CSC413: Programming Assignment 3: Attention-Based Neural Machine Translation

Anzhe Dong, 1002608163

March 2021

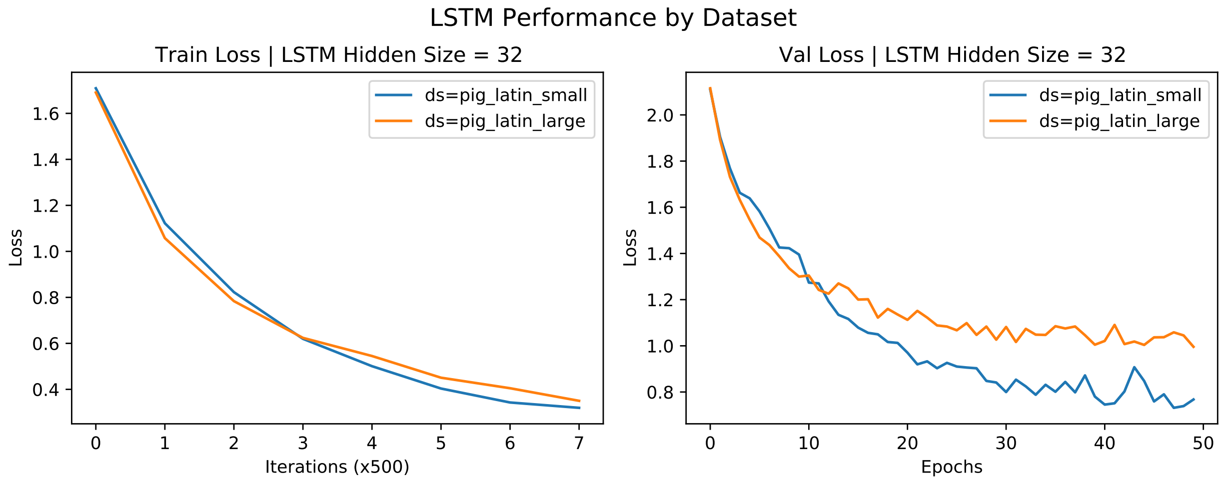
Part 1: LSTMs

1. LSTM Training

A screenshot of your `fullMyLSTMCell` implementation

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

        self.input_size = input_size
        self.hidden_size = hidden_size

        # ------------
        # FILL THIS IN
        # ------------
        self.Wii = nn.Linear(input_size, hidden_size)
        self.Whi = nn.Linear(hidden_size, hidden_size)

        self.Wif = nn.Linear(input_size, hidden_size)
        self.Whf = nn.Linear(hidden_size, hidden_size)

        self.Wig = nn.Linear(input_size, hidden_size)
        self.Whg = nn.Linear(hidden_size, hidden_size)

        self.Wio = nn.Linear(input_size, hidden_size)
        self.Who = nn.Linear(hidden_size, hidden_size)


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

        Arguments
            x: batch_size x input_size
            h_prev: batch_size x hidden_size
            c_prev: batch_size x hidden_size

        Returns:
            h_new: batch_size x hidden_size
            c_new: batch_size x hidden_size
        """

        # ------------
        # FILL THIS IN
        # ------------
        i = torch.sigmoid(self.Wii(x) + self.Whi(h_prev))
        f = torch.sigmoid(self.Wif(x) + self.Whf(h_prev))
        g = torch.tanh(self.Wig(x) + self.Whg(h_prev))
        o = torch.sigmoid(self.Wio(x) + self.Who(h_prev))
        c_new = f * c_prev + i * g
        h_new = o * torch.tanh(c_new)
        return h_new, c_new
```

... the loss plots output by `save_loss_comparison_lstm`



... and your analysis

Does either model perform significantly better? Why might this be the case?

The `pig_latin_small` model performed better with lower validation loss. The small model was trained with smaller batch sizes (`'batch_size':64`), which tend to have better generalization ability than ones trained with larger batch sizes (`'batch_size':512`). [Reference: Keskar NS, Mudigere D, Nocedal J, Smelyanskiy M, Tang PT. On large-batch training for deep learning: Generalization gap and sharp minima. arXiv preprint arXiv:1609.04836. 2016 Sep 15.]

Model Failure

Identify a distinct failure mode and briefly describe it.

```
source:    the air conditioning is working
translated: ethay airway oniningbay-intway isway orkingway

source:    this has a lot of errors
translated: isthay ashay away otlay ofay errorsway

source:    the princess listened peacefully to what the frogs had to sing
translated: ethay insecspray istenedlay eaffulecay otay athay ethay ogsgay adhay otay ingsay
```

The model fails apparently at (1) when the leading character is an vowel, (2) when the leading consonant pairs such as "sh" and "wh", (3) long words.

Model size

Write down the number of neurons and connections of this encoder model as a function of H, K, and D. For simplicity, you may ignore the bias units.

Number of neurons:
Number of connections:

Part 2: Additive Attention

1

$$\tilde{\alpha}_i^{(t)} = W_2^\intercal \mathrm{ReLU}(W_1^\intercal \begin{bmatrix} Q_t \\ K_i \end{bmatrix})$$

$$\alpha_i^{(t)} = \mathrm{softmax}(\tilde{\alpha}^{(t)})_i = \frac{\exp\left(\tilde{\alpha}_i^{(t)}\right)}{\sum_{j=1}^{\text{seq\_len}} \exp\left(\tilde{\alpha}_j^{(t)}\right)}$$

$$c_t = \alpha^{(t)\intercal} K = \sum_{i=1}^{\text{seq\_len}} \alpha_i^{(t)} K_i$$

2

3

The training speed is faster with the attention model which reached less than 1.0 validation loss within 7 epochs, whereas previously it took the "small" non-attention model 20 epochs to reach this level of validation loss.

4

Part 3: Scaled Dot Product Attention

1. Implement the scaled dot-product attention mechanism.

A screenshot of your `ScaledDotAttention` implementation

```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 1)

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

        # ------------
        # FILL THIS IN
        # ------------
        batch_size, _i, _j = queries.size()
        q = self.Q(queries).view(batch_size, -1, self.hidden_size)
        k = self.K(keys).view(batch_size, -1, self.hidden_size)
        v = self.V(values).view(batch_size, -1, self.hidden_size)
        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
```

2. Implement the causal scaled dot-product attention mechanism.

A screenshot of your `CausalScaledDotAttention` implementation

```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 1)

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

```
        """

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

3.

4.

5.