CPEN 455: Deep Learning

PA2

Qihang Zhang

University of British Columbia

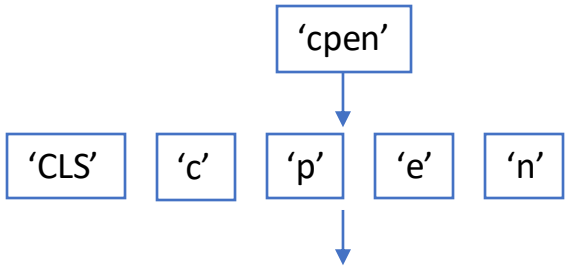
Winter, Term 2, 2024

Preprocessing and Tokenization

1.1 [10pts] Implement the tokenization and vectorization functionality in the Tokenizer class. This function takes in a string, (1) splits the string, and returns a list of characters (or *tokens* in Transformer terminology) (2) (optionally) add a "[CLS]" token to the beginning of the list (later, we will see why we need this). (3) convert each token into a one-hot vector and return the resulting matrix. For a string of length n, this function must return a row-wise one-hot matrix $\mathbf{X} \in \mathbb{R}^{(n+1)\times d_{\text{voc}}}$ where $d_{\text{voc}} = 4 + 1 = 5$ is the size of our token vocabulary.

```python
def tokenize_string(self, string, add_cls_token=True) -> torch.Tensor:
    """
    Tokenize the input string according to the above vocab

    START BLOCK
    """
    arr = list(string)
    if add_cls_token:
        arr = ['[CLS]'] + arr
    indices = torch.tensor(list(map(lambda x: self.vocab[x], arr)))
    tokenized_string = F.one_hot(indices, num_classes=len(self.vocab)).float()
    """
    END BLOCK
    """
    return tokenized_string
```

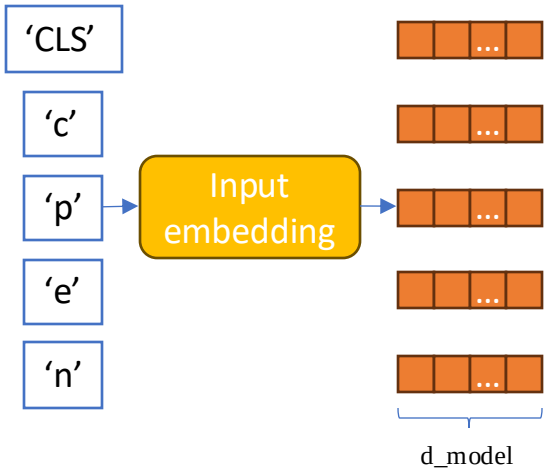


tensor([0, 1, 2, 3, 4])

tensor([[1., 0., 0., 0., 0.],
[0., 1., 0., 0., 0.],
[0., 0., 1., 0., 0.],
[0., 0., 0., 1., 0.],
[0., 0., 0., 0., 1.]])

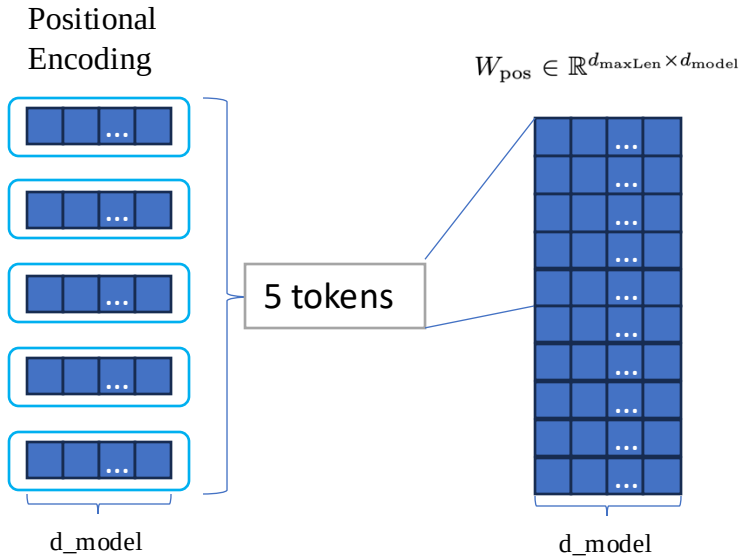
Positional Encoding

1.2 [5pts] Implement the (absolute) learnable positional encoding module. This module has a learnable weight matrix $W_{\text{pos}} \in \mathbb{R}^{d_{\text{maxLen}} \times d_{\text{model}}}$. The i-th row $(1 \leq i \leq d_{\text{maxLen}})$ of this matrix is a learnable vector corresponding to the i-th position in a sequence. This module applies positional encoding to a sequence by element-wise adding rows of this matrix to their corresponding position in the input.



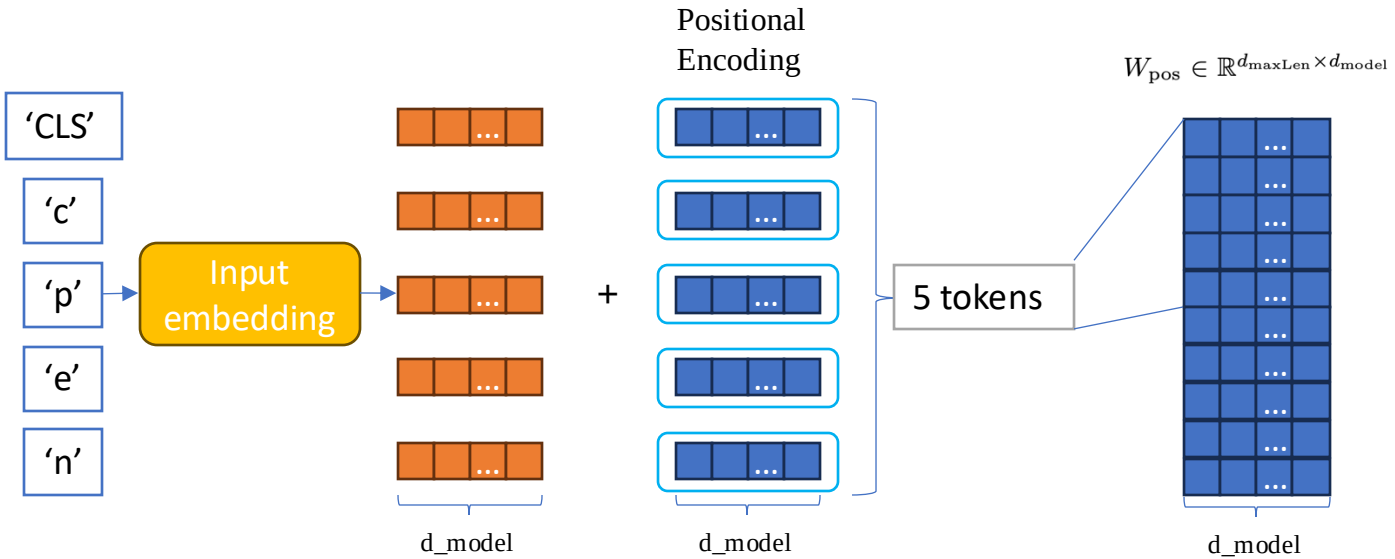
Positional Encoding

1.2 [5pts] Implement the (absolute) learnable positional encoding module. This module has a learnable weight matrix $W_{\text{pos}} \in \mathbb{R}^{d_{\text{maxLen}} \times d_{\text{model}}}$. The i-th row $(1 \leq i \leq d_{\text{maxLen}})$ of this matrix is a learnable vector corresponding to the i-th position in a sequence. This module applies positional encoding to a sequence by element-wise adding rows of this matrix to their corresponding position in the input.



Positional Encoding

1.2 [5pts] Implement the (absolute) learnable positional encoding module. This module has a learnable weight matrix $W_{\text{pos}} \in \mathbb{R}^{d_{\text{maxLen}} \times d_{\text{model}}}$. The i-th row $(1 \leq i \leq d_{\text{maxLen}})$ of this matrix is a learnable vector corresponding to the i-th position in a sequence. This module applies positional encoding to a sequence by element-wise adding rows of this matrix to their corresponding position in the input.



Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_h}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```

$\mathbf{X}_Q\mathbf{W}_{Q,h}$

$\mathbf{X}_K\mathbf{W}_{K,h}$

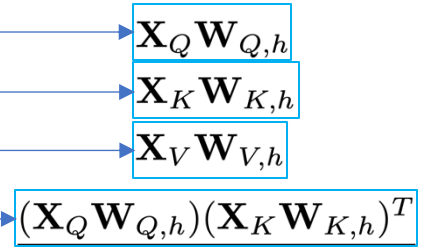
$\mathbf{X}_V\mathbf{W}_{V,h}$

Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_h}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```



Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```

$\mathbf{X}_Q\mathbf{W}_{Q,h}$

$\mathbf{X}_K\mathbf{W}_{K,h}$

$\mathbf{X}_V\mathbf{W}_{V,h}$

$(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T$

$\text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)$

$(\mathbf{X}_V\mathbf{W}_{V,h})$

Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$\text{Attention}(\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}_O.$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

$$\text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)$$

$$(\mathbf{X}_V\mathbf{W}_{V,h})$$

Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$\text{Attention}(\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}_O.$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

$$\text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)$$

Multi-Head Attention

1.3 [20pts] Implement the multi-head attention module.

$$\text{Attention}(\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}_O.$$

```
for h in range(self.n_heads):
    Q = query @ self.Wq[h][None, ...]
    K = key @ self.Wk[h][None, ...]
    V = value @ self.Wv[h][None, ...]
    QK = Q @ torch.transpose(K, -1, -2)
    if self.rpe:
        QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
    QKV_soft = torch.softmax(QK / math.sqrt(self.d_h), dim=-1) @ V
    Vs.append(QKV_soft)
Vs = torch.concat(Vs, dim=-1)
O = Vs @ self.Wo
```

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

$$\text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)$$

$$\text{Attention}(\mathbf{X}_K, \mathbf{X}_Q, \mathbf{X}_V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_H)\mathbf{W}_O.$$

Multi-Head Attention

1.5 [5pts] An alternative way of imposing positional encoding in the Transformer is to apply it in the attention Softmax operator. That is, modify the equation of each head as follows:

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T + \mathbf{M}_h}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h}),$$

where $\mathbf{M}_h \in \mathbb{R}^{n\times n}$ is a learnable Toeplitz matrix

$$M_h = \begin{bmatrix} m_{0,h} & m_{1,h} & m_{2,h} & \cdots & m_{n-1,h} \\ m_{-1,h} & m_{0,h} & m_{1,h} & \cdots & m_{n-2,h} \\ m_{-2,h} & m_{-1,h} & m_{0,h} & \cdots & m_{n-3,h} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{-(n-1),h} & m_{-(n-2),h} & m_{-(n-3),h} & \cdots & m_{0,h} \end{bmatrix},$$

```
if rpe:
    # -MAX_LEN, -MAX_LEN+1, ..., -1, 0, 1, ..., MAX_LEN-1, MAXLEN
    self.rpe_w = nn.ParameterList([nn.Parameter(torch.empty((2*self.MAX_LEN+1, ))) for _ in range(n_heads)])
```

Multi-Head Attention

where $\mathbf{M}_h \in \mathbb{R}^{n \times n}$ is a learnable Toeplitz matrix

$$M_h = \begin{bmatrix} m_{0,h} & m_{1,h} & m_{2,h} & \cdots & m_{n-1,h} \\ m_{-1,h} & m_{0,h} & m_{1,h} & \cdots & m_{n-2,h} \\ m_{-2,h} & m_{-1,h} & m_{0,h} & \cdots & m_{n-3,h} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{-(n-1),h} & m_{-(n-2),h} & m_{-(n-3),h} & \cdots & m_{0,h} \end{bmatrix},$$

```python
if rpe:
    # -MAX_LEN, -MAX_LEN+1, ..., -1, 0, 1, ..., MAX_LEN-1, MAXLEN
    self.rpe_w = nn.ParameterList([nn.Parameter(torch.empty((2*self.MAX_LEN+1, ))) for _ in range(n_heads)])
```

```python
def _rpe_square_mat(N, h):
    # Let's be inefficient for better readability
    device = key.device
    rpe_mat = torch.zeros((N, N), device=device)
    for offset in range(-(N-1), N):
        a = torch.diagflat(torch.ones(N - abs(offset), device=device), offset=offset)
        rpe_mat = rpe_mat + a * self.rpe_w[h][self.MAX_LEN + offset]
    return rpe_mat
```

N = 3, offset = -1

0, 0, 0
1, 0, 0
0, 1, 0

Multi-Head Attention

$$head_h = \text{Softmax}\left(\frac{(\mathbf{X}_Q\mathbf{W}_{Q,h})(\mathbf{X}_K\mathbf{W}_{K,h})^T + \mathbf{M}_h}{\sqrt{d_\text{h}}}\right)(\mathbf{X}_V\mathbf{W}_{V,h})$$

```
if self.rpe:
    QK = QK + _rpe_square_mat(key.size(1), h)[None, ...]
```

Transformer Backbone

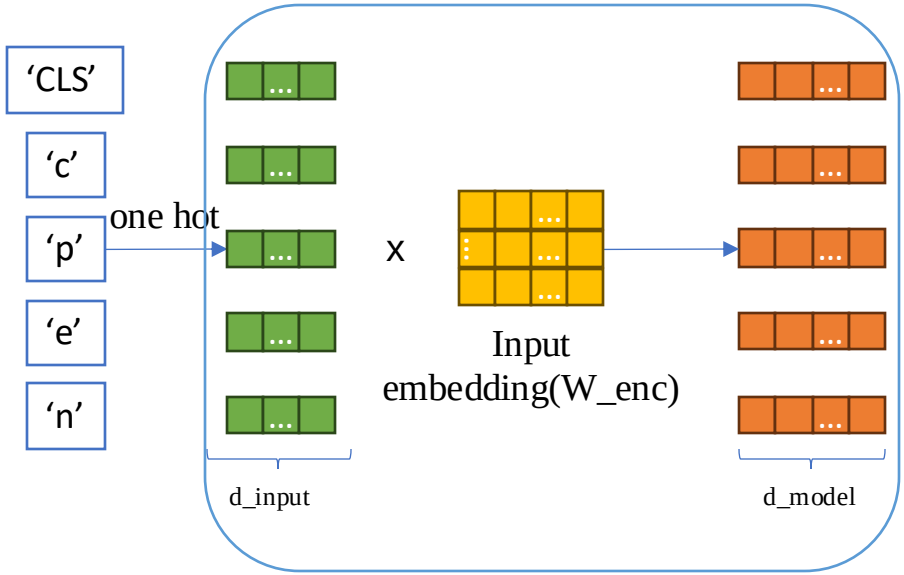
$$FC(X) = \text{ReLU}(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2.$$

1.4 [15pts] Implement the transformer layer module. The *prenorm* parameter determines whether the transformer layer is Pre-Norm or Post-Norm.

```
def forward(self, x):
    """
    args:
        x: shape B x N x D
    returns:
        out: shape B x N x D
    START BLOCK
    """
    out = x
    if self.prenorm:
        out_norm = self.ln1(out)
        out = out + self.attention(out_norm, out_norm, out_norm)
        out_norm = self.ln2(out)
        out = out + self.relu(out_norm @ self.fc_W1) @ self.fc_W2
    else:
        out = self.ln1(out + self.attention(out, out, out))
        out = self.ln2(out + self.relu(out @ self.fc_W1) @ self.fc_W2)
    """
    END BLOCK
    """
    return out
```

Transformer Backbone

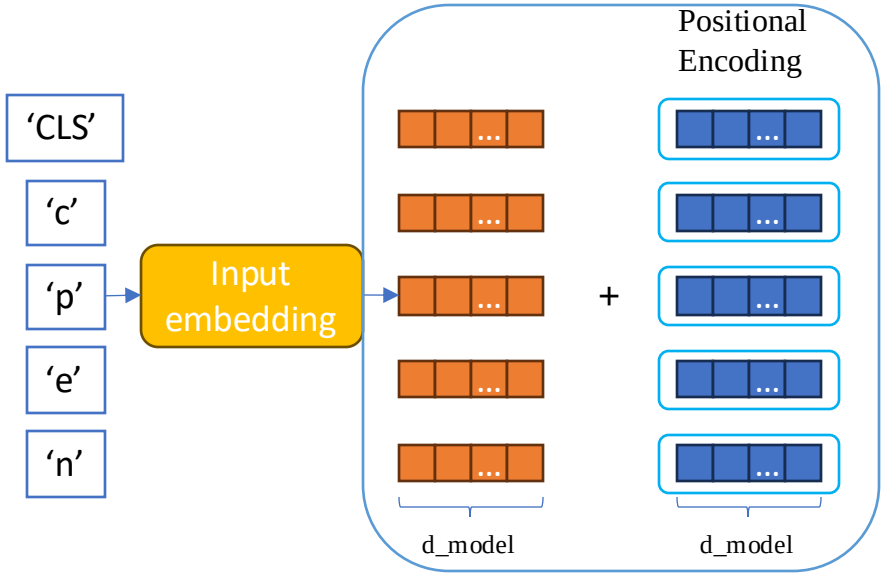
1.5 [15pts] Implement the transformer model module. This module contains an encoder, *nLayers* layers of transformer layer, and a decoder. The encoder weight $\mathbf{W}_{\text{enc}} \in \mathbb{R}^{d_{\text{input}} \times d_{\text{model}}}$ is a linear layer that takes in tokens of dimension $d_{\text{input}} = d_{\text{vocab}}$ and encodes each of them into a high-dimensional space of d_{model}. After applying the required layers of the Transformer layer, the decoder weight matrix $\mathbf{W}_{\text{dec}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{out}}}$ decodes each token from a high-dimensional space into the output space.




```python
def forward(self, x):
    """
    args:
        x: shape B x N x D_in
    returns:
        out: shape B x N x D_out
    START BLOCK
    """
    out = x @ self.enc_W
    if self.cfg.pos_enc_type == 'ape':
        out = self.ape(out)
    for layer in self.transformer_layers:
        out = layer(out)
    out = out @ self.dec_W
    """
    END BLOCK
    """
    return out
```

Transformer Backbone

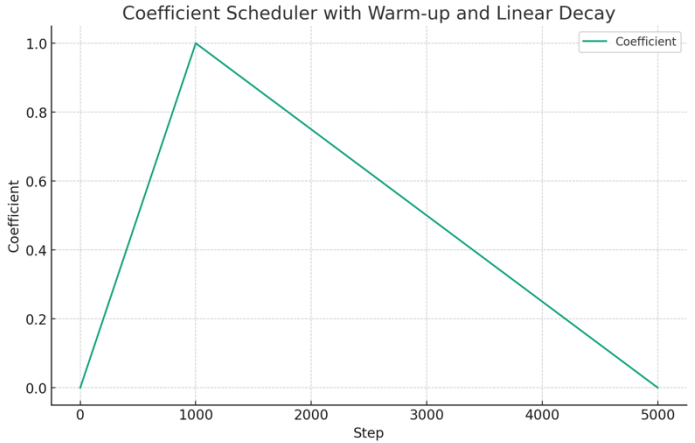
1.5 [15pts] Implement the transformer model module. This module contains an encoder, *nLayers* layers of transformer layer, and a decoder. The encoder weight $\mathbf{W}_{\text{enc}} \in \mathbb{R}^{d_{\text{input}} \times d_{\text{model}}}$ is a linear layer that takes in tokens of dimension $d_{\text{input}} = d_{\text{vocab}}$ and encodes each of them into a high-dimensional space of d_{model}. After applying the required layers of the Transformer layer, the decoder weight matrix $\mathbf{W}_{\text{dec}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{out}}}$ decodes each token from a high-dimensional space into the output space.




```python
def forward(self, x):
    """
    args:
        x: shape B x N x D_in
    returns:
        out: shape B x N x D_out
    START BLOCK
    """
    out = x @ self.enc_W
    if self.cfg.pos_enc_type == 'ape':
        out = self.ape(out)
    for layer in self.transformer_layers:
        out = layer(out)
    out = out @ self.dec_W
    """
    END BLOCK
    """
    return out
```

Optimization Scheduling

1.6 [5pts] Implement a scheduler with *warmUp* warmup steps and *maxSteps* total number of steps. In other words, your scheduler must return a zero learning rate at step 0, increase the learning rate linearly to *lr* until step *warmUp*, and decrease it again linearly to zero until step *maxSteps*.




```python
def get_lr(self):
    """
    Compute the custom scheduler with warmup and cooldown
    Hint: self.last_epoch contains the current step number
    START BLOCK
    """
    mult_factor = 1.0
    if self.last_epoch <= self.warmup_steps:
        mult_factor = self.last_epoch / self.warmup_steps
    elif self.warmup_steps < self.last_epoch < self.total_steps:
        mult_factor = 1.0 - (self.last_epoch - self.warmup_steps) / (self.total_steps - self.warmup_steps)
    else:
        mult_factor = 0.0
    """
    END BLOCK
    """
    return [group['initial_lr'] * mult_factor for group in self.optimizer.param_groups]
```

Train Substring Matching

1.7 [15pts] Implement the loss computation function in the Trainer class. The Transformer contains n output vectors for input with n tokens. However, our task is to have a single output for the entire sequence (*i.e.* the label). Hence, we have added a dummy token called "[CLS]" token, and will only look at the output of this token to compute the loss and accuracy. The loss/accuracy computation function should compute the predicted label, and binary cross entropy with the ground truth labels, and return both the loss and accuracy for a given batch. You may use the cross entropy loss function from PyTorch to compute loss.

self.model(x).shape:
[bs, L + 1, d_out]

```
def compute_batch_loss_acc(self, x, y):
    """
    Compute the loss and accuracy of the model on batch (x, y)
    args:
        x: B x N x D_in
        y: B
    return:
        loss, accuracy
    START BLOCK
    """
    x, y = x.to(self.device), y.to(self.device)
    y_logits = self.model(x)[:, 0, :].squeeze(-1)
    y_hat = (y_logits > 0.0)
    loss = F.binary_cross_entropy_with_logits(y_logits, y.float())
    acc = (y == y_hat).float().mean()
    """
    END BLOCK
    """
    return loss, acc
```