CS 533: Natural Language Processing **(Due: 03/10/20)**

Assignment 3

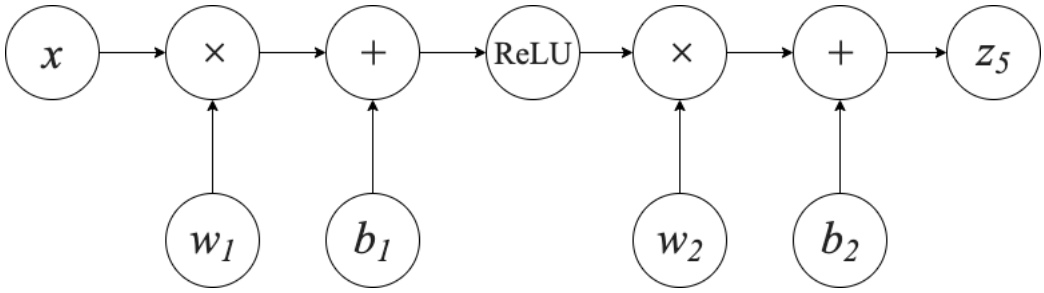
Name: Madhu Sivaraj, *NetID:* ms2407

Problem 1: Backpropagation

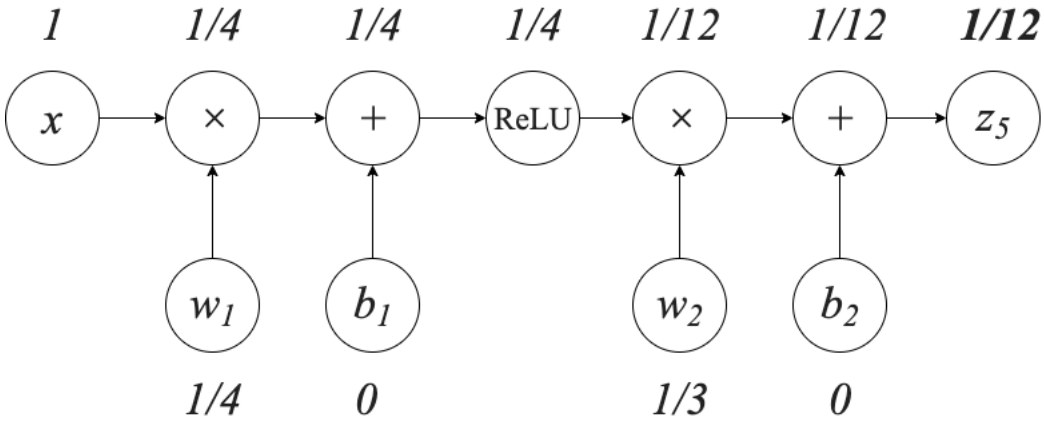
((1+1+3+1+1)+(1+1+4)=13 points)

1. Scalar-Valued Variables

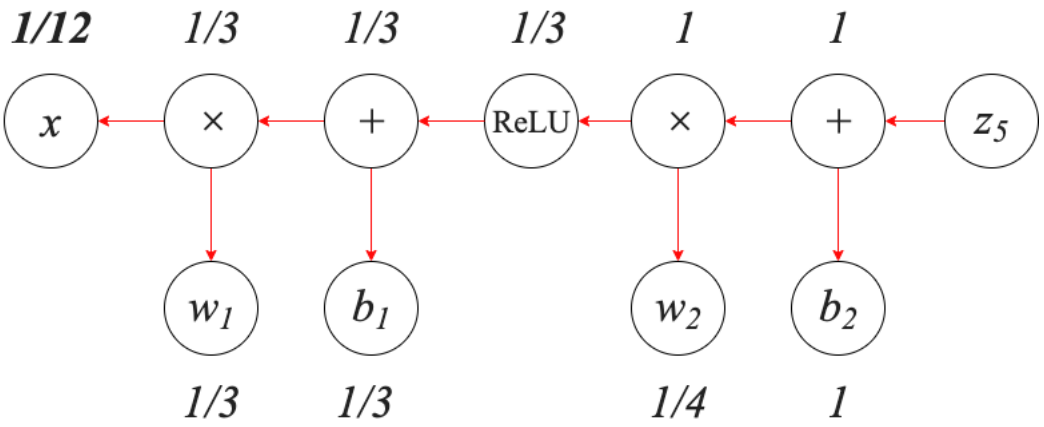
   (a) Below is the corresponding computation graph with x,w_1,b_1,w_2,b_2 as the input nodes, z_5 as the output node, and ×/+/ReLU as internal node types (corresponding to multiplication, addition, and ReLU).



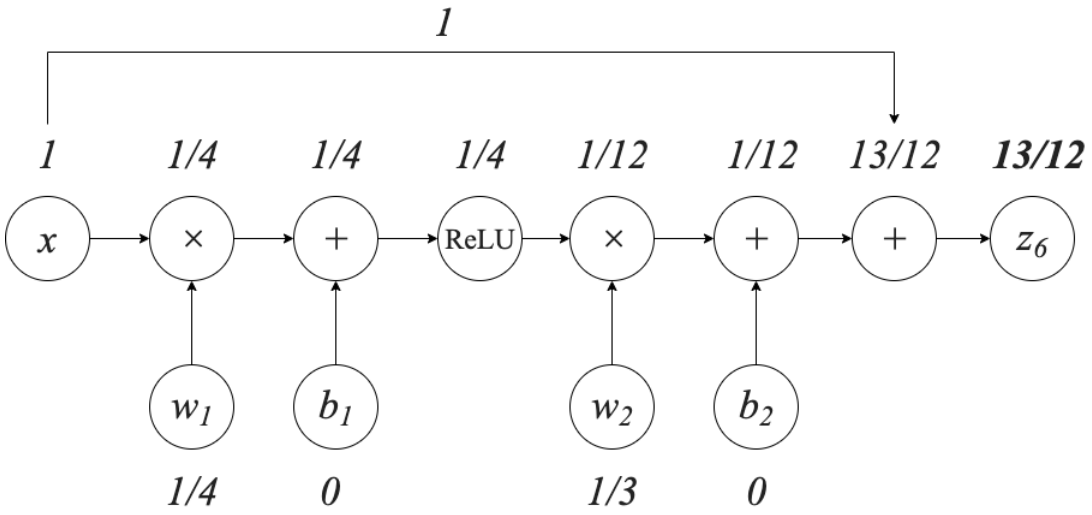
   (b) By running the forward pass on the computation graph, I evaluated $z_5 = \frac{1}{12}$.



(c) By running backpropagation, I calculated the gradient of z_5 with respect to x, w_1, b_1, w_2, and b_2.



(d) Below is a resulting computation graph with z_6 as the output node, where $z_6 = z_5 + x$. I ran the forward pass to evaluate z_6 and backpropagation to calculate the gradient of z_6 with respect to x, w_1, b_1, w_2, and b_2 as input values.



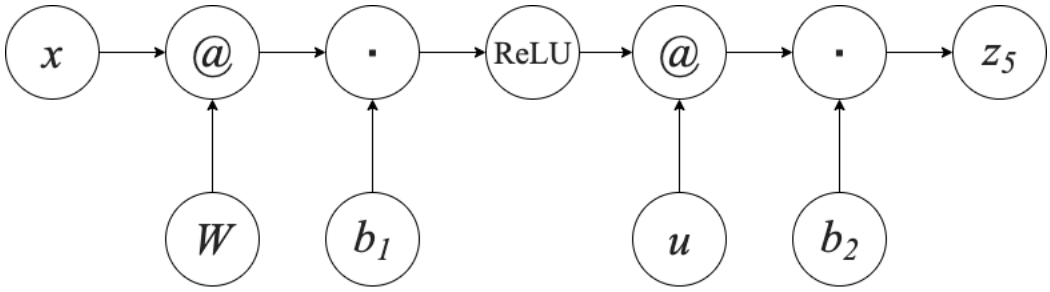
Forward pass: $z_6 = z_5 + x = \frac{13}{12}$

Backpropagation: $\frac{\partial z_5}{\partial z_6} = 1$; $\frac{\partial x}{\partial z_6} = \frac{13}{12}$

(e) Prior to the skip connection, the function is more sensitive to the weights and biases in the network, suggested from the gradient. After adding the skip connection, the gradient increases, and thus the function is more sensitive to x.

2. Vector-Valued Variables

(a) Below is the corresponding computation graph with X, w, b_1, u, and b_2 as the input nodes, z_5 as the output node, and @/+/ReLU/$\cdot$ as internal node types (corresponding to matrix multiplication, vector addition, elementwise ReLU, and dot product).



(b) By running the forward pass on the graph, I evaluated $z_5 = 2$.

(c) Below I run backpropagation to calculate the gradient of z_5 with respect to b_2, u, b_1, W, and x evaluated at the input values.

$$\frac{\partial z_5}{\partial b_2} = 1$$

$$\frac{\partial z_5}{\partial u} = \begin{bmatrix} 0 \\ 4 \end{bmatrix}$$

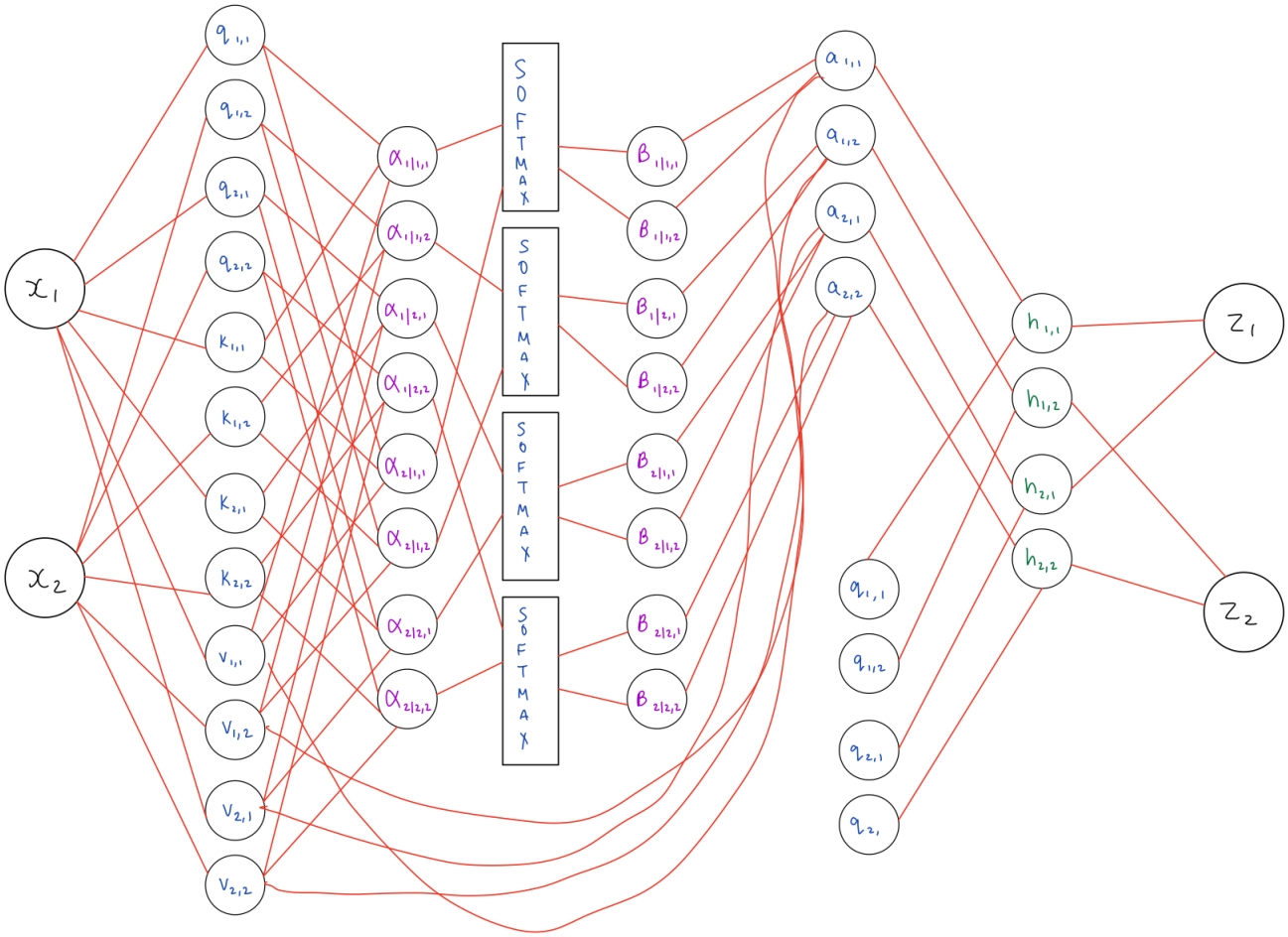
$$\frac{\partial z_5}{\partial b_1} = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

$$\frac{\partial z_5}{\partial W} = \begin{bmatrix} 0 & 0 \\ 2 & 2 \end{bmatrix}$$

$$\frac{\partial z_5}{\partial x} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

Problem 2: Self-Attention (4 + 4 = 8 points)

1. See graph below.



2. After running the forward pass with the following input nodes (d = 2):

$$z_1 = \begin{bmatrix} 0.8 \\ -0.8 \end{bmatrix}$$

$$z_2 = \begin{bmatrix} 2.8 \\ 3 \end{bmatrix}$$

Problem 3: Programming (1 + 2 + (1 + 1 + 1) + (1 + 1 + 1) + 5 + 4 + 1 + 1 + 3 = 23 points)

Code is attached in the submission zip file.

1. I implemented the get_ngram_counts function. Code can be found in bleu.py.

```
def get_ngram_counts(refs, hyp, n):
    # ref_ngrams = [tuple(r[i:i + n]) for r in r for i in range(len(r) - n + 1)]
    ref_ngrams = []
    for r in refs:
        ref_ngrams.append([tuple(r[i:i + n]) for i in range(len(r) - n + 1)])
    hyp_ngrams = [tuple(hyp[i:i + n]) for i in range(len(hyp) - n + 1)]
    num_hyp_ngrams = max(1, len(hyp_ngrams)) # Avoid empty
    num_hyp_ngrams_in_refs_clipped = 0 # TODO: Implement
    count = Counter(hyp_ngrams)
    for i in list(set(hyp_ngrams)):
```

```
            ceiling = math.inf
            for j in ref_ngrams:
                ceiling = max(ceiling, j.count(i))
            num_hyp_ngrams_in_refs_clipped += min(count[i], ceiling)
        return num_hyp_ngrams_in_refs_clipped, num_hyp_ngrams
```

2. I implemented the compute_bleu function. Code can be found in bleu.py.

```
def compute_bleu(reflists, hyps, n_max=4, use_shortest_ref=False):
    assert len(reflists) == len(hyps)
    prec_mean = 0 # TODO: Implement
    p = 1
    for n in range(n_max):
        r, h = 0,0
        for i in range(len(reflists)):
            num_hyp_ngrams_in_refs_clipped, num_hyp_ngrams = get_ngram_counts(reflists[i
                ↪ ], hyps[i], n+1)
            r += num_hyp_ngrams_in_refs_clipped
            h += num_hyp_ngrams
        p *= float(r/h)
    if p > 0:
        prec_mean = math.exp((1/n_max)*math.log(p))
    brevity_penalty, rl, hl = 0,0,0 # TODO: Implement
    for i in range(len(reflists)):
        ceiling = math.inf
        lr = 0
        for ref in reflists[i]:
            if ceiling > (abs(len(ref)-len(hyps[i]))):
                lr = len(ref)
        rl += lr
        hl += len(hyps[i])
    if float(hl/rl) <= 1:
        brevity_penalty = math.exp(1-1/float(hl/rl))
    else:
        brevity_penalty = 1
    # TODO: Implement
    bleu = brevity_penalty * prec_mean
    return bleu
```

3. (1) Backpropagation through time also known as bptt is an application of backprogation. It is often used in recurrent neural networks which expands the RNN in increments of time to obtain the dependencies between model variables and parameters. Then, backpropagation is applied to compute and store gradients. Exploding or vanishing gradients can occur in long seq2seq models so an alternative is a truncated version of bptt, which backpropogates through a certain number of layers.

   (2) The model computes the loss and calculate gradients in the step_on_batch function in control.py.

```
def step_on_batch(self, subblock, golds, src=None, lengths=None,start=True):
        self.s2s.zero_grad()
        output, attn = self.s2s(subblock, src=src, lengths=lengths,start=start)
        loss = self.avgCE(output, golds)
        loss.backward()

        nn.utils.clip_grad_norm_(self.s2s.parameters(), 0.25)
        for p in self.s2s.parameters():
            p.data.add_(-self.lr, p.grad.data)

        return loss.item()
```

(3) The model "carries over" the hidden state from the previous batch in the by calling the self.dec.detach_state function. In this function, it sets the hidden state from the previous batch as its current.

4. (1) In this case, bptt is no longer used because for the translation model bundles up sequences of input and translation sentences, so the sentences are of a reasonable length. Thus there is no need for a truncated version of backpropagation through time.

   (2) The model encodes the source sentence in the forward function in encoder.py.

```
def forward(self, src, lengths):
   packed_emb = pack(self.embeddings(src), lengths)
   memory_bank, encoder_final = self.lstm(packed_emb)
   memory_bank = unpack(memory_bank)[0]

   if self.use_bridge:
       encoder_final = self._bridge(encoder_final)

   # T x B x d L x B x d (2L x B x d/2 if bidir)
   return memory_bank, encoder_final
```

   (3) The model conditions on the final encoding of the source sentence in the init_state function. See below.

```
def init_state(self, batch_size=None, encoder_final=None):
    if encoder_final:
        def _fix_enc_hidden(hidden):
            # The encoder hidden is (layers*directions) x batch x dim.
            # We need to convert it to layers x batch x (directions*dim).
            if self.bidirectional_encoder:
                hidden = torch.cat([hidden[0:hidden.size(0):2],
                                    hidden[1:hidden.size(0):2]], 2)
            return hidden

        self.state['hidden'] = tuple([_fix_enc_hidden(enc_hid)
                                      for enc_hid in encoder_final])

        # Init the input feed.
        batch_size = self.state['hidden'][0].size(1)
        h_size = (batch_size, self.dim)
        self.state['input_feed'] = \
            self.state['hidden'][0].data.new(*h_size).zero_().unsqueeze(0)

    else:
        assert batch_size

        # Following PT ex.: get whatever Parameter object to create hidden
        weight = next(self.parameters())
        self.state['hidden'] = tuple([weight.new_zeros(self.num_layers,
                                                       batch_size, self.dim)
                                      for _ in range(2)])
        self.state['input_feed'] = None
```

5. See attention.py.

6. See attention.py.

7. See attention.py.

8. Final Perplexity: 196.77

```
-----------------------------------------------------------------------------------------
| end of epoch   1 | time:  1.34s | valid loss  6.10 | valid ppl   444.28 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   2 | time:  1.22s | valid loss  5.76 | valid ppl   317.98 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   3 | time:  1.17s | valid loss  5.79 | valid ppl   326.14 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   4 | time:  1.83s | valid loss  5.51 | valid ppl   247.36 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   5 | time:  1.18s | valid loss  5.49 | valid ppl   242.97 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   6 | time:  1.29s | valid loss  5.47 | valid ppl   238.34 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   7 | time:  1.82s | valid loss  5.44 | valid ppl   230.28 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   8 | time:  1.19s | valid loss  5.38 | valid ppl   217.39 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch   9 | time:  1.39s | valid loss  5.32 | valid ppl   204.00 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------
| end of epoch  10 | time:  1.32s | valid loss  5.28 | valid ppl   196.77 | valid sqxent    -1.00
-----------------------------------------------------------------------------------------
=========================================================================================
| End of training | final loss  5.28 | final ppl   196.77 | final sqxent    -1.00
=========================================================================================
```

9. When training the attention model to convergence, the perplexity gets does not decrease as much as I expected.