AI6127: Deep Neural Networks for Natural Language Processing

ASSIGNMENT 2: SEQUENCE-TO-SEQUENCE WITH ATTENTION

Chen Yongquan | G2002341D

2aii.

This is the code for randomly splitting data into 5 subsets:

```
def prepareData5Fold(lang1, lang2, reverse=False):
    input_lang, output_lang, pairs = readLangs(lang1, lang2, reverse)
    print("Read %s sentence pairs" % len(pairs))

    # collect test pairs
    num_test = int(len(pairs)*0.2)
    random.shuffle(pairs)

    input_langs = []
    output_langs = []
    train_pairs = []
    test_pairs = []
    for i in range(5):
      print("Processing set %d..." % (i + 1))
      if i == 0:
        train_pairs.append(pairs[num_test:])
        test_pairs.append(pairs[:num_test])
      elif i == 4:
        train_pairs.append(pairs[:i * num_test])
        test_pairs.append(pairs[i * num_test:])
      else:
        train_pairs.append(pairs[:i * num_test] + pairs[(i + 1) * num_test:])
        test_pairs.append(pairs[i * num_test:(i + 1) * num_test])
      input_langs.append(Lang(input_lang.name))
      output_langs.append(Lang(output_lang.name))
      for pair in train_pairs[i]:
        input_langs[i].addSentence(pair[0])
        output_langs[i].addSentence(pair[1])

    subsets = []
    for i in range(5):
      subsets.append({
        'input_lang': input_langs[i],
        'output_lang': output_langs[i],
        'train_pairs': train_pairs[i],
        'test_pairs': test_pairs[i]
      })
      print('')
      print('Subset %d:' % (i + 1))
      print("Number of train pairs:", len(subsets[i]['train_pairs']))
      print("Number of test pairs:", len(subsets[i]['test_pairs']))
      print("Counted words:")
      print(subsets[i]['input_lang'].name, subsets[i]['input_lang'].n_words)
      print(subsets[i]['output_lang'].name, subsets[i]['output_lang'].n_words)

    return subsets
```

This is the code for test set evaluation and calculation of cumulative BLEU scores (BLEU-1, BLEU-2, BLEU-3, BLEU-4):

```
from nltk.translate.bleu_score import corpus_bleu

def evaluateBleu(encoder, decoder, input_lang, output_lang, test_pairs, max_length, print_every=1000):
    start = time.time()
    n_iters = len(test_pairs)
    references, candidates, bleu = [], [], []
    for i, pair in enumerate(test_pairs):
        iter = i + 1
        sent_eng, sents_fre = pair
        sents_fre = [sents_fre.split(' ')]
        output_words, _ = evaluate(encoder, decoder, input_lang, output_lang, sent_eng, max_length)
        references.append(sents_fre)
        candidates.append(output_words[:-1]) # Remove <EOS> token
        if iter % print_every == 0:
            print('%s (%d %d%%)' % (timeSince(start, iter / n_iters),
                                    iter, iter / n_iters * 100))
    print('%s (%d %d%%)' % (timeSince(start, 1.0),
                            n_iters-1, 100))
    bleu.append(corpus_bleu(references, candidates, weights=(1, 0, 0, 0))) # BLEU-1
    bleu.append(corpus_bleu(references, candidates, weights=(0.5, 0.5, 0, 0))) # BLEU-2
    bleu.append(corpus_bleu(references, candidates, weights=(0.33, 0.33, 0.33, 0))) # BLEU-3
    bleu.append(corpus_bleu(references, candidates)) # BLEU-4
    return bleu
```

The BLEU scores are returned as an array with BLEU-1 to BLEU-4 in ascending order. To get the average BLEU scores across five subsets, we stack the 5 returned arrays and get the average using:

```
np.average(np.stack((CS_BLEU_1, CS_BLEU_2, CS_BLEU_3, CS_BLEU_4, CS_BLEU_5)), 0)
```

We present the BLEU scores and model parameters used for the four datasets below. For each dataset we train 5 separate models using each subset. We do not filter any of datasets by a maximum sentence length. As such, the attention layer in the decoder uses the length of the longest sentence for the respective dataset. For instance, for the Czech dataset, the longest sentence length is given by:

```
CS_MAX_LENGTH = max([len(a.split(' ')) for a,b in CS_subsets[0]['train_pairs']]
                    + [len(a.split(' ')) for a,b in CS_subsets[0]['test_pairs']]) + 1
```

CZECH

For the Czech dataset, we train using these parameters for all 5 subsets for *50,000* iterations:
teacher_forcing_ratio = 0.5
hidden_size = 512
embedding_size = 512
dropout_p = 0.0
learning_rate = 0.01
Optimizer = *torch.optim.SGD*

These are the BLEU scores for the Czech dataset:

Czech	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Subset 1	0.21471153	0.07556336	0.03059952	0.01211028
Subset 2	0.22669316	0.08177491	0.03412746	0.01398683
Subset 3	0.22146324	0.07704423	0.03103799	0.01226700
Subset 4	0.20601178	0.06828818	0.02613545	0.00977268
Subset 5	0.20557047	0.07260543	0.02897470	0.01159137
Average	0.21489003	0.07505522	0.03017502	0.01194563

GERMAN

For the German dataset, we train using these parameters for all 5 subsets for *10,000* iterations:
teacher_forcing_ratio = 0.5
hidden_size = 256
embedding_size = 256
dropout_p = 0.0
learning_rate = 0.01
Optimizer = *torch.optim.SGD*

These are the BLEU scores for the German dataset:

German	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Subset 1	0.09472372	0.02535524	0.00333306	0.00049535
Subset 2	0.15310388	0.04201180	0.00691645	0.00162218
Subset 3	0.14314955	0.03546754	0.00393145	0.00058283
Subset 4	0.13768847	0.03420737	0.00528321	0.00080393
Subset 5	0.10920896	0.01932068	0.00242002	0.00034857
Average	0.12757492	0.03127253	0.00437684	0.00077057

FRENCH

For the French dataset, we train using these parameters for all 5 subsets for *10,000* iterations:
teacher_forcing_ratio = 0.5
hidden_size = 256
embedding_size = 15
dropout_p = 0.0
learning_rate = 0.01
Optimizer = *torch.optim.SGD*

These are the BLEU scores for the French dataset:

French	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Subset 1	0.08399222	0.01422584	0.00083962	0.00000000
Subset 2	0.13700216	0.03323354	0.00234166	0.00000000
Subset 3	0.09825465	0.02626911	0.00125985	0.00000000
Subset 4	0.10596270	0.02009770	0.00085318	0.00000000
Subset 5	0.10318393	0.02579763	0.00124228	0.00000000
Average	0.10567913	0.02392476	0.00130732	0.00000000

RUSSIAN

For the Russian dataset, we train using these parameters for all 5 subsets for *10,000* iterations:
teacher_forcing_ratio = 0.5
hidden_size = 256
embedding_size = 256
dropout_p = 0.0
learning_rate = 0.01
Optimizer = *torch.optim.SGD*

These are the BLEU scores for the Russian dataset:

Russian	BLEU-1	BLEU-2	BLEU-3	BLEU-4
Subset 1	0.09048324	0.02562698	0.00225848	0.00000000
Subset 2	0.06742667	0.01221874	0.00247385	0.00053839
Subset 3	0.09387307	0.02082649	0.00220163	0.00000000
Subset 4	0.08271116	0.01056125	0.00193754	0.00000000
Subset 5	0.10947957	0.02340083	0.00191932	0.00030375
Average	0.08879474	0.01852686	0.00215816	0.00016843

BLEU scores calculate similarity of candidate to references based on n-gram overlaps. As such, scores exhibit a decreasing trend from BLEU-1 to BLEU-4. We provide cumulative BLEU scores for 1 to 4-grams to describe the performance of the models clearer.

The models trained on the Czech dataset have the best scores because they are the have largest model and was also trained 5 times as long as the others. The average BLEU-1 score is 21%.

The German dataset trained using the default embedding and hidden size of 256 gave reasonable performance with non-zero BLEU-4 scores.

For the French dataset, we experimented with reducing the word embedding output size, setting it to a smaller value based on recommended formula from Google, which is the fourth root of the size of vocabulary. Noticeably, model performance fell but on average is still better than the Russian dataset for BLEU-1 and BLEU-2. It may be possible that reducing embedding size can reduce the relational expressivity of the word embeddings between word pairs, resulting in lower scores for BLEU-3 and BLEU-4.

For the Russian dataset, Russian Unicode characters do not normalize properly to ASCII characters resulting in a small Russian vocabulary as compared to the English one. Training sentences are also not normalized correctly and reduced the training sample size as well as introducing erroneous ground-truths. As such, even though all hyperparameters remain the same as the German dataset, model performance is worse with low BLEU-4 scores. However, we note that there are still subsets that are less affected by the aforementioned problem and exhibit non-zero BLEU-4 scores. BLEU-3 and BLEU-4 scores all surpass that of the French models.

2aiii.

This is the code for test set evaluation using beam search decoding:

```
from nltk.translate.bleu_score import corpus_bleu

def evaluateBleu_beam_search(encoder, decoder, input_lang, output_lang, test_pairs, max_length, beam_size, print_every=1000):
    start = time.time()
    n_iters = len(test_pairs)
    references, candidates, bleu = [], [], []
    for i, pair in enumerate(test_pairs):
        iter = i + 1
        sent_eng, sents_fre = pair
        sents_fre = [sents_fre.split(' ')]
        output_words, _ = evaluate_beam_search(encoder,
                                               decoder,
                                               input_lang,
                                               output_lang,
                                               sent_eng,
                                               max_length,
                                               beam_size=beam_size)
        references.append(sents_fre)
        candidates.append(output_words[:-1]) # Remove <EOS> token
        if iter % print_every == 0:
            print('%s (%d %d%%)' % (timeSince(start, iter / n_iters),
                                    iter, iter / n_iters * 100))
    print('%s (%d %d%%)' % (timeSince(start, 1.0),
                            n_iters-1, 100))
    bleu.append(corpus_bleu(references, candidates, weights=(1, 0, 0, 0))) # BLEU-1
    bleu.append(corpus_bleu(references, candidates, weights=(0.5, 0.5, 0, 0))) # BLEU-2
    bleu.append(corpus_bleu(references, candidates, weights=(0.33, 0.33, 0.33, 0))) # BLEU-3
    bleu.append(corpus_bleu(references, candidates)) # BLEU-4
    return bleu
```

To reduce time required for optimal hyperparameter tuning, we perform hyperparameter optimization of the beam search algorithm using only a small section of the Czech subset 1 test set (first 100 samples). The table below shows the BLEU scores obtained using different *beam_size*

beam_size	BLEU-1	BLEU-2	BLEU-3	BLEU-4
evaluateBleu()	0.171568627	0.054653902	0.021614566	0.009154652
1	0.164544236	0.052331638	0.020682919	0.008748415
2	0.200437637	0.071673387	0.03201049	0.014533712
3	0.209016393	0.074728893	0.033381626	0.015175791
4	0.221906117	0.078648181	0.03404655	0.014083547
5	0.219314938	0.077621747	0.032747206	0.013687697
6	0.217080153	0.077341241	0.032703116	0.013685126
7	0.219289827	0.077968603	0.031821036	0.013427025
8	0.219311014	0.078428294	0.032074416	0.013551785

We see that BLEU scores increase as we increase beam_size from 1 to 4, however increasing it further causes BLEU scores to start decreasing. To reduce time needed for tuning we do not repeat the experiments for the other datasets or subsets, and we use 4 as the optimal beam_size for all our beam search comparisons.

To shorten training time, we only use the first subset for each dataset to compare between greedy decoding and beam search decoding. *beam_size = 4* for all datasets. The BLEU scores for greedy decoding and beam search decoding are presented as follows:

CZECH

Czech Subset 1	*evaluateBleu()*	*evaluateBleu_beam_search()*
BLEU-1	0.214711525	0.228570162
BLEU-2	0.075563359	0.081977273
BLEU-3	0.030599521	0.034893191
BLEU-4	0.012110277	0.014763999

GERMAN

German Subset 1	*evaluateBleu()*	*evaluateBleu_beam_search()*
BLEU-1	0.094723718	0.109013662
BLEU-2	0.025355243	0.027716168
BLEU-3	0.003333059	0.005392660
BLEU-4	0.000495349	0.000866394

FRENCH

French Subset 1	*evaluateBleu()*	*evaluateBleu_beam_search()*
BLEU-1	0.083992218	0.085906979
BLEU-2	0.014225840	0.022637272
BLEU-3	0.000839618	0.001254814
BLEU-4	0.000000000	0.000000000

RUSSIAN

Russian Subset 1	*evaluateBleu()*	*evaluateBleu_beam_search()*
BLEU-1	0.090483243	0.080862522
BLEU-2	0.025626978	0.023386427
BLEU-3	0.002258484	0.001985613
BLEU-4	0.000000000	0.000000000

We can see that beam search is effective in increasing model performance across all 4 n-gram scores for the first 3 datasets, provided that an optimal *beam_size* is used. This is within our expectations as greedy decoding only selects the best candidate as the predicted word at each decoding step, which may not be the most probable one in subsequent steps. Thus, once a wrong word is predicted, there is no way to reverse the error and it is forward propagated along all subsequent steps. Conversely, beam search decoding allows for some degree of error by selecting top-k alternatives at each decoding step and choosing the most probable sentence at the end out of all alternative sentences. If the ground truth is within the top-k alternatives, even if there is a single wrongly predicted word in the middle decoding steps, the algorithm can still return the ground truth sentence based on the most probable sentence generated from at the end.

However, because beam search decoding is not an exhaustive search and is a compromise between the near intractability of exhaustive search and the reliability of greedy decoding, there is a chance that the result returned is still sub-optimal. We can see that for the Russian dataset, *evaluateBleu_beam_search()* actually gave worse performance. This may be due to a smaller dataset, within Russian having the second least sentence pairs out of all datasets. However, it is more likely due to the small Russian vocabulary size with only around 4000 unique words for all subsets compared to around 40,000 English words. Because the Russian characters do not normalize properly to ASCII characters, a large swathe of Russian vocabulary are excised from the dataset after normalization. As such, without enough training data, word embeddings may not be distinct enough from one another after training and beam search may select a worse output than greedy decoding, given that the output from greedy decoding is also not optimal.

2bi.

There are a few differences in which the AttnDecoderRNN of tutorial 6 defers from the attention decoder of lecture 6.

Firstly, the attention decoder of lecture 6 at each decoding step uses the decoder hidden state after passing through the gated recurrent unit layer for computation of the attention scores.
However, the attention decoder of the tutorial computes the attention scores and output before passing through the GRU layer, and as such uses the initial decoder hidden state at each decoding step instead.

We see that the attention weights and outputs are calculated first:

```
attn_weights = F.softmax(
    self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                         encoder_outputs.unsqueeze(0))
```

Before the intermediate outputs are passed to the GRU layer:

```
output = torch.cat((embedded[0], attn_applied[0]), 1)
output = self.attn_combine(output).unsqueeze(0)

output = F.relu(output)
output, hidden = self.gru(output, hidden)
```

Secondly, for calculation of the attention score, the attention decoder of lecture 6 only uses encoder and decoder hidden states for computation of the attention scores, while the attention decoder of the tutorial also uses the input word embedding along with the decoder hidden state to get the intermediate attention scores.

Here we see the attention scores computed by concatenating the input embeddings with initial hidden state, and multiplying them with a trainable weight matrix *self.attn* before taking the softmax of the intermediate output to give the attention scores:

```
attn_weights = F.softmax(
    self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
```

Thirdly, the attention decoder of lecture 6 takes the softmax of the matrix multiplication between the decoder and encoder hidden states as the attention distribution, which is the simple dot-product score described in Luong2015, with no trainable weights.

However, the attention decoder of tutorial 6 has two trainable linear weight matrixes, the first weight matrix, *self.attn*, is for transforming the concatenated input embedding and initial hidden state, while the second weight matrix, *self.attn_combine*, is for tranforming the concatenation of the input embedding and the intermediate attention outputs (after multiplying the first attention scores with the encoder hidden states).

First linear transformation:

```
attn_weights = F.softmax(
    self.attn(torch.cat((embedded[0], hidden[0]), 1)), dim=1)
```

Second linear transformation:

```
output = torch.cat((embedded[0], attn_applied[0]), 1)
output = self.attn_combine(output).unsqueeze(0)
```

The 2 attention decoders are similar in that they do not use the previous step attention outputs at each decoding step for token generation through the GRU layer, though the AttnDecoderRNN of tutorial 6 does return the attn_weights at the end of the forward() function. Supposing we perform attention step after the GRU layer, it should be possible to use the previous attention weights or outputs as additional features for the GRU layer, on top of using the current attention outputs for the final linear layer, as shown in the AttnDecoderRNN here: https://github.com/spro/practical-pytorch/blob/master/seq2seq-translation/seq2seq-translation.ipynb

2bii.

The attention score function from the lecture is akin to the *dot-product* attention function described in Luong2015 [1]. Meanwhile the attention decoder from the tutorial has more modelling capacity with a linear layer for transforming the concatenated input word embedding and decoder hidden state, and another linear layer for combining the attention output with the input word embedding before they are fed into the GRU layer. It is thus expected that the attention decoder from the lecture performs worse than one from the tutorial ceteris paribus.

The modified code of AttnDecoderRNN to follow that of the lecture is shown below:

```
class LectureAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding_size, output_size, dropout_p=0.1):
        super(LectureAttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size
        self.output_size = output_size
        self.dropout_p = dropout_p

        self.embedding = nn.Embedding(self.output_size, self.embedding_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.embedding_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1,1,-1)
        embedded = self.dropout(embedded)
        output = F.relu(embedded)

        output, hidden = self.gru(output, hidden)
        # Attention scores are the softmax of the simple dot-product
        # between encoder and decoder hidden states.
        # FOR UNIDIRECTIONAL GRU HERE, OUTPUT AND HIDDEN RETURN TENSORS ARE THE SAME.
        attn_weights = torch.bmm(encoder_outputs.unsqueeze(0),
                                 output.view(1,-1,1)
                                ).view(1,-1)
        attn_weights = F.softmax(attn_weights, dim=1)
        # Attention output is attention scores multiplied by encoder hidden states
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))
        # Attention outputs are concatenated with decoder hidden states at the end
        output = torch.cat((attn_applied[0], output[0]), 1)
        output = self.out(output)
        output = F.log_softmax(output, dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

Due to time constraints, we will not be training:

$$5\ subsets \times 4\ datasets \times 4\ decoders\ =\ 80\ separate\ models$$

We will only compare the results for the German dataset and for its first subset. Models are trained for *10,000* iterations and the hyperparameters used for both the original decoder and the modified decoder are as follows:

teacher_forcing_ratio = 0.5
hidden_size = 256
embedding_size = 256
dropout_p = 0.0
learning_rate = 0.01
Optimizer = *torch.optim.SGD*

The table below shows the BLEU score for the tutorial and lecture decoders using greedy decoding:

Decoder	BLEU-1	BLEU-2	BLEU-3	BLEU-4
AttnDecoderRNN	0.0947237176	0.0253552435	0.0033330591	0.0004953494
LectureAttnDecoderRNN	0.1764239442	0.0470057794	0.0085627644	0.0014762479

Converse to our hypothesis, we see that the lecture attention decoder actually performs better than the original attention decoder from the tutorial. This is not unexpected though as we did not train our models to near saturation point and halt training after 10,000 iterations, which is not even enough to cover all the training samples in German subset 1. We can postulate that without an additional attention weight matrix to train, the lecture attention decoder does converge faster than the tutorial attention decoder to saturation. However, we also do not reject our hypothesis that the additional weight matrix confers more capacity for our model and in turn better performance, as the models need to be trained for more iterations in order for the rejection to be conclusive.

2biii.

The multiplicative attention described in the question is the *general* attention function described in Luong2015 [1]. This is the code for the multiplicative attention decoder:

```
class MultiplicativeAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding_size, output_size, dropout_p=0.1):
        super(MultiplicativeAttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size
        self.output_size = output_size
        self.dropout_p = dropout_p

        self.embedding = nn.Embedding(self.output_size, self.embedding_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.embedding_size, self.hidden_size)
        self.attn_W = nn.Linear(self.hidden_size, self.hidden_size)
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)
        output = F.relu(embedded)

        output, hidden = self.gru(output, hidden)
        # Attention scores are the encoder hidden states multiplied by the
        # dot product between the weight matrix and the decoder hidden states.
        # FOR UNIDIRECTIONAL GRU HERE, OUTPUT AND HIDDEN RETURN TENSORS ARE THE SAME.
        attn_weights = self.attn_W(output)
        attn_weights = torch.bmm(encoder_outputs.unsqueeze(0),
                                 output.view(1,-1,1)
                                ).view(1,-1)
        attn_weights = F.softmax(attn_weights, dim=1)
        # Attention output is attention scores multiplied by encoder hidden states
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))
        # Attention outputs are concatenated with decoder hidden states at the end
        output = torch.cat((attn_applied[0], output[0]), 1)
        output = self.out(output)
        output = F.log_softmax(output, dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

The additive attention function is the attention function as described in Bahdanau2014 [2]. This is similar to the *concat* attention function described in Luong2015, however Luong only uses a single weight matrix for transforming the concatenation of both encoder and decoder hidden states while Bahdanau uses 2 weight matrices for transforming the encoder and decoder hidden states separately then summing them up. The weight vector v and the $tanh$ function remain the same for both. There may be minor differences between both in computational efficiency, and predictive performance resulting from the *concat* vs *addition* operation. Else, the number of learnable parameters for both are the same. This is the code for the additive attention decoder:

```
class AdditiveAttnDecoderRNN(nn.Module):
    def __init__(self, hidden_size, embedding_size, output_size, dropout_p=0.1):
        super(AdditiveAttnDecoderRNN, self).__init__()
        self.hidden_size = hidden_size
        self.embedding_size = embedding_size
        self.output_size = output_size
        self.dropout_p = dropout_p

        self.embedding = nn.Embedding(self.output_size, self.embedding_size)
        self.dropout = nn.Dropout(self.dropout_p)
        self.gru = nn.GRU(self.embedding_size, self.hidden_size)
        self.attn_Wh = nn.Linear(self.hidden_size, self.hidden_size)
        self.attn_We = nn.Linear(self.hidden_size, self.hidden_size)
        self.attn_v = nn.Parameter(torch.FloatTensor(1, self.hidden_size))
        self.out = nn.Linear(self.hidden_size * 2, self.output_size)

    def forward(self, input, hidden, encoder_outputs):
        embedded = self.embedding(input).view(1, 1, -1)
        embedded = self.dropout(embedded)
        output = F.relu(embedded)

        output, hidden = self.gru(output, hidden)

        # Attention scores are computed by first transforming both the decoder
        # and encoder hidden states with separate linear transformation matrices.
        # They are then summed together and passed through a tanh() function
        # before being multiplied by a weight vector v.
        # Finally the scores are given by taking the softmax of this result.
        # FOR UNIDIRECTIONAL GRU HERE, OUTPUT AND HIDDEN RETURN TENSORS ARE THE SAME.
        attn_weights = torch.tanh(self.attn_Wh(output)
                                  + self.attn_We(encoder_outputs))
        attn_weights = attn_weights.bmm(self.attn_v.unsqueeze(2))
        attn_weights = F.softmax(attn_weights.view(1,-1), dim=1)
        # Attention output is attention scores multiplied by encoder hidden states
        attn_applied = torch.bmm(attn_weights.unsqueeze(0),
                                 encoder_outputs.unsqueeze(0))
        # Attention outputs are concatenated with decoder hidden states at the end
        output = torch.cat((attn_applied[0], output[0]), 1)
        output = self.out(output)
        output = F.log_softmax(output, dim=1)
        return output, hidden, attn_weights

    def initHidden(self):
        return torch.zeros(1, 1, self.hidden_size, device=device)
```

We again compare the results for the German dataset for its first subset for the two attention variants against the attention from both the tutorial and lecture.

Decoder	BLEU-1	BLEU-2	BLEU-3	BLEU-4
AttnDecoderRNN	0.09472372	0.02535524	0.00333306	0.00049535
LectureAttnDecoderRNN	0.17642394	0.04700578	0.00856276	0.00147625
MultiplicativeAttnDecoderRNN	0.19035445	0.05332327	0.01647639	0.00454600
AdditiveAttnDecoderRNN	0.13881143	0.03873484	0.00930649	0.00211090

We see that the multiplicative attention decoder performs better than the lecture attention decoder across the board even though it too has an additional attention weight matrix, as was the tutorial attention decoder. This means that our postulation from Q2bii may only be partially right, while our initial hypothesis for that part also holds true. Having an additional weight matrix does indeed increase the model capacity and increase translation performance. Then, the only remaining difference the tutorial attention decoder differs from the other 3 is in the position of the attention layer and the features used for computing the attention scores and output. We can thereby postulate that either

1. putting the attention layer after the RNN layer and using the current instead of initial decoder hidden state or
2. using the encoder hidden states and current step decoder hidden states for attention scores instead of input word embedding and initial decoder hidden state

contributed to the better model performance for all latter 3 attention variants. As for our additive attention yielding lower performance than both the dot-product (Lecture attention) and multiplicative attention, our results mirror those reported by Luong: “For content-based functions, our implementation *concat* does not yield good performances and more analysis should be done to understand the reason.” [1]

References

[1] D. Bahdanau, K. Cho and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," 2014.

[2] M.-T. Luong, H. Pham and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," 2015.