Problem 1: Backpropagation

1. (Scalar-Valued Variables): Let \( x, w_1, b_1, w_2, b_2 \in \mathbb{R} \) and define
   \[
   \begin{align*}
   z_1 &= w_1 x \\
   z_2 &= z_1 + b_1 \\
   z_3 &= \text{ReLU}(z_2) = \max\{0, z_2\} \\
   z_4 &= w_2 z_3 \\
   z_5 &= z_4 + b_2
   \end{align*}
   \]
   Thus \( z_5 = w_2 \text{ReLU}(w_1 x + b_1) + b_2 \in \mathbb{R} \) is the output of a feedforward network with one nonlinear (ReLU) layer in which all variables are scalars.

   (a) Draw 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 \( \times/+/\text{ReLU} \) as internal node types (corresponding to multiplication, addition, and ReLU).

   (b) Evaluate \( z_5 \) with input values \( x = 1, w_1 = 1/4, b_1 = 0, w_2 = 1/3, \) and \( b_2 = 0 \) by running the forward pass on the graph.

   (c) Run backpropagation to calculate the gradient of \( z_5 \) with respect to \( b_2, w_2, b_1, w_1, x \) evaluated at the input values.

   (d) Add the skip connection \( z_6 = z_5 + x \) and draw the resulting computation graph (with \( z_6 \) as the output node). Run the forward pass to evaluate \( z_6 \) and backpropagation to calculate the gradient of \( z_6 \) with respect to \( b_2, w_2, b_1, w_1, x \) evaluated at the same input values. You only need to do two additional computations (one in forward, one in backward) on top of what you have already done in the previous problem.

   (e) What does the gradient with respect to \( x \) say about the sensitivity of the function with \( x \) before (\( z_5 \)) and after (\( z_6 \)) adding the skip connection?

2. (Vector-Valued Variables): Let \( x, b_1, u \in \mathbb{R}^2, W \in \mathbb{R}^{2 \times 2}, \) and \( b_2 \in \mathbb{R}; \) define
   \[
   \begin{align*}
   z_1 &= W x \\
   z_2 &= z_1 + b_1 \\
   z_3 &= \text{ReLU}(z_2) \\
   z_4 &= u^\top z_3 \\
   z_5 &= z_4 + b_2
   \end{align*}
   \]
   Thus \( z_5 = u^\top \text{ReLU}(W x + b_1) + b_2 \in \mathbb{R}. \)
(a) Draw the corresponding computation graph with \( x, W, b_1, u, b_2 \) as the input nodes, \( z_5 \) as the output node, and \( \oplus / + / \text{ReLU} / \cdot \) as internal node types (corresponding to matrix multiplication, vector addition, elementwise ReLU, and dot product).

(b) Evaluate \( z_5 \) with input values
\[
x = \begin{bmatrix} 1 \\
1
\end{bmatrix} \quad W = \begin{bmatrix}
-1 & -1 \\
1 & 1
\end{bmatrix} \quad b_1 = \begin{bmatrix} 0 \\
0
\end{bmatrix} \quad u = \begin{bmatrix} 1 \\
1
\end{bmatrix} \quad b_2 = 0
\]
by running the forward pass on the graph.

(c) Run backpropagation to calculate the gradient of \( z_5 \) with respect to \( b_2, u, b_1, W, x \) evaluated at the input values. You may use Lemma 1 below.

**Lemma 1** (Gradients of the Matrix Product). Let \( W = UV \) where \( W \in \mathbb{R}^{n \times p}, U \in \mathbb{R}^{n \times m}, \) and \( V \in \mathbb{R}^{m \times p} \). Let \( z \in \mathbb{R} \) be a differentiable function of \( W \). Assume \( z \) is a function of \( U \) and \( V \) only though \( W \). Let \( \nabla W \in \mathbb{R}^{n \times p} \) denote a matrix of partial derivatives \( \nabla_{i,j} = \partial z / \partial W_{i,j} \), similarly for \( U \in \mathbb{R}^{n \times m} \) and \( V \in \mathbb{R}^{m \times p} \). Then \( \nabla = \nabla V^T \) and \( \nabla = U^T \nabla U \).

**Proof.**
\[
\nabla_{i,j} = \frac{\partial z}{\partial U_{i,j}} = \sum_{k,l} \frac{\partial z}{\partial W_{k,l}} \frac{\partial W_{k,l}}{\partial U_{i,j}} = \sum_{k,l} \nabla_{k,l} \begin{cases} V_{j,l} & \text{if } k = i \\ 0 & \text{otherwise} \end{cases} = \sum_l \nabla_{i,l} V_{j,l}
\]

\[\Box\]

**Problem 2: Self-Attention**

You will manually run the forward pass on a simplified Transformer encoder layer. This does not include important details such as layer normalization and dropout. For full details check out the implementation The Annotated Transformer.

The layer has \( H \) heads with parameters \( W_h^Q, W_h^K, W_h^V \in \mathbb{R}^{d/H \times d} \) for \( h = 1 \ldots H \) and \( W \in \mathbb{R}^{d \times d} \). It receives a sequence of input vectors \( x_1 \ldots x_T \in \mathbb{R}^d \) and performs the following calculation for \( h = 1 \ldots H \):

\[
q_{h,t} = W_h^Q x_t \quad \forall t = 1 \ldots T
\]
\[
k_{h,t} = W_h^K x_t \quad \forall t = 1 \ldots T
\]
\[
v_{h,t} = W_h^V x_t \quad \forall t = 1 \ldots T
\]
\[
\alpha_{t'|h,t} = h_{t',h}^T q_{h,t} \quad \forall t = 1 \ldots T, t' = 1 \ldots T
\]
\[
(\beta_{1|h,t} \ldots \beta_{T|h,t}) = \text{softmax} (\alpha_{1|h,t} \ldots \alpha_T|h,t) \quad \forall t = 1 \ldots T
\]
\[
a_{h,t} = \sum_{t'=1}^T \beta_{t'|h,t} v_{h,t'} \quad \forall t = 1 \ldots T
\]
\[
h_{h,t} = a_{h,t} + q_{h,t} \quad \forall t = 1 \ldots T
\]

We define the final representation corresponding to \( x_t \in \mathbb{R}^d \) to be
\[
z_t = W(h_{1,t} \oplus \ldots \oplus h_{H,t}) \in \mathbb{R}^d
\]
where \( \oplus \) mean the vector concatenation operation.

1. Let \( H = 2 \) and \( T = 2 \). Draw the underlying computation graph. Be as simple as possible, do not overly complicate the picture. You may simplify things a bit by drawing connections directly between nodes and hide the underlying functions (e.g., \( x_t \to q_{h,t} \)). Be careful with the softmax function: it intertwines all \( \beta_{1|h,t} \ldots \beta_{T|h,t} \) with all \( \alpha_{1|h,t} \ldots \alpha_T|h,t \).

\[\footnote{This assumption is merely for simplicity. As usual in a computation graph, if \( U \) is used multiple times we can accrue the “local” gradients to obtain the “global” gradient. For instance, if \( A = UV \) and \( z \) is a function of \( U \) through both \( W \) and \( A \), then \( \nabla = \nabla V^T + A B^T \).}\]
2. Run the forward pass with the following input nodes ($d = 2$):

\[
\begin{align*}
  x_1 &= \begin{bmatrix} 1 \\ 2 \end{bmatrix}, & x_2 &= \begin{bmatrix} -1 \\ 3 \end{bmatrix}, & W_1^Q &= \begin{bmatrix} 1 & 0 \end{bmatrix}, & W_1^K &= \begin{bmatrix} 1 & 0 \end{bmatrix}, & W_1^V &= \begin{bmatrix} 1 & 0 \end{bmatrix}, & W &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \\
  W_2^Q &= \begin{bmatrix} 0 & 1 \end{bmatrix}, & W_2^K &= \begin{bmatrix} 0 & 1 \end{bmatrix}, & W_2^V &= \begin{bmatrix} 0 & 1 \end{bmatrix}.
\end{align*}
\]

### Problem 3: Programming

(1 + 2 + (1 + 1 + 1) + (1 + 1 + 1) + 5 + 4 + 1 + 1 + 3 = 23 points)

As usual, download the starter kit provided and follow the instructions below.

#### BLEU. Pass the tests in `test_bleu.py` by finishing the implementation of `get_ngram_counts` and `compute_bleu` in `bleu.py` (you can just type `python test_bleu.py` to run the tests). We follow the standard practice described here.

1. Implement `get_ngram_counts`. It receives a list of reference sentences $r^{(1)} \ldots r^{(M)}$, a hypothesis sentence $h$, and an integer $n$. Let $\text{NGRAMS}(h, n)$ denote the list of all $n$-grams in $h$. The function outputs a pair of integers $a_n, b_n$ such that $b_n = \max \{1,|\text{NGRAMS}(h, n)|\}$ (already implemented) and

\[
a_n = \sum_{(g, c) \in \text{Counter}(\text{NGRAMS}(h, n))} \min \left\{ c, \max_{j=1}^{M} \#(g|r^{(j)}) \right\}
\]

where $(g, c)$ denotes an $n$-gram type $g$ that appears $c$ times in $h$, and $\#(g|r^{(j)})$ is the number of times $g$ appears in reference $r^{(j)}$.

2. Implement `compute_bleu`. It receives $(r^{(l,1)} \ldots r^{(l,M_l)}, h^{(l)})$ for $l = 1 \ldots N$ and computes

\[
\text{BLEU} = \min \left\{ 1, \exp \left( 1 - \frac{R}{H} \right) \right\} \times \left( \prod_{n=1}^{4} p_{n} \right)^{1/4}
\]

where $R = \sum_{l=1}^{N} |r^{(l,j^*)}|$ with $j^* = \arg \min_{j=1}^{M_l} ||r^{(l,j)}|| - ||h^{(l)}||$, $H = \sum_{l=1}^{N} |h^{(l)}|$, and $p_{n} = \sum_{l=1}^{N} a_{l}^{(l)} / \sum_{l=1}^{N} b_{l}^{(l)}$.

#### Input-feeding attention. For this part, we again train and test on the same data (!) for computational reasons. Keep in mind that this is something you will never do in practice; reporting performance on the training set is meaningless (i.e., for generalization purposes) except for checking the model’s ability to “fit data”, which is what we do here.

3. Type `python main.py --train` to train an LSTM language model on `data/train.txt` and `data/valid.txt`. The default command should yield perplexity 179.27 (or similar) after 10 epochs. Study how this is done by tracking computation. You will see that the training corpus is now organized as one matrix of parallel sequences $(T \times B)$ where $B$ is the batch size). Explain: (1) What does `bptt` (stands for “backpropagation through time”) do? (2) In which function does the model compute the loss and calculate gradients? (3) How does the model “carry over” the hidden state from the previous batch?

4. Type `python main.py --train --cond --batch_method translation` to train an LSTM language model on `data/train.txt` and `data/valid.txt` where for each sentence the model condition on the last hidden state of the source sentence in `data/src-train.txt` and `data/src-valid.txt`. These source sentences are actually the very same sentences the model is predicting! So we obviously expect the perplexity should go down. The default command should yield perplexity 145.86 (or similar) after 10 epochs. Again, study how this is done by tracking computation. You will see that the training corpus is now organized as bundles of source-target sequences with padding. Explain: (1) In this case `bptt` is no longer used. Why? (2) In which function does the model encode the source sentence? (3) In which function does the model condition on the final encoding of the source sentence?
5. Implement score in `attention.py`. It receives (batched) query vectors \( Q \in \mathbb{R}^{B \times T' \times d} \) and key vectors \( K \in \mathbb{R}^{B \times T \times d} \) and computes score matrix \( L \in \mathbb{R}^{B \times T' \times T} \) where

\[
L[i,j] = Q[i,:].T W K[j,:,;]
\]

\( \forall l = 1 \ldots B \)

where \( W \in \mathbb{R}^{d \times d} \) is a parameter of the model. Multiplying by \( W \) corresponds to feeding \( Q \) to the linear layer already in the model (``self.linear_in``), so do that. You should not use any for loops. You can do this using `torch.bmm` (batched matrix multiplication, look it up). You will want to change/switch dimensions of the input tensors efficiently by using `view` and `transpose`.

6. Implement Eq. (5) of Luong et al. (2015) in `forward` in `attention.py`. The \( c_t \) is already given (assuming score). The \( h_t \) is just queries. The multiplication by \( W_s \) should be done by feeding \( (c_t, h_t) \) to `self.linear_out` already in the model. You will want to use `torch.cat`, `view`, `torch.tanh` to align dimensions correctly.

7. Pass the simple tests in `test_attention.py` to check if you did the previous two problems correctly.

8. Type `python main.py --train --cond --batch_method translation --attn` to train an LSTM language model on `data/train.txt` and `data/valid.txt` where for each sentence the model conditions on the source sentence in `data/src-train.txt` and `data/src-valid.txt` using input-feeding attention. The default command should yield perplexity 137.25 (or similar) after 10 epochs. Attach a screenshot of the training session.

9. Train the attention model to convergence (e.g., set epochs to 1000) to see how small the perplexity can get. Examine the attention weights of the decoder. Do they correspond to what you’d expect? If so, give one illustrative example (e.g., \( [a/0.1, b/0.5, c/0.4], [a, b, c], b \) with the meaning that the attention weights at \( b \) (we want to predict \( c \)) has a distribution \((0.1, 0.5, 0.4)\) over \( (a, b, c) \)).