CSC413PA3

Che Liu, 1002246839

March 19, 2020

Part 1: Gated Recurrent Unit (GRU) [2pt]

1. A screenshot of your full MyGRUCell implementation. [1pt]

```python
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.Wih = 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.Whh = nn.Linear(hidden_size, hidden_size)


    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
        # ------------
        sig = nn.Sigmoid()
        tanh = nn.Tanh()

        z = sig(self.Wiz(x)+self.Whz(h_prev))
        r = sig(self.Wir(x)+self.Whr(h_prev))
        g = tanh(self.Wih(x)+r*self.Whh(h_prev))
        h_new = (1-z)*g+z*h_prev
        return h_new
```

Figure 1: A screenshot of my full MyGRUCell implementation.

2. The training/validation loss plots. [0pts]

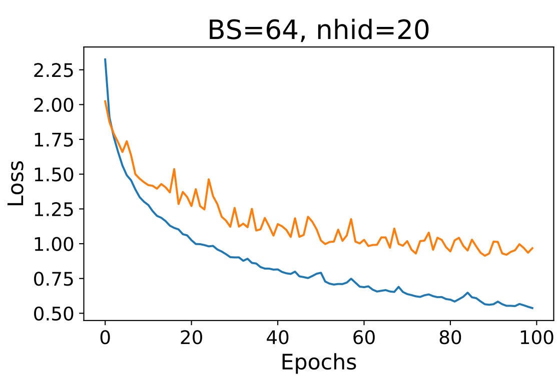

Figure 2: Training and validationl loss plot for the GRU.

3. Train the RNN encoder/decoder model. Try a few of your own words. Identify two distinct failure modes and briefly describe them.

Some English to PigLatin outputs of the trained RNN model are as follows:
English: 'the air conditioning is working'
PigLatin (output): 'etay airway onditingculay isway orkngray'
PigLatin (correct): 'ethay airway onditioningcay isway orkingway'

English: 'i went shopping'
PigLatin (output): 'iway entway opingscay'
PigLatin (correct): 'iway entway oppingshay'

As can be seen in the first example, the model does not seem to learn that the middle portion of the English word should not change. Furthermore, as can be seen in the second example, the model does not seem to learn that consonant pairs such as 'sh' should be moved together.

Part 2: Additive Attention [2pt]

1. Write down the mathematical expression for $\tilde{\alpha}_i^{(t)}$, $\alpha_i^{(t)}$, and c_t as a function of W_1, W_2, b_1, b_2, Q_t, K_i.

$$\tilde{\alpha}_i^{(t)} = f\left(Q_t, K_i\right) = W_2\,\mathrm{ReLu}(W_1(\mathrm{concat}(Q_t, K_i)) + b_1) + b_2$$

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

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

2. A screenshot of your RNNAttentionDecoder class implementation. [1pt]

```python
class RNNAttentionDecoder(nn.Module):
    def __init__(self, vocab_size, hidden_size, attention_type='scaled_dot'):
        super(RNNAttentionDecoder, self).__init__()
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(vocab_size, hidden_size)

        self.rnn = MyGRUCell(input_size=hidden_size*2, hidden_size=hidden_size)
        if attention_type == 'additive':
          self.attention = AdditiveAttention(hidden_size=hidden_size)
        elif attention_type == 'scaled_dot':
          self.attention = ScaledDotAttention(hidden_size=hidden_size)

        self.out = nn.Linear(hidden_size, vocab_size)


    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(h_prev, annotations, annotations)  # batch_size x 1 x hidden_size
            embed_and_context = torch.cat([np.squeeze(context, axis = 1), embed_current], dim = 1)  # batch_size x (2*hidden_size)
            h_prev = self.rnn(embed_and_context, h_prev)  # batch_size x hidden_size

            # ------------
            # FILL THIS IN - START
            # ------------

            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
```

Figure 3: A screenshot of my RNNAttentionDecoder class implementation.

Part 3: Scaled Dot Product Attention [4pt]

1. Screenshots of your ScaledDotProduct, CausalScaledDotProduct, TransformerEncoder and TransformerDecoder implementations. Highlight the lines you've added. [2pt]

```python
class ScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(ScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype= torch.float))

    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 (k) x hidden_size)
            keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
            values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

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

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

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = keys.size(0)
        q = self.Q(queries)
        if len(q.size()) == 2:
            q = q.view(batch_size, 1, hidden_size)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = torch.bmm(k, q.transpose(1,2))*self.scaling_factor
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(1,2), v)
        return context, attention_weights
```

Figure 4: A screenshot of my implementation of the scaled dot-product attention mechanism.

```python
class CausalScaledDotAttention(nn.Module):
    def __init__(self, hidden_size):
        super(CausalScaledDotAttention, self).__init__()

        self.hidden_size = hidden_size
        self.neg_inf = torch.tensor(-1e7)

        self.Q = nn.Linear(hidden_size, hidden_size)
        self.K = nn.Linear(hidden_size, hidden_size)
        self.V = nn.Linear(hidden_size, hidden_size)
        self.softmax = nn.Softmax(dim=1)
        self.scaling_factor = torch.rsqrt(torch.tensor(self.hidden_size, dtype= torch.float))

    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 (k) x hidden_size)
            keys: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)
            values: The encoder hidden states for each step of the input sequence. (batch_size x seq_len x hidden_size)

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

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

        # ------------
        # FILL THIS IN
        # ------------
        batch_size = keys.size(0)
        q = self.Q(queries)
        if len(q.size()) == 2:
            q = q.view(batch_size, 1, hidden_size)
        k = self.K(keys)
        v = self.V(values)
        unnormalized_attention = torch.bmm(k, q.transpose(1,2))*self.scaling_factor
        mask = torch.tril(torch.ones(unnormalized_attention.shape),diagonal=-1).cuda()
        unnormalized_attention = unnormalized_attention+mask*self.neg_inf
        attention_weights = self.softmax(unnormalized_attention)
        context = torch.bmm(attention_weights.transpose(1,2), v)
        return context, attention_weights
```

Figure 5: A screenshot of my implementation of the causal scaled dot-product attention mechanism.

```python
class TransformerEncoder(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers, opts):
        super(TransformerEncoder, self).__init__()

        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.opts = opts

        self.embedding = nn.Embedding(vocab_size, hidden_size)

        # IMPORTANT CORRECTION: NON-CAUSAL ATTENTION SHOULD HAVE BEEN
        # USED IN THE TRANSFORMER ENCODER.
        # NEW VERSION:
        self.self_attentions = nn.ModuleList([ScaledDotAttention(
                                    hidden_size=hidden_size,
                                 ) for i in range(self.num_layers)])
        # PREVIONS VERSION:
        # self.self_attentions = nn.ModuleList([CausalScaledDotAttention(
        #                             hidden_size=hidden_size,
        #                          ) for i in range(self.num_layers)])
        self.attention_mlps = nn.ModuleList([nn.Sequential(
                                    nn.Linear(hidden_size, hidden_size),
                                    nn.ReLU(),
                                 ) for i in range(self.num_layers)])

        self.positional_encodings = self.create_positional_encodings()

    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 seq_len)

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

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

        # Add positinal embeddings from self.create_positional_encodings. (a'la https://arxiv.org/pdf/1706.03762.pdf, section 3.5)
        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,encoded,encoded)  # batch_size x seq_len x hidden_size
          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
```

Figure 6: A screenshot of my implementation of TransformerEncoder.

```python
class TransformerDecoder(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers):
        super(TransformerDecoder, self).__init__()
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        self.num_layers = num_layers

        self.self_attentions = nn.ModuleList([CausalScaledDotAttention(
                                    hidden_size=hidden_size,
                                 ) for i in range(self.num_layers)])
        self.encoder_attentions = nn.ModuleList([ScaledDotAttention(
                                    hidden_size=hidden_size,
                                 ) for i in range(self.num_layers)])
        self.attention_mlps = nn.ModuleList([nn.Sequential(
                                    nn.Linear(hidden_size, hidden_size),
                                    nn.ReLU(),
                                 ) for i in range(self.num_layers)])
        self.out = nn.Linear(hidden_size, vocab_size)

        self.positional_encodings = self.create_positional_encodings()

    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 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

        # 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 seq_len x hidden_size
          residual_contexts = contexts + new_contexts
          new_contexts, encoder_attention_weights = self.encoder_attentions[i](residual_contexts, annotations, annotations) # batch_size x seq_len x hidden_size
          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)
        encoder_attention_weights = torch.stack(encoder_attention_weights_list)
        self_attention_weights = torch.stack(self_attention_weights_list)

        return output, (encoder_attention_weights, self_attention_weights)
```

Figure 7: A screenshot of my implementation of TransformerDecoder.

2. Training/validation plots you've generated. Your response to question 5. Your analysis should not exceed three sentences (excluding the failure cases you've identified). [1pt]

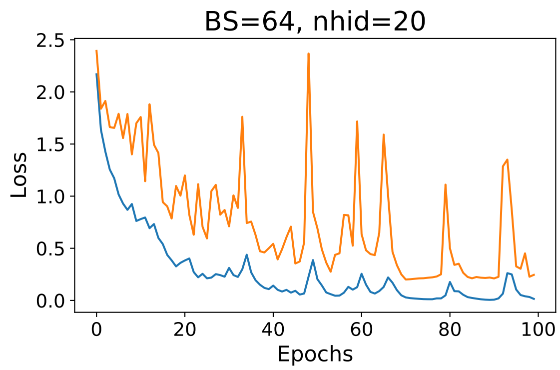

Figure 8: Training and validationl loss plot for the Transformer.

Some English to PigLatin outputs of the trained RNN model are as follows:
English: 'the air conditioning is working'
PigLatin (output): 'ethay airway onditioningcay isway orkingway'
PigLatin (correct): 'ethay airway onditioningcay isway orkingway'

English: 'i went shopping'
PigLatin (output): 'iway entway oppingshay'
PigLatin (correct): 'iway entway oppingshay'

The model with Transformer encoder and decoder is significantly better than before. The model translated both sentences perfectly and does not seem to have the failure modes identified before.

3. Your response to question 6. Your response should not exceed three sentences. [1pt]

The training loss went to 0 during training. The trained model always output a single character repeated many times no matter what the input was. This is to be expected because the model would simply learn an identity mapping when the decoder could see the next word (this way the reconstruction error is 0).

Part 4: BERT for arithmetic sentiment analysis [2pt]

1. 10 inference results in question 5 as well as brief comments on why they are interesting or representative results. Your answer should not exceed 3 sentences, you don't need to describe all 10 inference results [1pt]

Ten inference results are shown below:
1.
Input: 'twelve minus fourteen'
Output: 'negative'
2.
Input: 'twelve plus fourteen'
Output: 'positive'
3.
Input: '1 minus 14'
Output: 'negative'
4.
Input: '1 minus twelve'
Output: 'negative'
5.
Input: '1 plus twelve'
Output: 'positive'
6.
Input: '1 minus 1'

Output: 'positive'
7.
Input: 'five plus zero'
Output: 'positive'
8.
Input: 'one plus one plus one'
Output: 'positive'
9.
Input: 'one minus one minus one'
Output: 'negative'
10.
Input: 'one plus one minus one'
Output: 'positive'

Example #4 is interesting because it involves a number and a word that represents a number.
Example #10 is interesting because it involves one operation followed by another.
Example #6 is interesting because it suggests that the trained model did not learn zero.

2. Explanation of what you did for the open question and some preliminary results. Your answer should not exceed 4 sentences. [1pt]

I chose hyperparameter tuning with a grid search for this part. The code is as shown below.

```
learning_rates = [1e-5, 2e-5, 3e-5, 4e-5]
num_epochs = [5, 6, 7, 8, 9, 10]

best_hyper = [None, None]
best_accuracy = 0
for learning_rate in learning_rates:
  for num_epoch in num_epochs:
    finttune_bert_loss_vals = train_model(model_finetune_bert, num_epoch, learning_rate)
    loss_values, val_accuracy = finttune_bert_loss_vals
    if val_accuracy > best_accuracy:
      best_hyper = [learning_rate, num_epoch]
      best_accuracy = val_accuracy
      print('best hyperparameter:', best_hyper)
      print('best validation accuracy:', best_accuracy)
```

Figure 9: Grid search for the best learning rate and epoches combination.

The best combination was a learning rate of 1e-5 and number of epoches of 7 which lead to a validation accuracy of 1.