Deep Learning 2019

Assignment # 5

Report

Topic: Seq2seq Chatbot

➢ Chatbot:

Submission:

In this assignment we are required to make a chatbot with multiple variations. We are required to submit 8 notebooks out of which I'm submitting 6 as I couldn't complete beam search decoder. I was writing the code of beam search decoder but couldn't complete it for lack of time as we had exams. I'm submitting following notebooks:

1) GRU with greedy search decoder
2) Evaluation notebook of GRU with greedy decoder
3) LSTM with greedy search decoder
4) Evaluation notebook of LSTM with greedy search decoder
5) GRU with beam search decoder (there's an error in beam search decoder, I didn't have time to debug))
6) LSTM with beam search decoder (there's an error in beam search decoder, I didn't have time to debug)

I have submitted trained models of GRU with Greedy Decoder and LSTM with Greedy Decoder. I've directly uploaded the models from my drive as each model is ~300Mbs, there won't be any issue accessing them but if you face any issue please let me know I'll share them with you personally. I haven't submitted the models and evaluation notebooks for beam search decoder as the training phase was incomplete.

Environment:

I've performed all experiments on colab and accessed dataset from my drive. As colab imports latest versions of all libraries so I ran in to some issues especially in the implementation of LSTM as there was some compatibility issue. It took a lot of time as installations on colab were a bit difficult but I finally fixed it and the code started working.

Resources Used:

I tried to implement the complete code on my own and I did but there were lot of syntax problems that I couldn't fix on my own so I followed pytorch's documentation and tutorials for bi-directional rnns to rectify the syntax issues.

Evaluation Measure:

I've performed the evaluation using BLEU score. I split the data of questions and answers to training and testing sets. Total pairs for training were 49638 after all the pre-processing in all of the experiments. Testing was performed on 4271 pairs with the assumption that there was no out of vocabulary word in the test data. This problem could have been rectified by adding <unk> character in my vocabulary to model out of vocabulary words but I couldn't do it for lack of time. So when I encountered any OOV (out of vocabulary) word in any question from the test data I excluded that pair from the evaluation and it didn't play any role in the computation of BLEU score. I've reported mean bleu score and max bleu score achieved with each model.

➢ Models used in Chatbot:

Chatbot is designed using seq2seq model. That means entire sequences are modelled instead of words. This functionality is achieved by training on Q/A pairs.

➢ GRU MODEL:

GRU is gated recurrent unit, specially designed to model long distance dependencies efficiently.

Embedding:

This is a method to compute relations between different words. The easiest explanation is to keep in mind the saying: “Every word is known by the company it keeps”. This helps us getting synonymous words or the words used in same settings/location. In this method a fixed length vector of numbers is learned for each word in the vocabulary depending on its use. I’ve defined an embedding vector of size 500 for each word. Different vector sizes can be tested to check which one performs better.

Encoder:

Bi-directional GRU unit is used with 500 hidden size, number of layers 2 and dropout of 0.1.

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding

        # Initialize GRU; the input_size and hidden_size params are both set to 'hidden_size'
        #   because our input size is a word embedding with number of features == hidden_size
        self.gru = nn.GRU(hidden_size,hidden_size,n_layers,dropout=(0 if n_layers == 1 else dropout), bidirectional=True)


    def forward(self, input_seq, input_lengths, hidden=None):
        # Convert word indexes to embeddings
        embeddings = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module
        pack_padded_seq = torch.nn.utils.rnn.pack_padded_sequence(embeddings, input_lengths)
        # Forward pass through GRU
        gru_output, gru_hidden = self.gru(pack_padded_seq, hidden)
        # Unpack padding
        unpack_padded_seq, sec = torch.nn.utils.rnn.pad_packed_sequence(gru_output)
        # Sum bidirectional GRU outputs
        outputs = unpack_padded_seq[:, :, :self.hidden_size] + unpack_padded_seq[:, : ,self.hidden_size:]
        # Return output and final hidden state
        hidden = gru_hidden
        return outputs, hidden
```

Decoder:

The decoder uses attention mechanism, 2 hidden layers, uni-directional GRU of size 500, dropout of 0.1 and then I added 3 fully connected layers for getting final sequence.

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers such as embedding, dropout, GRU etc.
        self.embedding = embedding
        self.dropout = nn.Dropout(dropout)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers, dropout=(0 if n_layers == 1 else dropout))#uni-directional GRU
        self.fc1 = nn.Linear(hidden_size * 2, hidden_size) #from above attention module
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)


        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # Note: we run this one step (word) at a time
        # Get embedding of current input word
        embedding = self.embedding(input_step)
        embedding = self.dropout(embedding)

        # Forward through unidirectional GRU
        gru_output, gru_hidden = self.gru(embedding, last_hidden)
        # Calculate attention weights from the current GRU output
        weights = self.attn(gru_output, encoder_outputs)
        # Multiply attention weights to encoder outputs to get new "weighted sum" context vector
        context_vector = torch.bmm(weights,encoder_outputs.transpose(0, 1))
        # Concatenate weighted context vector and GRU output using Luong eq. 5
        gru_output = gru_output.squeeze(0)
        context_vector = context_vector.squeeze(1)
        concatenated = torch.cat((gru_output, context_vector), 1)
        fc1output = self.fc1(concatenated) #pass through first fc layer
        fc1output = torch.tanh(fc1output)
        fc2output = self.fc2(fc1output)
        fc2output = torch.tanh(fc2output)
        output = self.fc3(fc2output)
        output = F.softmax(output, dim=1)
        # Predict next word using Luong eq. 6
        # Return output and final hidden state
        hidden = gru_hidden
        return output, hidden
```

Results and Evaluation:

Iterations	5000
Encoder optimizer	Adam
Decoder optimizer	Adam
Learning rate	0.0001
Decoding	Greedy
Average BLEU-score	0.17
Maximum BLEU-score	0.57

```
Building optimizers ...
Starting Training!
Initializing ...
Training...
Iteration: 200; Percent complete: 4.0%; Average loss: 4.7449
Iteration: 400; Percent complete: 8.0%; Average loss: 4.0933
Iteration: 600; Percent complete: 12.0%; Average loss: 3.8665
Iteration: 800; Percent complete: 16.0%; Average loss: 3.6922
Iteration: 1000; Percent complete: 20.0%; Average loss: 3.6053
Iteration: 1200; Percent complete: 24.0%; Average loss: 3.5473
Iteration: 1400; Percent complete: 28.0%; Average loss: 3.4729
Iteration: 1600; Percent complete: 32.0%; Average loss: 3.4202
Iteration: 1800; Percent complete: 36.0%; Average loss: 3.3896
Iteration: 2000; Percent complete: 40.0%; Average loss: 3.3310
Iteration: 2200; Percent complete: 44.0%; Average loss: 3.2759
Iteration: 2400; Percent complete: 48.0%; Average loss: 3.2220
Iteration: 2600; Percent complete: 52.0%; Average loss: 3.1663
Iteration: 2800; Percent complete: 56.0%; Average loss: 3.1414
Iteration: 3000; Percent complete: 60.0%; Average loss: 3.1037
Iteration: 3200; Percent complete: 64.0%; Average loss: 3.0531
Iteration: 3400; Percent complete: 68.0%; Average loss: 3.0216
Iteration: 3600; Percent complete: 72.0%; Average loss: 2.9560
Iteration: 3800; Percent complete: 76.0%; Average loss: 2.9210
Iteration: 4000; Percent complete: 80.0%; Average loss: 2.8797
Iteration: 4200; Percent complete: 84.0%; Average loss: 2.8441
Iteration: 4400; Percent complete: 88.0%; Average loss: 2.7850
Iteration: 4600; Percent complete: 92.0%; Average loss: 2.7435
Iteration: 4800; Percent complete: 96.0%; Average loss: 2.6992
Iteration: 5000; Percent complete: 100.0%; Average loss: 2.6506
```

```
# Begin chatting (uncomment and run the following line to begin)
evaluateInput(encoder, decoder, searcher, voc)
```

```
> can t you go to sleep ?
answer:
no . . . . .
> why not
answer:
i m not gonna see you .
> please see me
answer:
i m sorry i m sorry . you want
```

➢ LSTM MODEL:

LSTM is Long Short Term Memory model, specially designed to model long distance dependencies. LSTM's implementation was a bit different as it has an additional memory unit so it returns not only previous hidden state but also previous memory.

Embedding:

This is a method to compute relations between different words. The easiest explanation is to keep in mind the saying: "Every word is known by the company it keeps". This helps us getting synonymous words or the words used in same settings/location. In this method a fixed length vector of numbers is learned for each word in the vocabulary depending on its use. I've defined an embedding vector of size 500 for each word. Different vector sizes can be tested to check which one performs better.

Encoder:

Bi-directional LSTM unit is used with 500 hidden size, number of layers 2 and dropout of 0.1.

```
class EncoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding, n_layers=1, dropout=0):
        super(EncoderRNN, self).__init__()
        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.embedding = embedding
        #init.normal_(self.embedding.weight, 0.0, 0.2)
        # Initialize GRU; the input_size and hidden_size params are both set to 'hidden_size'
        #   because our input size is a word embedding with number of features == hidden_size
        self.LSTM = nn.LSTM(hidden_size,hidden_size,num_layers=n_layers,dropout=(0 if n_layers == 1 else dropout), bidirectional=True)


    def forward(self, input_seq, input_lengths, hidden=None):
        # Convert word indexes to embeddings
        embeddings = self.embedding(input_seq)
        # Pack padded batch of sequences for RNN module
        #pack_padded_seq = torch.nn.utils.rnn.pack_padded_sequence(embeddings, input_lengths)
        # Forward pass through GRU
        LSTM_output, LSTM_hidden = self.LSTM(embeddings, hidden)
        # Unpack padding
        #unpack_padded_seq, _ = torch.nn.utils.rnn.pad_packed_sequence(LSTM_output)
        # Sum bidirectional GRU outputs
        #outputs = unpack_padded_seq[:, :, :self.hidden_size] + unpack_padded_seq[:, : ,self.hidden_size:]
        outputs = LSTM_output[:, :, :self.hidden_size] + LSTM_output[:, : ,self.hidden_size:]
        # Return output and final hidden state
        hidden = LSTM_hidden
        return outputs, hidden
```

Decoder:

The decoder uses attention mechanism, 2 hidden layers, uni-directional LSTM of size 500, dropout of 0.1 and then I added 3 fully connected layers for getting final sequence.

```
class LuongAttnDecoderRNN(nn.Module):
    def __init__(self, attn_model, embedding, hidden_size, output_size, n_layers=1, dropout=0.1):
        super(LuongAttnDecoderRNN, self).__init__()

        # Keep for reference
        self.attn_model = attn_model
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout = dropout

        # Define layers such as embedding, dropout, GRU etc.
        self.embedding = embedding
        #init.normal_(self.embedding.weight, 0.0, 0.2)
        self.dropout = nn.Dropout(dropout)
        self.LSTM = nn.LSTM(hidden_size, hidden_size, num_layers=n_layers, dropout=(0 if n_layers == 1 else dropout))#uni-directional GRU
        print(self.LSTM)
        self.fc1 = nn.Linear(hidden_size * 2, hidden_size) #from above attention module
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)


        self.attn = Attn(attn_model, hidden_size)

    def forward(self, input_step, last_hidden, encoder_outputs):
        # Note: we run this one step (word) at a time
        # Get embedding of current input word
        embedding = self.embedding(input_step)
        embedding = self.dropout(embedding)

        # Forward through unidirectional GRU
        LSTM_output, LSTM_hidden = self.LSTM(embedding, last_hidden)
        # Calculate attention weights from the current GRU output
        weights = self.attn(LSTM_output, encoder_outputs)
        # Multiply attention weights to encoder outputs to get new "weighted sum" context vector
        context_vector = torch.bmm(weights,encoder_outputs.transpose(0, 1))
        # Concatenate weighted context vector and GRU output using Luong eq. 5
        LSTM_output = LSTM_output.squeeze(0)
        context_vector = context_vector.squeeze(1)
        concatenated = torch.cat((LSTM_output, context_vector), 1)
        fc1output = self.fc1(concatenated) #pass through first fc layer
        fc1output = torch.tanh(fc1output)
        fc2output = self.fc2(fc1output)
        fc2output = torch.tanh(fc2output)
        output = self.fc3(fc2output)
        output = F.softmax(output, dim=1)
        # Predict next word using Luong eq. 6
        # Return output and final hidden state
        hidden = LSTM_hidden
        return output, hidden
```

Results and Evaluation:

Iterations	5000
Encoder optimizer	Adam
Decoder optimizer	Adam
Learning rate	0.0001
Decoding	Greedy
Average BLEU-score	0.19
Maximum BLEU-score	1

```
Building optimizers ...
Starting Training!
Initializing ...
Training...
Iteration: 200; Percent complete: 4.0%; Average loss: 4.9371
Iteration: 400; Percent complete: 8.0%; Average loss: 4.3057
Iteration: 600; Percent complete: 12.0%; Average loss: 4.1178
Iteration: 800; Percent complete: 16.0%; Average loss: 4.0109
Iteration: 1000; Percent complete: 20.0%; Average loss: 3.9630
Iteration: 1200; Percent complete: 24.0%; Average loss: 3.9141
Iteration: 1400; Percent complete: 28.0%; Average loss: 3.8487
Iteration: 1600; Percent complete: 32.0%; Average loss: 3.8039
Iteration: 1800; Percent complete: 36.0%; Average loss: 3.7605
Iteration: 2000; Percent complete: 40.0%; Average loss: 3.7284
Iteration: 2200; Percent complete: 44.0%; Average loss: 3.7147
Iteration: 2400; Percent complete: 48.0%; Average loss: 3.6917
Iteration: 2600; Percent complete: 52.0%; Average loss: 3.6585
Iteration: 2800; Percent complete: 56.0%; Average loss: 3.6380
Iteration: 3000; Percent complete: 60.0%; Average loss: 3.6160
Iteration: 3200; Percent complete: 64.0%; Average loss: 3.5754
Iteration: 3400; Percent complete: 68.0%; Average loss: 3.5605
Iteration: 3600; Percent complete: 72.0%; Average loss: 3.5402
Iteration: 3800; Percent complete: 76.0%; Average loss: 3.5176
Iteration: 4000; Percent complete: 80.0%; Average loss: 3.4925
Iteration: 4200; Percent complete: 84.0%; Average loss: 3.4683
Iteration: 4400; Percent complete: 88.0%; Average loss: 3.4458
Iteration: 4600; Percent complete: 92.0%; Average loss: 3.4232
Iteration: 4800; Percent complete: 96.0%; Average loss: 3.3951
Iteration: 5000; Percent complete: 100.0%; Average loss: 3.3724
```

```
# Begin chatting (uncomment and run the following line to begin)
evaluateInput(encoder, decoder, searcher, voc)
```

```
> can t you go to sleep ?
answer:
no . . . . .
> why not
answer:
i don t . . . .
> you should
answer:
i don t . . . .
> what is your name
answer:
i m going to the window . .
> which window ?
answer:
the current . . . .
```

➢ Comparison of LSTM and GRU performance:

LSTM Average BLEU-score	0.19
GRU Average BLEU-score	0.17
LSTM Maximum BLEU-score	1
GRU Maximum BLEU-score	0.57

As is evident from the above model LSTM is performing better in this seq2seq model. Average Bleu score is also better for LSTM and maximum score is 1 which means LSTM model was able to some sequences perfectly. This data was very less. If we have more data and diverse data, I believe result can be improved. Also I trained both models for 5000 iterations and even at the last iteration the loss was still decreasing and overfitting didn't happen, so I think if run for more iterations result will improve further.

➢ Final Thoughts:

We've done so many assignments on image data. This assignment was important in the sense that it gave an overview of how to apply deep learning on text data and not just deep learning but seq2seq modelling, GR, LSTM which can be used for any sequential data. Also chatbot is a very hot shot application nowadays and around $2bn industry, so I hope doing an assignment of this real-world application will help us in future in case we are to model any such problem. I couldn't understand the working too well as so much of my time went into understanding the given code and then making addition to the same code but from learning perspective the assignment was novel.

➢References:

https://arxiv.org/pdf/1508.04025.pdf

https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/02-intermediate/bidirectional_recurrent_neural_network/main.py

https://github.com/pytorch/tutorials/blob/master/beginner_source/chatbot_tutorial.py

https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

https://pytorch.org/tutorials/beginner/chatbot_tutorial.html