CSC413 Programming Assignment Three

ZhenDi Pan 1003241823

2020-03-16

Part 1: Gated Recurrent Unit

Question 1

Note: If any of the outputs in my write-up are not consistent with my notebook, it is because I ran the training many more times for testing after finishing this write-up.

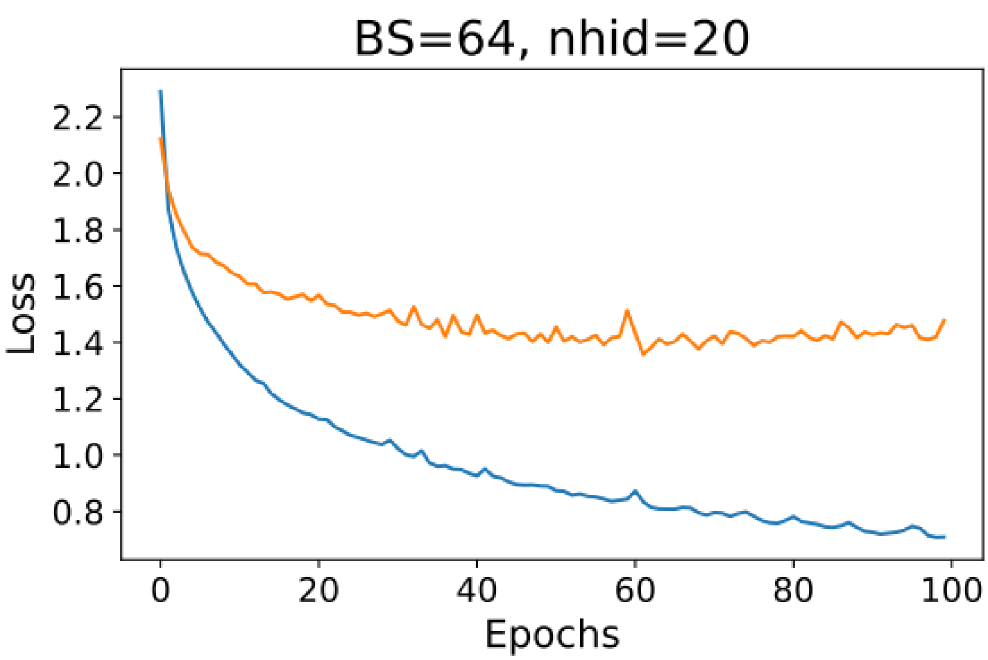
Screenshots of my full MyGRUCell implementation:

```
class MyGRUCell(nn.Module):
        def __init__(self, input_size, hidden_size):
                super(MyGRUCell, self).__init__()

                self.input_size = input_size
                self.hidden_size = hidden_size

                # ------------
                # FILL THIS IN
                # ------------
                ## Input linear layers
                self.Wiz = nn.Linear(input_size, hidden_size)
                self.Wir = nn.Linear(input_size, hidden_size)
                self.Win = nn.Linear(input_size, hidden_size)

                ## Hidden linear layers
                self.Whz = nn.Linear(hidden_size, hidden_size)
                self.Whr = nn.Linear(hidden_size, hidden_size)
                self.Whn = nn.Linear(hidden_size, hidden_size)
```

```python
def forward(self, x, h_prev):
    """Forward pass of the GRU computation for one time step.

    Arguments
        x: batch_size x input_size
        h_prev: batch_size x hidden_size

    Returns:
        h_new: batch_size x hidden_size
    """

    # ------------
    # FILL THIS IN
    # ------------
    z = torch.sigmoid(self.Wiz(x) + self.Whz(h_prev))
    r = torch.sigmoid(self.Wir(x) + self.Whr(h_prev))
    g = torch.tanh(self.Win(x) + r * self.Whn(h_prev))
    h_new = (1-z)*g + z*h_prev
    return h_new
```

Question 3

The training/validation loss plot:



After the training, the results are shown below:

```
Epoch: 99 | Train loss: 0.650 | Val loss: 1.090 | Gen: ethay airway onitinglysay isway
    onshingray
source:   the air conditioning is working
```

```
translated:  ethay airway onitinglysay isway onshingray
```

We can see the model picks up the fact that words always end in 'ay', and the rule of moving the first consonant letter to the back (although in the wrong order) and not changing the word when it is a vowel. I first changed the TEST_SENTENCE to "may the force be with you". The output is as below:

```
source:   may the force be with you

translated:  ayway ethay orceway etay ithay ybay
```

The model makes a few correct predictions. Such as "with" translated to "ithway". I also tried "the best possible solution":

```
source:   it is the best possible solution

translated:  itway isway ethay estbay ossicedblay oodinceway
```

To briefly describe our failure cases, the model seems to perform poorly on longer words, especially the ones with more than 4 letters, such as "best" translated to "etstay", "solution" translated to "oluntionway". It also sometimes simply omits letters or adds letters out of nowhere, such as "solution" translated to "oodinceway".

Part 2: Additive Attention

Question 1

Three equations:

$$\tilde{\alpha}_i^{(t)} = f(Q_t, K_i) = W_2(max(0, W_1[Q_t; K_i] + b_1)) + b_2$$

$$\alpha_i^{(t)} = softmax(\tilde{\alpha}^{(t)})_i$$

$$c_t = \sum_{i=1}^{T} \alpha_i^{(t)} V_i$$

Question 2

The forward method of the RNNAttentionDecoder class implementation is shown below:

```
def forward(self, inputs, annotations, hidden_init):
        """Forward pass of the attention-based decoder RNN.

        Arguments:
                inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
                annotations: The encoder hidden states for each step of the input.
                                sequence. (batch_size x seq_len x hidden_size)
                hidden_init: The final hidden states from the encoder, across a batch. (batch_size x hidden_size)

        Returns:
                output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding time steps.
                attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_len x decode
        """

        batch_size, seq_len = inputs.size()
        embed = self.embedding(inputs)  # batch_size x seq_len x hidden_size

        hiddens = []
        attentions = []
        h_prev = hidden_init
        for i in range(seq_len):
                # ------------
                # FILL THIS IN - START
                # ------------
                embed_current = embed[:,i,:]  # Get the current time step, across the whole batch
                context, attention_weights = self.attention(embed_current, annotations, annotations)  # batch_size x 1 x hidden_size
                embed_and_context = torch.cat((embed_current, context.squeeze(1)), 1)  # batch_size x (2*hidden_size)
                h_prev = self.rnn(embed_and_context, h_prev)  # batch_size x hidden_size

                hiddens.append(h_prev)
                attentions.append(attention_weights)

        hiddens = torch.stack(hiddens, dim=1) # batch_size x seq_len x hidden_size
        attentions = torch.cat(attentions, dim=2) # batch_size x seq_len x seq_len

        output = self.out(hiddens)  # batch_size x seq_len x vocab_size
        return output, attentions
```

Just in case this is not clear enough to see, a code listing is also included:

```
def forward(self, inputs, annotations, hidden_init):
  """Forward pass of the attention-based decoder RNN.

  Arguments:
      inputs: Input token indexes across a batch for all the time step. (batch_size x
         decoder_seq_len)
      annotations: The encoder hidden states for each step of the input.
                sequence. (batch_size x seq_len x hidden_size)
      hidden_init: The final hidden states from the encoder, across a batch. (batch_size x
         hidden_size)

  Returns:
      output: Un-normalized scores for each token in the vocabulary, across a batch for
         all the decoding time steps. (batch_size x decoder_seq_len x vocab_size)
      attentions: The stacked attention weights applied to the encoder annotations
```

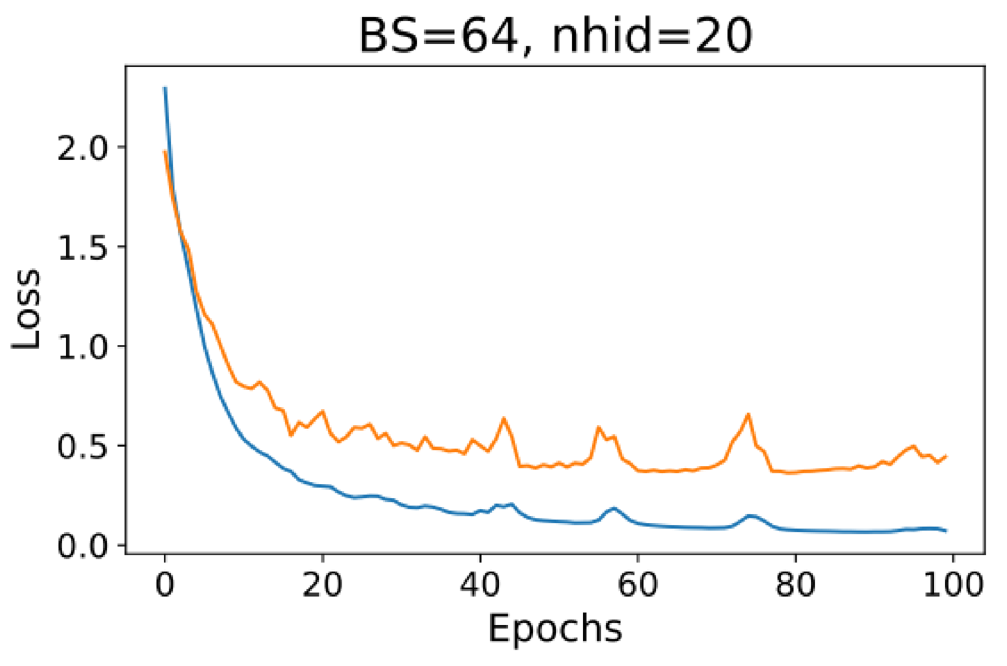
```
        (batch_size x encoder_seq_len x decoder_seq_len)
    """

    batch_size, seq_len = inputs.size()
    embed = self.embedding(inputs) # batch_size x seq_len x hidden_size

    hiddens = []
    attentions = []
    h_prev = hidden_init
    for i in range(seq_len):
        # ------------
        # FILL THIS IN - START
        # ------------
        embed_current = embed[:,i,:] # Get the current time step, across the whole batch
        context, attention_weights = self.attention(embed_current, annotations, annotations)
            # batch_size x 1 x hidden_size
        embed_and_context = torch.cat((embed_current, context.squeeze(1)), 1) # batch_size x
            (2*hidden_size)
        h_prev = self.rnn(embed_and_context, h_prev) # batch_size x hidden_size

        hiddens.append(h_prev)
        attentions.append(attention_weights)

    hiddens = torch.stack(hiddens, dim=1) # batch_size x seq_len x hidden_size
    attentions = torch.cat(attentions, dim=2) # batch_size x seq_len x seq_len

    output = self.out(hiddens) # batch_size x seq_len x vocab_size
    return output, attentions
```

Question 3

The prediction result is

```
Epoch: 99 | Train loss: 0.072 | Val loss: 0.444 | Gen: ethay airway onditingcay isway
    orkingway
source:   the air conditioning is working
translated:  ethay airway onditingcay isway orkingway
```

We can see the result is visibly better than RNN decoder without attention. The words are very similar to correct translation with some minor errors. The training/validation plot is shown below:



Problem 4

The training speed with attention is slower since there are more computations and more weights to train at each time step.

Part 3: Scaled Dot Product Attention

Question 1

ScaledDotProduct Implementation (added lines highlighted in red circle):

```
def forward(self, queries, keys, values):
        """The forward pass of the scaled dot attention mechanism.

        Arguments:
                queries: The current decoder hidden state, 2D or 3D tensor. (batch_
                keys: The encoder hidden states for each step of the input sequen
                values: The encoder hidden states for each step of the input sequ

        Returns:
                context: weighted average of the values (batch_size x k x hidden_s
                attention_weights: Normalized attention weights for each encoder hidd

                The output must be a softmax weighting over the seq_len annotation
        """
        # ------------
        # FILL THIS IN
        # ------------
        batch_size, seq_len, hidden_size = keys.size()
        q = self.Q(queries.view(-1, hidden_size)).view(batch_size, -1, hidden_size)
        k = self.K(keys.view(-1,hidden_size)).view(batch_size, seq_len, hidden_size)
        v = self.V(values.view(-1, hidden_size)).view(batch_size, seq_len, hidden_size)
        unnormalized_attention = self.scaling_factor * torch.bmm(k, q.transpose(1,2))
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(1,2), v)
        return context, attention_weights
```

Question 2

CausalScaledDotProduct (added lines highlighted in red circle):

```python
def forward(self, queries, keys, values):
    """The forward pass of the scaled dot attention mechanism.

    Arguments:
        queries: The current decoder hidden state, 2D or 3D tensor. (batch_size x
        keys: The encoder hidden states for each step of the input sequence. (bat
        values: The encoder hidden states for each step of the input sequence. (b

    Returns:
        context: weighted average of the values (batch_size x k x hidden_size)
        attention_weights: Normalized attention weights for each encoder hidden state

        The output must be a softmax weighting over the seq_len annotations.
    """

    # ------------
    # FILL THIS IN
    #
    batch_size, seq_len, hidden_size = keys.size()
    q = self.Q(queries.view(-1, hidden_size)).view(batch_size, -1, hidden_size)
    k = self.K(keys.view(-1,hidden_size)).view(batch_size, seq_len, hidden_size)
    v = self.V(values.view(-1, hidden_size)).view(batch_size, seq_len, hidden_size)
    unnormalized_attention = self.scaling_factor * torch.bmm(k, q.transpose(1,2))
    mask = torch.tril(torch.ones(batch_size,seq_len,seq_len,dtype=torch.uint8)).transpose(1,2)
    unnormalized_attention[mask==0] = self.neg_inf
    attention_weights = self.softmax(unnormalized_attention)
    context = torch.bmm(attention_weights.transpose(1,2), v)
    return context, attention_weights
```

Question 3

TransformerEncoder (added lines highlighted by red arrows):

```python
def forward(self, inputs):
    """Forward pass of the encoder RNN.

    Arguments:
        inputs: Input token indexes across a batch for all time steps in the sequence. (batch_size x

    Returns:
        annotations: The hidden states computed at each step of the input sequence. (batch_size x seq_
        hidden: The final hidden state of the encoder, for each sequence in a batch. (batch_size x h
    """

    batch_size, seq_len = inputs.size()
    # ------------
    # FILL THIS IN - START
    # ------------
    encoded = self.embedding(inputs)  # batch_size x seq_len x hidden_size

    # Add positional embeddings from self.create_positional_encodings. (a'la https://arxiv.org/pdf/1706.03762.pdf,
    encoded = encoded + self.positional_encodings[:seq_len]

    annotations = encoded

    for i in range(self.num_layers):
        new_annotations, self_attention_weights = self.self_attentions[i](annotations, annotations, annotations)
        residual_annotations = annotations + new_annotations
        new_annotations = self.attention_mlps[i](residual_annotations)
        annotations = residual_annotations + new_annotations
    # ------------
    # FILL THIS IN - END
    # ------------

    # Transformer encoder does not have a last hidden layer.
    return annotations, None
```

Question 4

TransformerDecoder (added lines highlighted by red arrows):

```
def forward(self, inputs, annotations, hidden_init):
        """Forward pass of the attention-based decoder RNN.

        Arguments:
                inputs: Input token indexes across a batch for all the time step. (batch_size x decoder_seq_len)
                annotations: The encoder hidden states for each step of the input.
                                    sequence. (batch_size x seq_len x hidden_size)
                hidden_init: Not used in the transformer decoder
        Returns:
                output: Un-normalized scores for each token in the vocabulary, across a batch for all the decoding ti
                attentions: The stacked attention weights applied to the encoder annotations (batch_size x encoder_seq_le
        """

        batch_size, seq_len = inputs.size()
        embed = self.embedding(inputs)    # batch_size x seq_len x hidden_size

        # THIS LINE WAS ADDED AS A CORRECTION.
        embed = embed + self.positional_encodings[:seq_len]

        encoder_attention_weights_list = []
        self_attention_weights_list = []
        contexts = embed
        for i in range(self.num_layers):
            # ------------
            # FILL THIS IN - START
            # ------------
            new_contexts, self_attention_weights = self.self_attentions[i](contexts, contexts, contexts)   # batch_size x se
            residual_contexts = contexts + new_contexts
            new_contexts, encoder_attention_weights = self.encoder_attentions[i](residual_contexts, annotations, annotations) #
            residual_contexts = residual_contexts + new_contexts
            new_contexts = self.attention_mlps[i](residual_contexts)
            contexts = residual_contexts + new_contexts

            # ------------
            # FILL THIS IN - END
            # ------------

            encoder_attention_weights_list.append(encoder_attention_weights)
            self_attention_weights_list.append(self_attention_weights)

        output = self.out(contexts)
```

Question 5

```
Epoch: 99 | Train loss: 0.000 | Val loss: 0.392 | Gen: ethay airway onditioningcay isway
    orkingway
source:   the air conditioning is working
translated:  ethay airway onditioningcay isway orkingway
```

We can see a decrease in both the training error and the validation error. the training error decreased to approximately zero and the prediction result is also improved, the translated result is actually perfect, corresponding to the zero training loss. Lastly, the training speed is also faster compared to previous models, which is expected.

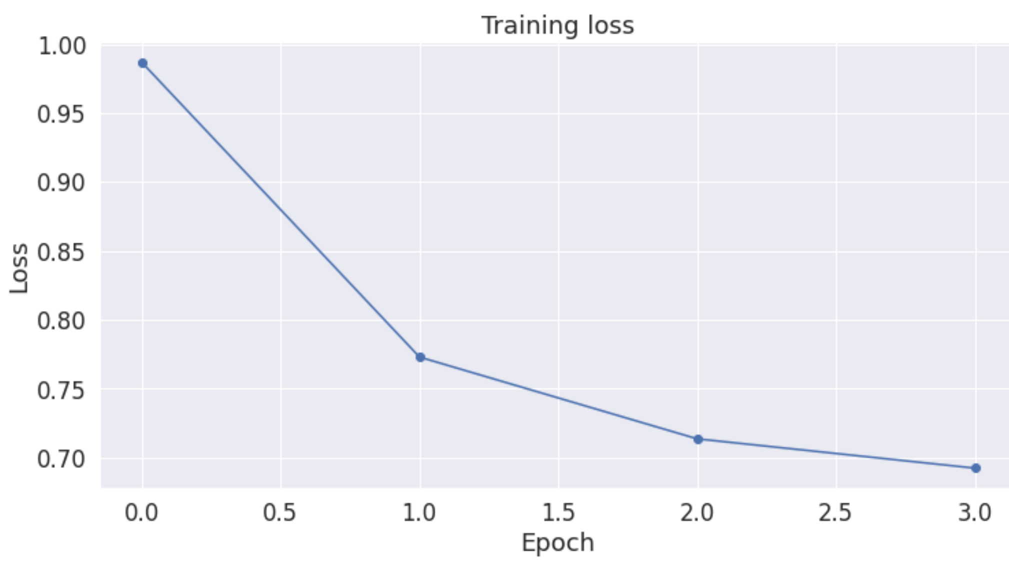
Question 6

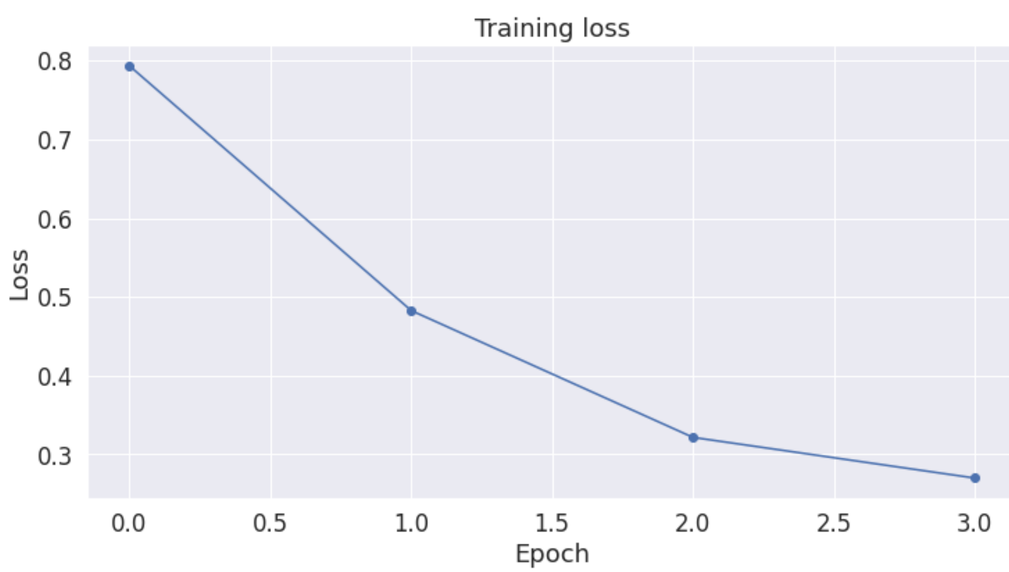
After modifying the transformer decoder __init__ to use non-causal attention for both self attention and encoder attention, the performance is much worse and both errors are very large (see code file for the modification). This is reasonable since if we don't use the casual attention for self attention, the decoder does not account for this. It doesn't know future letters whereas the encoder uses future outputs.

Part 4: BERT for arithmetic sentiment analysis

Training Curves

The two training curves:





Question 3

The ten inference results I chose:

```
what_is("0 minus 0")

negative

what_is("0 plus 0")

positive

what_is("1 plus 2 minus 4")

positive

what_is("1 minus 2 plus 4")

positive

what_is("thousand minus hundred")

negative

what_is("2 + 2 + 100 - 102")

positive

what_is("three minus two minus eight")

negative

what_is("three minus two minus one plus hundred")

positive

what_is("one minus one minus one plus 5")

positive

what_is("minus three plus three")

positive
```

I chose the first two examples because it seems that BERT does not recognize 0 as the boundary for positive and negative, so instead of giving a zero as the correct output, it seems the result is dependent on the operation. The third test case is surprisingly a failure whereas the fourth is a success, I chose them to see how well does the model handle three operations. I can't justify all my choices in three sentences, but I believe they are a fair representation of all data.

Question 4

I wrote a simple script to generate more training samples to perform data augmentation. The code is listed below:

```
from random import randrange
words = {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five', 6: 'six', 7: 'seven', 8:
    'eight', 9: 'nine', 10: 'ten'}
operations = ['plus', 'minus']
new_inputs = []
new_labels = np.zeros(100)
for i in range(0,100):
  a, b = randrange(1,11), randrange(1,11)
  first_num = words[a]
  second_num = words[b]
  operation = operations[randrange(2)]
  new_input = first_num + ' '+ operation + ' ' + second_num
  if operation == 'plus':
    new_label = a + b
  else:
    new_label = a - b
  if new_label > 0:
    new_label = 2
  if new_label == 0:
    new_label = 1
  if new_label < 0:
    new_label = 0
  new_inputs.append(new_input)
  new_labels[i] = new_label
df2 = pd.DataFrame({'input':new_inputs, 'label':new_labels})
```

After we get the new data as df2, we can simply append it to our training samples to achieve data augmentation. Obviously this script could be improved much better, but it is a simple idea that can perform data augmentation by providing more data to our training set.