide to Word2Vec

The Embedding RNN is trained on the dataset. A prediction using the model on the training dataset is printed in the notebook.

```
: def embed_model(input_shape, output_sequence_length, english_vocab_size, fr
  ench_vocab_size):
      """
      Build and train a RNN model using word embedding on x and y
      :param input_shape: Tuple of input shape
      :param output_sequence_length: Length of output sequence
      :param english_vocab_size: Number of unique English words in the datase
  t
      :param french_vocab_size: Number of unique French words in the dataset
      :return: Keras model built, but not trained
      """
      # TODO: Implement
      input_sequence = Input(input_shape[1:])
      embedding = Embedding(input_dim=english_vocab_size, output_dim=64)(inpu
  t_sequence)
      rnn = GRU(256, return_sequences=True)(embedding)
      dense = Dense(french_vocab_size, activation='softmax')
      logits = TimeDistributed(dense)(rnn)
      model = Model(input_sequence, Activation('tanh')(logits))

      model.compile(loss=sparse_categorical_crossentropy,
                    optimizer=Adam(lr=0.001),
                    metrics=['accuracy'])

      return model
```

✅ The model successfully trains on the dataset and achieves reliable validation accuracy.

✅ Notebook also contains predictions made by the model.

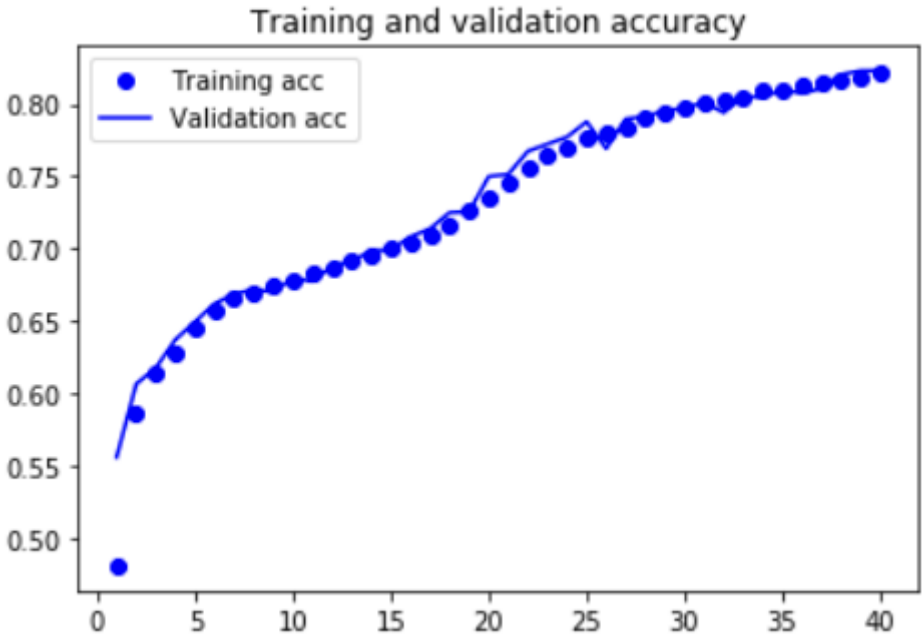
💡 Suggestion: You can also print the original english sentence, the french ground truth and french prediction as shown below

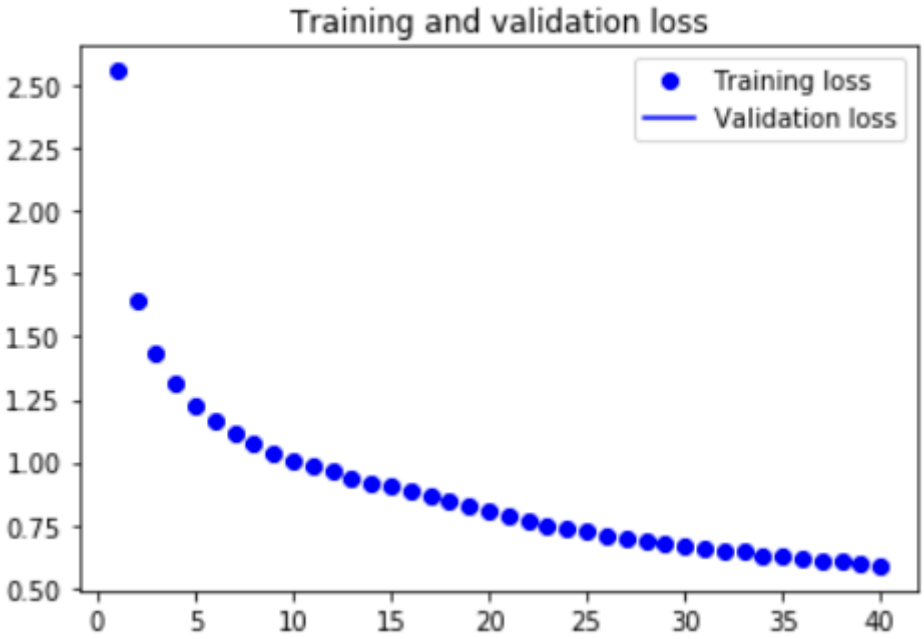
```
print('Model Prediction:')
print(logits_to_text(model.predict(tmp_x[:1])[0], french_tokenizer))

print('French Translation (Ground Truth):')
print(french_sentences[:1])

print('English Text:')
print(english_sentences[:1])
```

The function `bd_model` builds a bidirectional RNN model.

You can plot your training and validation losses as follows:

```
def plotting_results(history):
    """
    Plot results of a model trainig
    """
    acc = history.history['acc']

    val_acc = history.history['val_acc']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(1, len(acc) + 1)

    plt.plot(epochs, acc, 'bo', label='Training acc')
    plt.plot(epochs, val_acc, 'b', label='Validation acc')
    plt.title('Training and validation accuracy')
    plt.legend()

    plt.figure()

    plt.plot(epochs, loss, 'bo', label='Training loss')
    plt.plot(epochs, val_loss, 'b', label='Validation loss')
    plt.title('Training and validation loss')
    plt.legend()

    plt.show()
```






```
history = simple_rnn_model.fit(tmp_x,
                               preproc_french_sentences,
                               batch_size=1024,
                               epochs=40,
                               validation_split=0.2
                              )

plotting_results(history)
```

The Bidirectional RNN is trained on the dataset. A prediction using the model on the training dataset is printed in the notebook.

```
Epoch 7/10
110288/110288 [==============================] - 54s 487us/step - loss:
0.5118 - acc: 0.8338 - val_loss: nan - val_acc: 0.8444
Epoch 8/10
110288/110288 [==============================] - 54s 487us/step - loss:
0.4749 - acc: 0.8466 - val_loss: nan - val_acc: 0.8415
Epoch 9/10
110288/110288 [==============================] - 54s 488us/step - loss:
0.4432 - acc: 0.8565 - val_loss: nan - val_acc: 0.8595
Epoch 10/10
110288/110288 [==============================] - 54s 487us/step - loss:
0.4185 - acc: 0.8639 - val_loss: nan - val_acc: 0.8667
new jersey est parfois calme au mois automne l' automne et il en avril <P
AD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
```

✅ The bidirectional model successfully trains on the dataset and achieves reliable validation accuracy.

✅ Model is tested on a sequence and the corresponding predictions are printed in the notebook.

💡 Additional Reading:
1) Bidirectional LSTMs from scratch
2) A small chapter on Bidirectional RNNs from Dive into Deep Learning book.

The function `model_final` builds and trains a model that incorporates embedding, and bidirectional RNN using the dataset.

```
def model_final(input_shape, output_sequence_length, english_vocab_size, fr
ench_vocab_size):
    """
    Build and train a model that incorporates embedding, encoder-decoder, a
nd bidirectional RNN on x and y
    :param input_shape: Tuple of input shape
    :param output_sequence_length: Length of output sequence
    :param english_vocab_size: Number of unique English words in the datase
t
    :param french_vocab_size: Number of unique French words in the dataset
    :return: Keras model built, but not trained
    """
    # TODO: Implement
    input_sequence = Input(shape=(input_shape[1:]))
    embedding = Embedding(input_dim=english_vocab_size, output_dim=64)(inpu
t_sequence)
    rnn = Bidirectional(GRU(256, return_sequences=True))(embedding)
    rnn = Bidirectional(GRU(256))(rnn)
    repeatvector = RepeatVector(21)(rnn)
    dense = Dense(french_vocab_size, activation='softmax')
    decoder = Bidirectional(GRU(256, return_sequences=True))(repeatvector)
    outputs = TimeDistributed(dense)(decoder)
    model = Model(inputs=input_sequence, outputs=outputs)
```

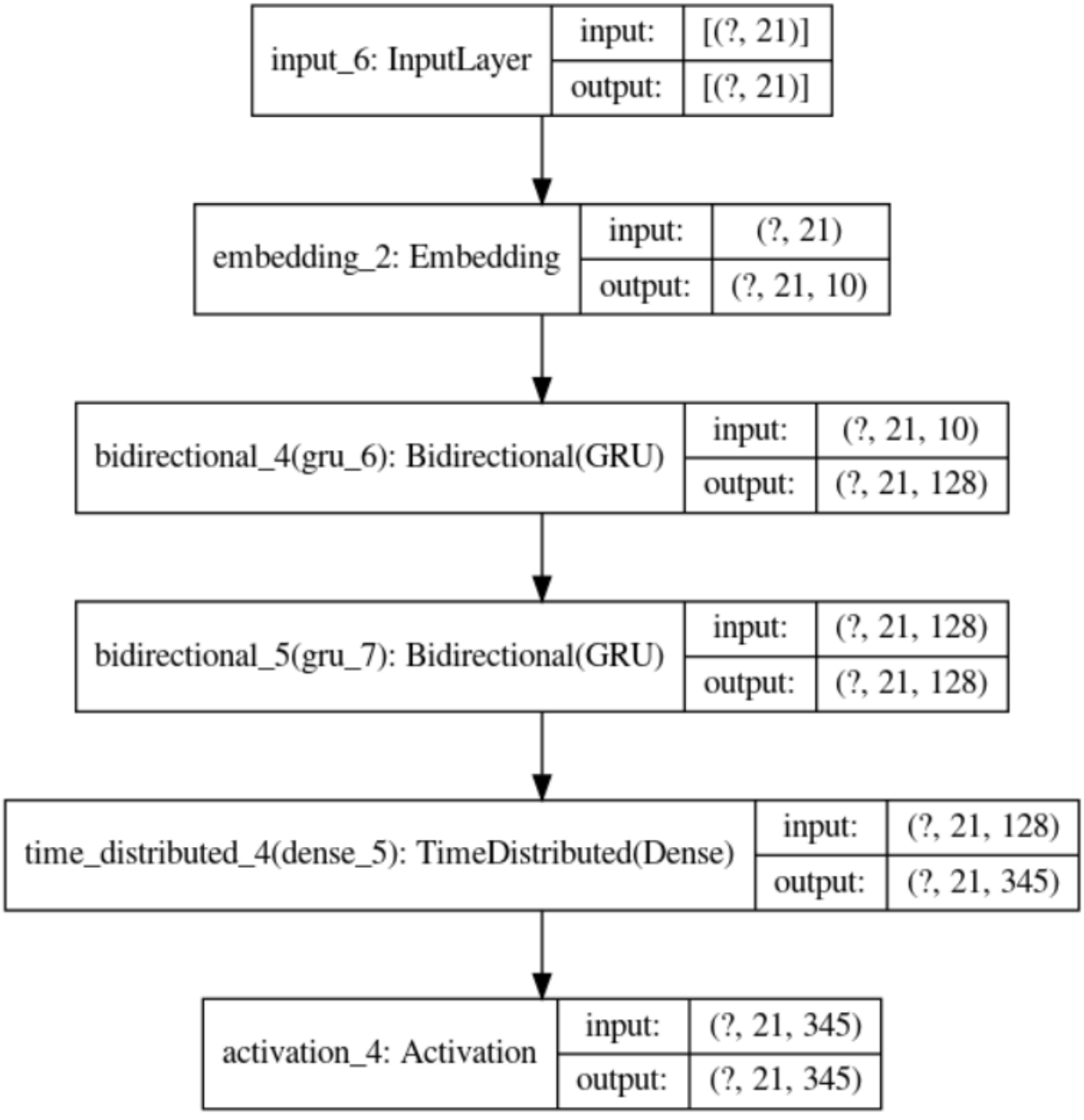
```
    model.compile(loss=sparse_categorical_crossentropy,
                  optimizer=Adam(lr=0.001),
                  metrics=['accuracy'])
    return model
```

Embedding layers and Bidirectional layers are used in building the final model and sensible hyperparameters have been chosen for the model. 👏🏻

💡 Suggestion:

You could also generate a visualisation of the model as shown below

```
from keras.utils import plot_model
from IPython.display import Image

plot_model(final_rnn_model, show_shapes=True, show_layer_names=True, to_file='fin
al_model.png')
Image('final_model.png')
```



Prediction

The final model correctly predicts both sentences.

Both sentences are correctly predicted ✅

```
Sample 1:
il a vu un vieux camion jaune <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
Il a vu un vieux camion jaune
Sample 2:
new jersey est parfois calme pendant l' automne et il est neigeux en avril <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
new jersey est parfois calme pendant l' automne et il est neigeux en avril <PAD> <PAD> <PAD> <PAD> <PAD> <PAD> <PAD>
```

DOWNLOAD PROJECT

RETURN TO PATH

Rate this review

START