DS-GA-1011: Natural Language Processing with Representation Learning, Fall 2024

HW2 - Machine Translation

Aditeya Baral
N19186654

Please write down any collaborators, AI tools (ChatGPT, Copliot, codex, etc.), and external resources you used for this assignment here.
Collaborators:
AI tools:
Resources:

By turning in this assignment, I agree by the honor code of the College of Arts and Science at New York University and declare that all of this is my own work.

Acknowledgement: Problem 1 was developed by Yilun Kuang. Problem 2 is based off of Annotated Transformers from Sasha Rash and developed by Nitish Joshi.

Before you get started, please read the Submission section thoroughly.

Submission

Submission is done on Gradescope.

Written: You can either directly type your solution in the released `.tex` file, or write your solution using pen or stylus. A `.pdf` file must be submitted.

Programming: Questions marked with "coding" at the start of the question require a coding part. Each question contains details of which functions you need to modify. We have also provided some unit tests for you to test your code. You should submit all `.py` files which you need to modify, along with the generated output files as mentioned in some questions.

Compute Budget: For question 2.3, you should expect the total code execution time to be less than 2 hours on a single NVIDIA Quadro RTX 8000 GPU from NYU Greene HPC. Please plan ahead, as requesting GPU resources on the cluster can take several hours or even longer during peak times.

Due Date: This homework is due on October 9, 2024, at noon 12pm Eastern Time.

1 Recurrent Neural Network

In this problem, you will show the problem of vanishing and exploding gradients for Recurrent Neural Network (RNN) analytically. To show this, we will first expand the gradient of the loss function with respect to the parameters using the chain rule. Then, we will bound the norm of each individual partial derivative with matrix norm inequalities. The last step will be to collect all of the partial derivative terms and show how repeated multiplication of a single weight matrix can lead to vanishing or exploding gradients.

1.1 RNN Derivatives

Let $S = (s_1, \cdots, s_T)$ be a sequence of input word tokens and T be the sequence length. For a particular token $s_t \in \mathcal{V}$ for $1 \leq t \leq T$, we can obtain its corresponding word embedding $x_t \in \mathbb{R}^d$ by applying equation (1), where $\phi_{\text{one-hot}}$ is the one-hot encoding function and W_e is the word embedding matrix.

The RNN forward pass computes the hidden state $h_t \in \mathbb{R}^{d'}$ using equation (2). Here $W_{\text{hh}} \in \mathbb{R}^{d' \times d'}$ is the recurrent weight matrix, $W_{\text{ih}} \in \mathbb{R}^{d' \times d}$ is the input-to-hidden weight matrix, $b_h \in \mathbb{R}^{d'}$ is the hidden states bias vector, and $\sigma : \mathbb{R}^{d'} \to [-1, 1]^{d'}$ is the tanh activation function. $W_{\text{hh}}, W_{\text{ih}}, b_h$ are shared across sequence index t.

The output of RNN $o_t \in \mathbb{R}^k$ at each sequence index t is given by equation (3), where $W_{h_o} \in \mathbb{R}^{k \times d'}$ is the hidden-to-output weight matrix and $b_o \in \mathbb{R}^k$ is the output bias vector. For an input sequence $S = (s_1, \cdots, s_T)$, we have a corresponding sequence of RNN hidden states $H = (h_1, \cdots, h_T)$ and outputs $O = (o_1, \cdots, o_T)$.

$$x_t = W_e \phi_{\text{one-hot}}(s_t) \tag{1}$$
$$h_t = \sigma(W_{\text{hh}} h_{t-1} + W_{\text{ih}} x_t + b_h) \tag{2}$$
$$o_t = W_{h_o} h_t + b_o \tag{3}$$

Let's now use this RNN model for classification. In particular, we consider the last output o_T to be the logits (scores for each class), which we then convert to the class probability vector $p_T \in [0, 1]^k$ by $p_T = g(W_{h_o} h_T + b_o)$ where $g(\cdot)$ is the softmax function and $\|p_T\|_1 = 1$.

1. (1 point, written) Write down the per-example cross-entropy loss $\ell(y, p_T)$ for the classification task. Here $y \in \{0, 1\}^k$ is a one-hot vector of the label and p_T is the class probability vector where $p_T[i] = p(y[i] = 1 \mid S)$ for $i = 1, \ldots, k$. ($[i]$ denotes the i-th entry of the corresponding vector.)

We can express the per-example cross-entropy loss $\ell(y, p_T)$ as,

$$\ell(y, p_T) = -\sum_{i}^{k} y[i] \cdot \log(p_T[i]) \tag{4}$$

But since y is one-hot, only the true class label's term will contribute to the loss. This simplifies the equation to,

$$\ell(y, p_T) = -\log(p_T[i_{true}]) \tag{5}$$

2. To perform backpropagation, we need to compute the derivative of the loss with respect to each parameter. Without loss of generality, let's consider the derivative with respect to a single parameter $w = W_{\text{hh}}[i, j]$ where $[i, j]$ denotes the (i, j)-th entry of the matrix. By chain rule, we have

$$\frac{\partial \ell}{\partial w} = \frac{\partial \ell}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial w} \tag{6}$$

Note that the first two derivatives in the 6 are easy to compute, so let's focus on the last term $\frac{\partial h_t}{\partial w}$. During the lecture, we have shown that

$$\frac{\partial h_t}{\partial w} = \sum_{i=1}^{t} \frac{\partial h_t}{\partial h_i} \frac{\partial h_i^+}{\partial w} \tag{7}$$

Here $\frac{\partial h_i^+}{\partial w}$ denotes the "immediate" gradient where h_{i-1} is taken as a constant.

(a) (1 point, written) Give an expression for $\frac{\partial h_i^+}{\partial w}$.

We can express the immediate gradient $\frac{\partial h_i^+}{\partial w}$ by,

$$\frac{\partial h_i^+}{\partial w} = \sigma'(W_{\text{hh}} h_{i-1} + W_{\text{ih}} x_i + b_h) \cdot h_{i-1}[j] \tag{8}$$

(b) (2 points, written) Expand the gradient vector $\frac{\partial h_t}{\partial h_i}$ using the chain rule as a product of partial derivatives of one hidden state with respect to the previous hidden state. You do not need to explicitly do differentiations beyond that.

We know that in an RNN, each hidden state h_t depends on the previous hidden state h_{t-1} and through the recurring propagations, this holds for all hidden states back to h_i where $i < t$. Using the chain rule, we can express the gradient vector $\frac{\partial h_t}{\partial h_i}$ to be,

$$\frac{\partial h_t}{\partial h_i} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial h_{t-2}} \cdot \ldots \cdot \frac{\partial h_{i+1}}{\partial h_i} \tag{9}$$

From equation 2, we can find $\frac{\partial h_k}{\partial h_{k-1}}$ as,

$$\frac{\partial h_k}{\partial h_{k-1}} = \sigma'(W_{\text{hh}} h_{k-1} + W_{\text{ih}} x_k + b_h) \cdot W_{\text{hh}} \tag{10}$$

Substituting this back in equation 9, we can thus represent the gradient of the hidden state h_t with respect to another hidden state h_i as,

$$\frac{\partial h_t}{\partial h_i} = \prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \tag{11}$$

$$\frac{\partial h_t}{\partial h_i} = \prod_{k=i+1}^{t} \sigma'(W_{\text{hh}} h_{k-1} + W_{\text{ih}} x_k + b_h) \cdot W_{\text{hh}} \tag{12}$$

3. (2 points, written) Now let's further expand one of the partial derivatives from the previous question. Write down the Jacobian matrix $\frac{\partial h_{i+1}}{\partial h_i}$ by rules of differentiations. You can directly use σ' as the derivative of the activation function in the expression.

From equation 2, we can represent the hidden state h_{i+1} of an RNN as,

$$h_{i+1} = \sigma(W_{\text{hh}} h_i + W_{\text{ih}} x_{i+1} + b_h) \tag{13}$$

To obtain the Jacobian matrix $\left|\frac{\partial h_{i+1}}{\partial h_i}\right|_{d' \times d'}$, from equation 10, we can find the partial derivatives of each element in the hidden state h_{i+1} with respect to each element in the hidden state h_i, given by,

$$J = \left|\frac{\partial h_{i+1}}{\partial h_i}\right|_{d' \times d'} = \begin{bmatrix} \frac{\partial h_{i+1,1}}{\partial h_{i1}} & \frac{\partial h_{i+1,1}}{\partial h_{i2}} & \cdots & \frac{\partial h_{i+1,1}}{\partial h_{id'}} \\ \frac{\partial h_{i+1,2}}{\partial h_{i1}} & \frac{\partial h_{i+1,2}}{\partial h_{i2}} & \cdots & \frac{\partial h_{i+1,2}}{\partial h_{id'}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_{i+1,d'}}{\partial h_{i1}} & \frac{\partial h_{i+1,d'}}{\partial h_{i2}} & \cdots & \frac{\partial h_{i+1,d'}}{\partial h_{id'}} \end{bmatrix} \tag{14}$$

We can represent $h_{i+1,j}$ and its partial derivative with respect to ∂h_{ik} as $\frac{\partial h_{i+1,j}}{\partial h_{ik}}$, given by the following expressions,

$$h_{i+1,j} = \sigma(W_{hh,j1} h_{i1} + W_{hh,j2} h_{i2} + ... + W_{hh,jd'} h_{id'} + W_{ih,j1} x_{i+1,1} + ... + W_{ih,jd'} x_{i+1,d} + b_{h,j}) \tag{15}$$

$$\frac{\partial h_{i+1,j}}{\partial h_{i,k}} = \sigma'(W_{hh,j1} h_{i1} + W_{hh,j2} h_{i2} + ... + W_{hh,jd'} h_{id'} + W_{ih,j1} x_{i+1,1} + ... + W_{ih,jd'} x_{i+1,d} + b_{h,j}) \cdot W_{hh,jk} \tag{16}$$

where d' is the dimension of the hidden state h and $[j]$ and $[k]$ denote the j-th and k-th elements of the respective vectors. We can further simplify the equation to be,

$$\frac{\partial h_{i+1}}{\partial h_i}[j,k] = \sigma'\left(\sum_{d''=1}^{d'} W_{hh}[j,d''] h_i[d''] + \sum_{d''=1}^{d} W_{ih}[j,d''] x_{i+1}[d''] + b_h[j]\right) \cdot W_{hh}[j,k] \tag{17}$$

We can thus express the Jacobian matrix $\left|\frac{\partial h_{i+1}}{\partial h_i}\right|_{d' \times d'}$ as,

$$J = \left|\frac{\partial h_{i+1}}{\partial h_i}\right|_{d' \times d'} = \begin{bmatrix} \sigma'(A[1]) \cdot W_{\text{hh}}[1,1] & \sigma'(A[1]) \cdot W_{\text{hh}}[1,2] & \cdots & \sigma'(A[1]) \cdot W_{\text{hh}}[1,d'] \\ \sigma'(A[2]) \cdot W_{\text{hh}}[2,1] & \sigma'(A[2]) \cdot W_{\text{hh}}[2,2] & \cdots & \sigma'(A[2]) \cdot W_{\text{hh}}[2,d'] \\ \vdots & \vdots & \ddots & \vdots \\ \sigma'(A[d']) \cdot W_{\text{hh}}[d',1] & \sigma'(A[d']) \cdot W_{\text{hh}}[d',2] & \cdots & \sigma'(A[d']) \cdot W_{\text{hh}}[d',d'] \end{bmatrix} \tag{18}$$

where $A[j] = \sum_{d''=1}^{d'} W_{hh}[j,d''] h_i[d''] + \sum_{d''=1}^{d} W_{ih}[j,d''] x_{i+1}[d''] + b_h[j]$.

1.2 Bounding Gradient Norm

To determine if the gradient will vanish or explode, we need a notion of magnitude. For the Jacobian matrix, we can use the induced matrix norm (or operator norm). For this question, we use the spectral norm $\|A\|_2 = \sqrt{\lambda_{\max}(A^\top A)} = s_{\max}(A)$ for a matrix $A \in \mathbb{R}^{m \times n}$. Here $\lambda_{\max}(A^\top A)$ denotes the maximum eigenvalue of the matrix $A^\top A$ and $s_{\max}(A)$ denotes the maximum singular value of the matrix A. You can learn more about matrix norms at this Wikipedia entry.
Now, to determine if the gradient $\frac{\partial \ell}{\partial w}$ will vanish or explode, we can focus on $\|\frac{\partial h_t}{\partial h_i}\|$. Note that if $\|\frac{\partial h_t}{\partial h_i}\|$ vanishes or explodes, $\|\frac{\partial \ell}{\partial w}\|$ also vanishes or explodes based on (6) and (7).

1. (2 points, written) Given the mathematical form of the Jacobian matrix $\frac{\partial h_{i+1}}{\partial h_i}$ we derived earlier, we can now bound the norm of the Jacobian with the following matrix norm inequality

$$\|AB\|_2 \leq \|A\|_2 \cdot \|B\|_2 \tag{19}$$

for matrices A, B with matched shapes. Write down a bound for $\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2$.

From our earlier derivations (equation 10), we know that,

$$\frac{\partial h_i}{\partial h_{i-1}} = \sigma'(W_{\text{hh}} h_{i-1} + W_{\text{ih}} x_i + b_h) \cdot W_{\text{hh}} \tag{20}$$

Thus on applying the spectral norm, we obtain the following expression for the bound,

$$\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 = \|\sigma'(W_{\text{hh}} h_{i-1} + W_{\text{ih}} x_i + b_h) \cdot W_{\text{hh}}\|_2 \leq \|\sigma'(W_{\text{hh}} h_{i-1} + W_{\text{ih}} x_i + b_h)\|_2 \cdot \|W_{\text{hh}}\|_2 \tag{21}$$

$$\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 \leq \|\sigma'(W_{\text{hh}} h_{i-1} + W_{\text{ih}} x_i + b_h)\|_2 \cdot \|W_{\text{hh}}\|_2 \tag{22}$$

2. (4 points, written) Now we have all the pieces we need. Derive a bound on the gradient norm $\|\frac{\partial h_t}{\partial h_i}\|$. Explain how the magnitude of the maximum singular value of W_{hh} can lead to either vanishing or exploding gradient problems. [**HINT:** You can use the fact that for the tanh activation function $\sigma(\cdot)$, the derivative $\sigma'(\cdot)$ is always less than or equal to 1.]

From equation 22 we know that the gradient norm can be expressed as,

$$\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 \leq \|\sigma'(W_{\text{hh}}h_{i-1} + W_{\text{ih}}x_i + b_h)\|_2 \cdot \|W_{\text{hh}}\|_2 \tag{23}$$

Since the derivative of the *tanh* function σ' is always ≤ 1, we can simplify the expression as,

$$\left\|\frac{\partial h_i}{\partial h_{i-1}}\right\|_2 \leq \|W_{\text{hh}}\|_2 \tag{24}$$

From equation 12 we know that the gradient $\frac{\partial h_t}{\partial h_i}$ can be expressed using chain rule as follows,

$$\frac{\partial h_t}{\partial h_i} = \prod_{k=i+1}^{t} \frac{\partial h_k}{\partial h_{k-1}} \tag{25}$$

Thus, we can represent the bound of the gradient norm as:

$$\left\|\frac{\partial h_t}{\partial h_i}\right\|_2 \leq \left\|\frac{\partial h_t}{\partial h_{t-1}}\right\|_2 \cdots \left\|\frac{\partial h_{i+1}}{\partial h_i}\right\|_2 \tag{26}$$

$$\left\|\frac{\partial h_t}{\partial h_i}\right\|_2 \leq \|W_{\text{hh}}\|_2 \cdots \|W_{\text{hh}}\|_2 \tag{27}$$

$$\left\|\frac{\partial h_t}{\partial h_i}\right\|_2 \leq \|W_{\text{hh}}\|_2^{t-i} \tag{28}$$

Interpreting W_{hh} and its effect on Backpropagation.

(a) We can thus see that the **maximum singular value of the recurrent weight matrix** W_{hh} plays a crucial role in determining whether the gradients will explode or vanish.

(b) **Exploding Gradients**: If $\|W_{\text{hh}}\|_2 > 1$, then as $t - i$ increases, $\|W_{\text{hh}}\|_2^{t-i}$ will also **grow exponentially** as the power increases and the gradients will become extremely large, leading to exploding gradients.

(c) **Vanishing Gradients**: If $\|W_{\text{hh}}\|_2 < 1$, then as $t - i$ increases, $\|W_{\text{hh}}\|_2^{t-i}$ will also **decrease exponentially** as the power increases and the gradients will become extremely small until they become zero, leading to vanishing gradients.

(d) Thus, the gradient norm directly depends on the magnitude of the maximum singular value of the recurrent weight matrix and determines whether the gradients will vanish or explode during backpropagation, underscoring the importance of initialising W_{hh} carefully to ensure numerically stable gradients.

3. (1 point, written) Propose one way to get around the vanishing and exploding gradient problem.

   (a) **Gradient Clipping**: This approach clips gradient values if they become too large and constrains them to a specific range, ensuring their value never exceeds the set threshold. This technique prevents the gradients from becoming excessively large or small during training and makes them more numerical stable.

   (b) **Normalised Initialisation of Weights**: Initialising weights to specific values using different initialisation techniques (Xavier, He etc) can prevent gradients from becoming too large or small during training.

   (c) **Using Long Short-Term Memory (LSTM) Networks**: Using a variant of the vanilla RNN such as the LSTM network can help reduce the vanishing and exploding gradient problem by leveraging the forget input and output gates. These gates help the model to propagate gradients without exploding or vanishing by controlling what information can be discarded or stored. Thus, regulating how much information is stored in a cell's state can effectively get around the problem since the network can selectively remember or forget information as needed at each time step, preventing gradients from getting too large or small during training.

2 Machine Translation

The goal of this homework is to build a machine translation system using sequence-to-sequence transformer models `https://arxiv.org/abs/1706.03762`. More specifically, you will build a system which translates German to English using the Multi30k dataset (`https://arxiv.org/abs/1605.00459`) You are provided with a code skeleton, which clearly marks out where you need to fill in code for each sub-question.

First go through the file `README.md` to set up the environment required for the class.

2.1 Attention

Transformers use scaled dot-product attention — given a set of queries Q (each of dimension d_k), a set of keys K (also each dimension d_k), and a set of values V (each of dimension d_v), the output is a weighted sum of the values. More specifically,

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{29}$$

Note that each of Q, K, V is a matrix of vectors packed together.

1. (2 points, written) The above function is called 'scaled' attention due to the scaling factor $\frac{1}{\sqrt{d_k}}$. The original transformers paper mentions that this is needed because dot products between keys and queries get large with larger d_k.

   For a query q and key k both of dimension d_k and each component being an independent random variable with mean 0 and variance 1, compute the mean and variance (with steps) of the dot product $q.k$ to demonstrate the point.

Computing Mean : We can represent the dot product $q \cdot k$ between a query q and key k by the following,

$$q \cdot k = \sum_{i=1}^{d_k} q_i k_i \tag{30}$$

where q_i and k_i are each component of the query q and key k respectively and each being an independent random variable with mean 0 and variance 1. We can compute the mean of the dot product using the linearity of expectation such that,

$$\mathbb{E}[q \cdot k] = \mathbb{E}\left[\sum_{i=1}^{d_k} q_i k_i\right] = \sum_{i=1}^{d_k} \mathbb{E}[q_i k_i] \tag{31}$$

Since each component is independent, the expected value of their product is the product of their expected values.

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \cdot \mathbb{E}[k_i] \tag{32}$$

Given that $\mathbb{E}[q_i] = \mathbb{E}[k_i] = 0$, this also implies that $\mathbb{E}[q_i k_i] = 0 \cdot 0 = 0$. We can substitute this in the previous equation to obtain the following expression for the mean of the dot product as,

$$\mathbb{E}[q \cdot k] = \sum_{i=1}^{d_k} \mathbb{E}[q_i k_i] = \sum_{i=1}^{d_k} 0 = 0 \tag{33}$$

Thus, the mean of the dot product $\mathbb{E}[q \cdot k]$ is zero.
Computing Variance : Given that the variance of a sum of independent random variables is the sum of their variances, we can represent the variance $Var(q \cdot k)$ as,

$$Var(q \cdot k) = Var\left(\sum_{i=1}^{d_k} q_i k_i\right) = \sum_{i=1}^{d_k} Var(q_i k_i) \tag{34}$$

Since each term *qiki* is a product of two independent random variables, the variance of their product is given by,

$$Var(q_ik_i) = \mathbb{E}[(q_ik_i)^2] - (\mathbb{E}[q_ik_i])^2 \tag{35}$$

Since q_i and k_i are independent components with $\mathbb{E}[q_i] = 0$ and $\mathbb{E}[k_i] = 0$, we also have $\mathbb{E}[q_ik_i] = 0$. This simplifies the expression to,

$$Var(q_ik_i) = \mathbb{E}[(q_ik_i)^2] \tag{36}$$

$$Var(q_ik_i) = \mathbb{E}[q_i^2] \cdot \mathbb{E}[k_i^2] = 1 \times 1 = 1 \tag{37}$$

We can thus express the variance $Var(q \cdot k)$ of the dot product as d_k,

$$Var(q \cdot k) = \sum_{i=1}^{d_k} 1 = d_k \tag{38}$$

Interpretation : As the dimension d_k of the query and key increases, the variance increases, thus in turn increasing the magnitude of the dot product $q \cdot k$. This leads to numerical instability. The scaling factor is included to prevent these values from becoming excessively large and prevent the softmax function from producing sharp distributions, constraining it to an acceptable range. It also mitigates the problem of the attention mechanism being less sensitive to differences in the similarity scores between the query and the key due to the high values from the softmax function.

2. (2 points, coding) Implement the above scaled dot-product attention in the `attention()` function present in `layers.py`. You can test the implementation after the next part.

```python
def attention(query, key, value, mask=None, dropout=None):
    """Applies scaled dot-product attention mechanism to query, key, value pairs."""
    d_k = query.size(-1)
    attn_scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    attn_weights = torch.softmax(attn_scores, dim=-1)
    return torch.matmul(attn_weights, value), attn_weights
```

3. (2 point, coding) In this part, you will modify the `attention()` function by making use of the parameters `mask` and `dropout` which were input to the function. The `mask` indicates positions where the attention values should be zero (e.g. when we have padded a sentence of length 5 to length 10, we do not want to attend to the extra tokens). `dropout` should be applied to the attention weights for regularization.

   To test the implementation against some unit tests, run `python3 test.py --attention`.

```python
def attention(query, key, value, mask=None, dropout=None):
    """Applies scaled dot-product attention mechanism to query, key, value pairs.
    Attention masks and dropout are applied if provided."""
    d_k = query.size(-1)
    attn_scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
    if mask is not None:
        attn_scores = attn_scores.masked_fill(mask.unsqueeze(1) == 0, -1e9)
    attn_weights = torch.softmax(attn_scores, dim=-1)
    if dropout is not None:
        attn_weights = torch.dropout(attn_weights, dropout, train=True)
    return torch.matmul(attn_weights, value), attn_weights
```

4. (3 points, coding) Instead of a single attention function, transformers use multi-headed attention function. For original keys, queries and values (each of dimension say d_{model}), we use h different projection matrices to obtain queries, keys and values of dimensions d_k, d_k and d_v respectively. Implement the function `MultiHeadedAttention()` in `layers.py`. To test the implementation against some unit tests, run `python3 test.py --multiheaded_attention`.

```python
class MultiHeadedAttention(nn.Module):
    """
    Multi-headed attention mechanism with 'h' heads.
    """

    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.num_heads = h
        self.dropout = dropout
        self.linears = nn.ModuleList([nn.Linear(d_model, d_model) for _ in range(4)])
        self.attn = None

    def forward(self, query, key, value, mask=None):
        """Compute 'h' attention outputs and concatenate them."""
        batch_size = query.size(0)
        query, key, value = [
            linear_projection(x)
            .view(batch_size, -1, self.num_heads, self.d_k)
            .transpose(1, 2)
            for linear_projection, x in zip(self.linears, (query, key, value))
        ]
        x, self.attn = attention(query, key, value, mask=mask, dropout=self.dropout)
        x = (
            x.transpose(1, 2)
            .contiguous()
            .view(batch_size, -1, self.num_heads * self.d_k)
        )
        return self.linears[-1](x)
```

2.2 Positional Encoding

Since the underlying blocks in a transformer (namely attention and feed forward layers) do not encode any information about the order of the input tokens, transformers use 'positional encodings' which are added to the input embeddings. If d_{model} is the dimension of the input embeddings, pos is the position, and i is the dimension, then the encoding is defined as:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{model}}) \tag{39}$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}}) \tag{40}$$

1. (2 points, written) Since the objective of the positional encoding is to add information about the position, can we simply use $PE_{pos} = sin(pos)$ as the positional encoding for pos position? Why or why not?

No, we cannot simply use the sine function as the positional encoding for *pos* position for the following reasons,

(a) **Periodicity**: The sine function is periodic, meaning its **values will repeat after a certain interval**. This implies that the positional encoding for a position *pos* and another position separated by multiples of the time period $pos + n \times timePeriod$ will hold the same value for the positional encoding. To alleviate this problem, positional encoding is implemented using a combination of sine and cosine functions with different frequencies, which allows the model to distinguish between positions separated by different distances.

(b) **Lack of Variation across Dimensions**: The positional encoding functions **vary across both the position** *pos* **and the dimension** i of the embedding and are accompanied by a scaling factor that changes the frequency of the function across each dimension, meaning that each position is encoded differently across each dimension. The gradual change in frequencies as i increases allows the model to **distinguish nearby positions** by combining information from different embedding dimensions – lower dimensions with higher frequencies capture local relationships between close-by tokens while higher dimensions with lower frequencies capture global patterns. This is not possible with a simple sinusoidal function.

Thus, using $PE_{pos} = sin(pos)$ as the positional encoding for *pos* position would not provide the advantages such as variation across dimensions to effectively represent positions and would also suffer from duplicate positional encodings due to its periodicity.

2. (2 points, coding) Implement the above positional encoding in the function `PositionalEncoding()` in the file `utils.py`. To test the implementation against some unit tests, run `python3 test.py --positional_encoding`.

```python
class PositionalEncoding(nn.Module):
    "Implement the PE function."

    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)

        self.register_buffer("pe", pe)

    def forward(self, x):
        pe = self.pe.unsqueeze(0)
        x = x + pe[:, : x.size(1)].requires_grad_(False)
        return self.dropout(x)
```

2.3 Training

1. (2 points, coding) The above questions should complete the missing parts in the training code and we can now train a machine translation system!

   Use the command `python3 main.py` to train your model. For the purpose of this homework, you are not required to tune any hyperparameters. You should submit the generated `out_greedy.txt` file containing outputs. You must obtain a BLEU score of atleast 35 for full points (By default we are using BLEU-4 for this and all subsequent questions).

2.4 Decoding & Evaluation

In the previous question, the code uses the default `greedy_decode()` to decode the output. In practice, people use algorithms such as beam search decoding, which have been shown to give better quality outputs. (Note: In the following questions, use a model trained with the default i.e. given hyperparameters)

1. (2 points, written) In the file `utils.py` you will notice a function `subsequent_mask()`. What does that function do and why is it required in our model?

   (a) The `subsequent_mask()` function is used to create a mask and ensure that the Transformer model's decoder **does not attend to future positions** during training while predicting the next token. It prevents the model from looking at future tokens while generating or processing sequences during training.

   (b) This is crucial since in tasks such as machine translation, the output sequence is generated one token at a time and during this process, we want the **decoder to not peek at future tokens and only attend to previously generated tokens** along with the original input sentence i.e, it should generate tokens sequentially with knowledge about only the tokens it has generated in the past. The `subsequent_mask()` function ensures that at each time step t, the model is able to attend to only the current and past tokens and thus ensure the autoregressive property of machine translation.

   (c) If this function is not applied, then the model would perform extremely well during training (since the entire output sequence and thus the future words are available) but would **perform poorly in real world inference** since future words would not be available for the model to depend on.

2. (5 points, coding) Implement the `beam_search()` function in `utils.py`. We have provided the main skeleton for this function and you are only required to fill in the important parts (more details in the code). You can run the code using the arguments `--beam_search` and `--beam_size`. You should submit the generated file `out_beam.txt` when `beam_size = 2`.

   To test the implementation against some unit tests, run `python3 test.py --beam_search`.

```python
def beam_search_decode(model, src, src_mask, max_len, start_symbol, beam_size, end_idx):
    """
    Implement beam search decoding with 'beam_size' width
    """
    memory = model.encode(src, src_mask)
    ys = torch.zeros(1, 1).fill_(start_symbol).type_as(src.data)
    scores = torch.zeros(1, device=src.device)  # Initialize scores for each beam
    # A list to hold beams and their scores at each step
    beams = [(ys, scores)]

    # Expand encoder output memory for each beam in the search
    memory = memory.expand(beam_size, *memory.shape[1:])

    for _ in range(max_len - 1):
        candidates = list()
        # Iterate over each beam and its score
        for ys, score in beams:
            # Decode the next token
            out = model.decode(
                memory[: ys.size(0)],
                src_mask,
                ys,
                subsequent_mask(ys.size(1)).type_as(src.data),
            )
            # Get probabilities for the next token
            prob = model.generator(out[:, -1])
            log_probs = torch.log_softmax(prob, dim=1)
            # Get top 'beam_size' candidates for the next token and their scores
            top_k_log_probs, top_k_tokens = log_probs.topk(beam_size, dim=1)
            top_k_log_probs += score  # Update scores with previous scores

            # Append candidates for the next beam state
            for j in range(beam_size):
                new_seq = torch.cat([ys, top_k_tokens[:, j].view(1, -1)], dim=1)
                new_score = top_k_log_probs[0, j]
                candidates.append((new_seq, new_score))

        # Sort all candidates by their scores and keep only the top 'beam_size' beams
        beams = sorted(candidates, key=lambda x: x[1], reverse=True)[:beam_size]

        # Check if all beams have generated the end token
        if all(beam[0][0, -1].item() == end_idx for beam in beams):
            break

    # Return the best sequence with the highest score
    return beams[0][0]
```

3. (3 points, written) For the model trained in question 1.3, plot the BLEU score as a function of beam size. You should plot the output from beam size 1 to 5. Is the trend as expected? Explain your answer.

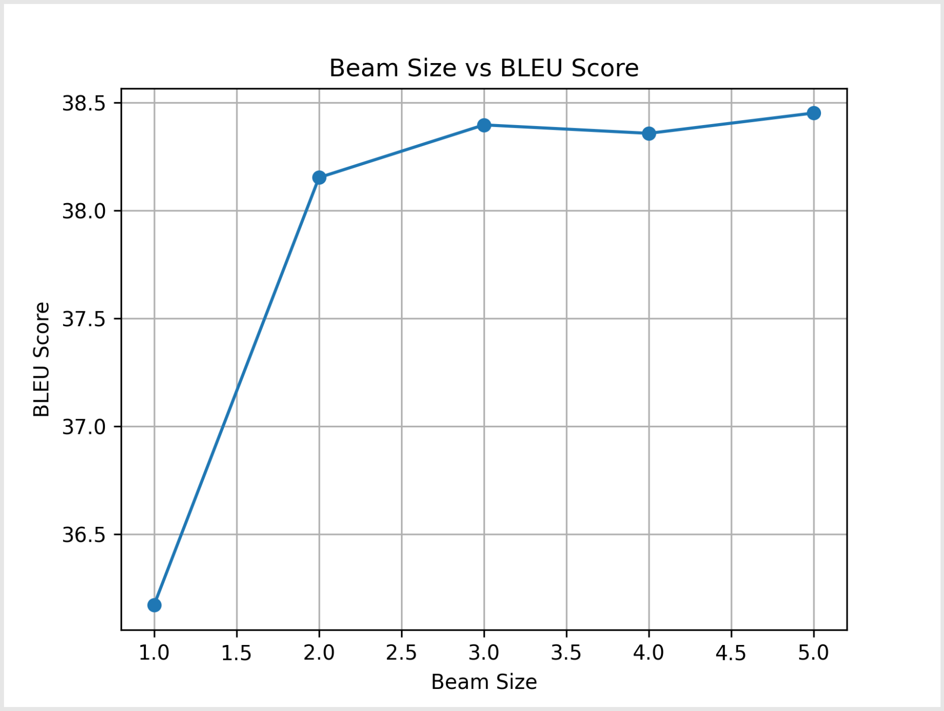

Figure 1: **Beam Size vs BLEU Score** shows the relationship between the beam size used during decoding and the BLEU score obtained for the Transformer model. We see that with an increase in beam size, there is a **significant increase in the BLEU score until it plateaus**.

(a) We initially observe a **sharp increase in the BLEU score** as the beam size increases from 1 to 3 with the BLEU score rising from **36.5** to **38.4**. Beyond a beam size of 3, we observe only a slight increase in the BLEU score.

(b) This behaviour is ***expected*** since as the beam size increases, **it allows the model to explore multiple possible translations** at each time step t and eliminates the candidate translations with a low probability. As the number of possible translations increases, **the model's chance of finding a more accurate output translation increases**. The model thus finds better translations that align with the source sentence, increasing the BLEU score.

(c) We observe the **largest jump from a beam size of 1 to 2** since a beam size of 1 implies a greedy decoding strategy where alternate translations aren't explored. Thus we observe the lowest BLEU score for this beam size. The significant increase is due to an increase in the likelihood of the model finding a more accurate translation.

(d) We also observe that at a **beam size of 3, the performance plateaus and there aren't significant improvements** beyond it. This implies that a beam size of 3 sufficiently allows the model to explore all the accurately translated candidates and it is already able to sample accurate translations. Since a **majority of the accurate translation paths have been explored, it implies that there aren't too many better translation candidates**, increasing the beam size does not greatly improve performance beyond a point.

(e) We also observe a slight fluctuation for a beam size of 4. This is because, with an increase in beam size, we **might also encounter low-probability translations** which might not best align with the source text and thus decrease the overall BLEU score.

4. (2 points, written) You might notice that some of the sentences contain the '⟨unk⟩' token which denotes a word not in the vocabulary. For systems such as Google Translate, you might not want such tokens in the outputs seen by the user. Describe a potential way to avoid (or reduce) the occurrence of these tokens in the output.

The '⟨unk⟩' token is a placeholder token and is generated during decoding when the model encounters a word that is **not in its vocabulary (OOV)** or was extremely rare in the training data. There are multiple ways to mitigate this problem for applications such as Google Translate where these tokens are undesirable such as,

(a) **Subword level Tokenisation**: By tokenising at the sub-word level instead of the word level, we can **break down words into smaller units or tokens**, which significantly reduces the occurrence of these '⟨unk⟩' tokens in the output, since the model can now **represent words it has not seen by combining various sub-word units**. This helps to artificially increase the vocabulary size without actually requiring more training data and helps handle rare or unknown words more meaningfully. Tokenisation techniques such as **SentencePiece** or **Byte-Pair Encoding** are examples of two approaches to sub-word level tokenisation. We can additionally also try character level tokenisation, however, it does not scale well at capturing sub-word level (or word level) relationships.

(b) **Increasing the Vocabulary**: While not always possible, **increasing the vocabulary size by training on more data** reduces the occurrences of '⟨unk⟩' tokens in the output. Additionally, modern-day transfer learning approaches can also be applied where a large **language model pre-trained on large amounts of source and translated text can be fine-tuned for translation**. The pre-training would help the model learn a vast vocabulary spanning both languages, thus reducing the occurrence of '⟨unk⟩' tokens being generated.

(c) **Copy Mechanism**: A copy mechanism can also be applied to allow the model to **copy unknown words from the source sentence into the target translation** instead of generating '⟨unk⟩' tokens by learning when to copy a word instead of translating it. This is typically done for words that are to remain unchanged between source and translated sentences but can be expanded upon for '⟨unk⟩' tokens as well.

5. (2 points, written) In this homework, you have been using BLEU score as your evaluation metric. Consider an example where the reference translation is ”I just went to the mall to buy a table.”, and consider two possible model generations: ”I just went to the mall to buy a knife.” and ”I just went to the mall to buy a desk.”. Which one will BLEU score higher? Suggest a potential fix if it does not score the intended one higher.

(a) The BLEU score evaluates sentences based on n-gram precision, meaning it evaluates **how many n-grams match between the reference and generated sentence** and applies a penalty if the generated sentence is too short. The value of n is usually taken up to 4.

(b) In both candidate sentences, only the last word of each sentence (”knife” and ”desk” respectively) differs from the reference sentence. This implies that their corresponding n-gram precision **score will be nearly identical** since only the last 1-gram will differ across all n-gram computations and every other word will match perfectly.

(c) Given that both sentences will score the same, both are treated as equally wrong. Since **BLEU only looks at word matches on the surface level and does not account for word meanings**, the sentence ending with the word ”desk” is not scored higher, even though it would be the more appropriate or closer translation compared to the sentence ending with the word ”knife” since they are semantically closer.

(d) To mitigate this problem, we can factor in **semantic similarity** using similarity metrics like cosine similarity as a combined metric to determine **how semantically similar two sentences are**. Since a ”desk” is closer in meaning compared to a ”knife”, it *should* be scored higher and thus be a valid translation. We can incorporate this change by first computing a weighted average of the n-gram similarity score between the reference and translated sentence and then combining this with the BLEU score.

$$\text{Semantic Score} = \frac{\sum_{n=1}^{N} n \cdot \text{similarity}(n_n\text{-grams}_{reference}, n_n\text{-grams}_{translation})}{\sum_{n=1}^{N} n} \tag{41}$$

$$\text{MT Score} = \alpha \cdot \text{Semantic Score} + (1-\alpha) \cdot BLEUscore \tag{42}$$

(e) The above metric would thus not only account for n-gram precision but also for semantic similarity between the sentences. The parameter α would allow us to maintain a **balance between the semantic similarity and structural or syntactical similarity** between sentences (since we do not want sentences to be extremely paraphrased [high semantic similarity, low syntactical similarity] either). Thus, by combining semantic similarity, we can improve the BLEU metric by not only looking at word overlap but also factoring for cases where a **semantically similar but syntactically dissimilar** sentence has been generated. This will help it to be closer to human judgement by scoring it higher than an irrelevant translation.