Name – Dhruv Singhal **B.Tech. (C.S.E) AI&ML**

SAP ID – 500075346 **Roll No. - R177219074**

Lab Experiment 09

Study of Model in NMT

Models

In addition to standard dimension settings like the number of layers, the hidden dimension size, etc

Encoders¶

Default encoder¶

The default encoder is a simple recurrent neural network (LSTM or GRU).

Bidirectional encoder¶

The bidirectional encoder consists of two independent encoders: one encoding the normal sequence and the other the reversed sequence. The output and final states are concatenated or summed depending on the brnn_merge option.

Pyramidal deep bidirectional encoder¶

The pyramidal deep bidirectional encoder is an alternative bidirectional encoder that reduces the time dimension after **each** layer based on the -pdbrnn_reduction factor and using -pdbrnn_merge as the reduction action (sum or concatenation)

Deep bidirectional encoder¶

The deep bidirectional encoder is an alternative bidirectional encoder where the outputs of every layers are summed (or concatenated) prior feeding to the next layer. It is a special case of a pyramidal deep bidirectional encoder without time reduction (i.e. -pdbrnn_reduction = 1).

Google's NMT encoder¶

The Google encoder is an encoder with a single bidirectional layer as described in bidirectional states are concatenated and residual connections are enabled by default.

Convolutional encoder¶

The convolutional encoder) is an encoder based on several convolutional layers .
In sequence-to-sequence models, it should be used either without a bridge or with a dense bridge. The default copy bridge is not compatible with this encoder.
It is also recommended to set a small learning rate when using SGD (e.g. -learning rate 0.1) or use Adam instead

Decoders¶

Default decoder¶

The default decoder applies attention over the source sequence and implements input feeding by default.

Input feeding is an approach to feed attentional vectors "*as inputs to the next time steps to inform the model about past alignment decisions*". This can be disabled by setting -input feed 0.

Residual connections¶

With residual connections the input of a layer is element-wise added to the output before feeding to the next layer. This approach proved to be useful for the gradient flow with deep RNN stacks (more than 4 layers).

Encoder-Decoder Model in NMT

Encoder-Decoder models are popular models for sequence-to-sequence tasks. They have many applications such as image captioning, neural machine translation, and creating chatbots. At their core, they consist of an encoder RNN which generates a 'context' vector from the input sequence and a decoder RNN which generates an output sequence based off the context.

Encoder

The encoder is just an RNN, typically an LSTM or GRU, which generates a 'context' vector. In the encoder RNN, we do not care about the outputs. Rather, we want to preserve the internal state of the RNN. The final internal state is what we refer to as the 'context' vector. This is what the decoder uses to return an output sequence.

Decoder

The decoder is another RNN which uses the 'context' vector (the final hidden state of the encoder RNN) to generate an output. In training, a start of sequence token signals the decoder to begin the translation. It then uses the output and the hidden states of the current timestep to produce the output and hidden states of the next timestep.

Problems With Long Term Dependencies

Encoder-Decoder models struggle to handle long input sequences. Although in theory, LSTMs were designed to handle long term dependencies, they does not work perfectly in practice. For example, say the encoder input is an English sentence that has 50 words. The encoder has to represent all this information into a fixed size vector. The decoder then has to use this vector and remember the information from 50 timesteps ago. To fix this issue, a strategy called 'Attention' is used.

Attention mechanism- basic working

Attention is an upgrade to the existing network of sequence-to-sequence models that address this limitation. The simple reason why it is called 'attention' is because of its ability to obtain significance in sequences.

First, it works by providing a more weighted or more signified context from the encoder to the decoder and a learning mechanism where the decoder can interpret were to actually give more ‘attention’ to the subsequent encoding network when predicting outputs at each time step in the output sequence.

We can consider that by using the attention mechanism, there is this idea of freeing the existing encoder-decoder architecture from the fixed-short-length internal representation of text. This is achieved by keeping the intermediate outputs from the encoder LSTM network which correspond to a certain level of significance, from each step of the input sequence and at the same time training the model to learn and give selective attention to these intermediate elements and then relate them to elements in the output sequence.

CODE SNIPPETS:

```
from sklearn.model_selection import train_test_split
X, y = lines['english'], lines['hindi']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,random_state=42)
X_train.shape, X_test.shape
```

```
((23799,), (5950,))
```

```
X_train
```

```
11625     add remote
5569      base card
17025     step
27010     push button
18313     toolbar
            ...
21749     shortcut
5432
864       lo gin helper
15948     new library…
23836     plugin dependencies
Name: english, Length: 23799, dtype: object
```

```
encoder_input_data = np.zeros((2, max_length_src),dtype='float32')
decoder_input_data = np.zeros((2, max_length_tar),dtype='float32')
decoder_target_data = np.zeros((2, max_length_tar, num_decoder_tokens),dtype='float32')
```

Name – Dhruv Singhal **B.Tech. (C.S.E) AI&ML**

SAP ID – 500075346 **Roll No. - R177219074**

```
def generate_batch(X = X_train, y = y_train, batch_size = 128):
    ''' Generate a batch of data '''
    while True:
        for j in range(0, len(X), batch_size):
            encoder_input_data = np.zeros((batch_size, max_length_src),dtype='float32')
            decoder_input_data = np.zeros((batch_size, max_length_tar),dtype='float32')
            decoder_target_data = np.zeros((batch_size, max_length_tar, num_decoder_tokens),dtype='float32')
            for i, (input_text, target_text) in enumerate(zip(X[j:j+batch_size], y[j:j+batch_size])):
                for t, word in enumerate(input_text.split()):
                    encoder_input_data[i, t] = input_token_index[word] # encoder input seq
                for t, word in enumerate(target_text.split()):
                    if t<len(target_text.split())-1:
                        decoder_input_data[i, t] = target_token_index[word] # decoder input seq
                    if t>0:
                        # decoder target sequence (one hot encoded)
                        # does not include the START_ token
                        # Offset by one timestep
                        decoder_target_data[i, t - 1, target_token_index[word]] = 1.
            yield([encoder_input_data, decoder_input_data], decoder_target_data)
```

```
latent_dim = 300
# Encoder
encoder_inputs = Input(shape=(None,))
enc_emb =  Embedding(num_encoder_tokens+1, latent_dim, mask_zero = True)(encoder_inputs)
encoder_lstm = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder_lstm(enc_emb)
# We discard `encoder_outputs` and only keep the states.
encoder_states = [state_h, state_c]
```

```
# Set up the decoder, using `encoder_states` as initial state.
decoder_inputs = Input(shape=(None,))
dec_emb_layer = Embedding(num_decoder_tokens+1, latent_dim, mask_zero = True)
dec_emb = dec_emb_layer(decoder_inputs)
# We set up our decoder to return full output sequences,
# and to return internal states as well. We don't use the
# return states in the training model, but we will use them in inference.
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
decoder_outputs, _, _ = decoder_lstm(dec_emb,
                                     initial_state=encoder_states)
decoder_dense = Dense(num_decoder_tokens, activation='softmax')
decoder_outputs = decoder_dense(decoder_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)
```

```
model.compile(optimizer='adam', loss='categorical_crossentropy',metrics=['accuracy'])
```

```
model.summary()
train_samples = len(X_train)
val_samples = len(X_test)
batch_size = 64
epochs = 10
```

```
Model: "model_1"
```

Name – Dhruv Singhal **B.Tech. (C.S.E) AI&ML**

SAP ID – 500075346 **Roll No. - R177219074**

```
model.save('eng-to-hindi.h5')
```

```
a, b = next(generate_batch())
```

```
b
```

```
model.fit_generator(generator = generate_batch(X_train, y_train, batch_size = batch_size),
                    steps_per_epoch = train_samples/batch_size,
                    epochs=10,
                    validation_data = generate_batch(X_test, y_test, batch_size = batch_size),
                    validation_steps = val_samples/batch_size)
```