Github: https://github.com/shtosti/mt-exercise-4

1. Understanding Code: LayerNorm in JoeyNMT

1.1 Background

See the repository for the added literature:

1.2 Instances of LayerNorm in JoeyNMT

JoeyNMT uses LayerNorm from the PyTorch nn module.

- **Where and how are pre- and post-norm implemented?**
  joeynmt/joeynmt/transformer_layers.py, in 3 different classes.

- **What is the default behavior?**
  The default behavior is defined in the parameters for each class (see below). This sets the default, which can be easily switched there, whereas inside each class the position of the layer normalization is defined as conditional (if, else).

- What are the specific differences between the two options and how do you control them?

  "pre" Layer Normalization:

  - layer normalization is applied before the sub-layer operation.
  - PositionwiseFeedForward: layer normalization is applied to the input x before the feed-forward operation.
  - TransformerEncoderLayer: layer normalization is applied to the input x before the self-attention operation.
  - TransformerDecoderLayer: layer normalization is applied to the input x and h1 before the self-attention and cross-attention operations, respectively.

  "post" Layer Normalization:

  - layer normalization is applied after the sub-layer operation.

- PositionwiseFeedForward: layer normalization is applied to the output of the feed-forward operation.
- TransformerEncoderLayer: layer normalization is applied to the output of the self-attention operation.
- TransformerDecoderLayer: layer normalization is applied to the output of both the self-attention and cross-attention operations.

See the code below for the highlighted instances of layer normalization with comments in the footnotes.

```
class PositionwiseFeedForward(nn.Module):
    """
    Position-wise Feed-forward layer
    Projects to ff_size and then back down to input_size.
    """

    def __init__(
        self,
        input_size: int,
        ff_size: int,
        dropout: float = 0.1,
        alpha: float = 1.0,
        layer_norm: str = "post",[1]
        activation: str = "relu",
    ) -> None:
        """
        Initializes position-wise feed-forward layer.
        :param input_size: dimensionality of the input.
        :param ff_size: dimensionality of intermediate representation
        :param dropout: dropout probability
        :param alpha: weight factor for residual connection
        :param layer_norm: either "pre" or "post"[2]
        :param activation: activation function
        """
        super().__init__()

        activation_fnc = build_activation(activation=activation)
```

[1] “Post” means layer normalization is applied after the sublayer as a post-processing step
[2] “Pre” would be a pre-processing step, in the sense that it would be inserted before further layer calculations

```
        self.layer_norm = nn.LayerNorm(input_size, eps=1e-6)[3]
        self.pwff_layer = nn.Sequential(
            nn.Linear(input_size, ff_size),
            activation_fnc(),
            nn.Dropout(dropout),
            nn.Linear(ff_size, input_size),
            nn.Dropout(dropout),
        )

        self.alpha = alpha
        self._layer_norm_position = layer_norm
        assert self._layer_norm_position in {"pre", "post"}

    def forward(self, x: Tensor) -> Tensor:
        residual = x
        if self._layer_norm_position == "pre":
            x = self.layer_norm(x)[4]

        x = self.pwff_layer(x) + self.alpha * residual

        if self._layer_norm_position == "post":
            x = self.layer_norm(x)[5]
        return x


class PositionalEncoding(nn.Module):
    """
    Pre-compute position encodings (PE).
    In forward pass, this adds the position-encodings to the input for as many time
    steps as necessary.

    Implementation based on OpenNMT-py.
    https://github.com/OpenNMT/OpenNMT-py
    """

    def __init__(self, size: int = 0, max_len: int = 5000) -> None:
        """
        Positional Encoding with maximum length

        :param size: embeddings dimension size
```

[3] Here, the parameters of the standard layer normalization method from PyTorch nn module are defined
[4] Here, the instance self.layer_norm is used *before* the position-wise feed-forward network
[5] Here, the instance self.layer_norm is used *after* the position-wise feed-forward network

```
        :param max_len: maximum sequence length
        """
        if size % 2 != 0:
            raise ValueError(
                f"Cannot use sin/cos positional encoding with odd dim (got dim={size})")
        pe = torch.zeros(max_len, size)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(
            (torch.arange(0, size, 2, dtype=torch.float) * -(math.log(10000.0) / size)))
        pe[:, 0::2] = torch.sin(position.float() * div_term)
        pe[:, 1::2] = torch.cos(position.float() * div_term)
        pe = pe.unsqueeze(0)  # shape: (1, max_len, size)
        super().__init__()[6]
        self.register_buffer("pe", pe)
        self.dim = size

    def forward(self, emb: Tensor) -> Tensor:
        """
        Embed inputs.

        :param emb: (Tensor) Sequence of word embeddings vectors
            shape (seq_len, batch_size, dim)
        :return: positionally encoded word embeddings
        """
        # Add position encodings
        return emb + self.pe[:, :emb.size(1)]


class TransformerEncoderLayer(nn.Module):
    """
    One Transformer encoder layer has a Multi-head attention layer plus a position-wise
    feed-forward layer.
    """

    def __init__(
        self,
        size: int = 0,
        ff_size: int = 0,
        num_heads: int = 0,
        dropout: float = 0.1,
        alpha: float = 1.0,
        layer_norm: str = "post",[7]
```

[6] initializes the parent class
[7] Define default

```
        activation: str = "relu",
    ) -> None:
        """
        A single Transformer encoder layer.

        Note: don't change the name or the order of members!
        otherwise pretrained models cannot be loaded correctly.

        :param size: model dimensionality
        :param ff_size: size of the feed-forward intermediate layer
        :param num_heads: number of heads
        :param dropout: dropout to apply to input
        :param alpha: weight factor for residual connection
        :param layer_norm: either "pre" or "post"
        :param activation: activation function
        """
        super().__init__()

        self.layer_norm = nn.LayerNorm(size, eps=1e-6)[8]
        self.src_src_att = MultiHeadedAttention(num_heads, size, dropout=dropout)

        self.feed_forward = PositionwiseFeedForward(
            size,
            ff_size=ff_size,
            dropout=dropout,
            alpha=alpha,
            layer_norm=layer_norm,
            activation=activation,
        )

        self.dropout = nn.Dropout(dropout)
        self.size = size

        self.alpha = alpha
        self._layer_norm_position = layer_norm
        assert self._layer_norm_position in {"pre", "post"}

    def forward(self, x: Tensor, mask: Tensor) -> Tensor:
        """
        Forward pass for a single transformer encoder layer.
        First applies self attention, then dropout with residual connection (adding
        the input to the result), then layer norm, and then a position-wise
        feed-forward layer.
```

[8] Here, the parameters of the standard layer normalization method from PyTorch nn module are defined

```
        :param x: layer input
        :param mask: input mask
        :return: output tensor
        """
        residual = x
        if self._layer_norm_position == "pre":
            x = self.layer_norm(x)[9]

        x, _ = self.src_src_att(x, x, x, mask)
        x = self.dropout(x) + self.alpha * residual

        if self._layer_norm_position == "post":
            x = self.layer_norm(x)[10]

        out = self.feed_forward(x)
        return out


class TransformerDecoderLayer(nn.Module):
    """
    Transformer decoder layer.

    Consists of self-attention, source-attention, and feed-forward.
    """

    def __init__(
        self,
        size: int = 0,
        ff_size: int = 0,
        num_heads: int = 0,
        dropout: float = 0.1,
        alpha: float = 1.0,
        layer_norm: str = "post",[11]
        activation: str = "relu",
    ) -> None:
        """
        Represents a single Transformer decoder layer.
        It attends to the source representation and the previous decoder states.

        Note: don't change the name or the order of members!
```

[9] Layer normalization applied *before* the position-wise ff layer
[10] Layer normalization applied *after* the position-wise ff layer
[11] Define default

```
        otherwise pretrained models cannot be loaded correctly.

        :param size: model dimensionality
        :param ff_size: size of the feed-forward intermediate layer
        :param num_heads: number of heads
        :param dropout: dropout to apply to input
        :param alpha: weight factor for residual connection
        :param layer_norm: either "pre" or "post"
        :param activation: activation function
        """
        super().__init__()
        self.size = size

        self.trg_trg_att = MultiHeadedAttention(num_heads, size, dropout=dropout)
        self.src_trg_att = MultiHeadedAttention(num_heads, size, dropout=dropout)

        self.feed_forward = PositionwiseFeedForward(
            size,
            ff_size=ff_size,
            dropout=dropout,
            alpha=alpha,
            layer_norm=layer_norm,
            activation=activation,
        )

        self.x_layer_norm = nn.LayerNorm(size, eps=1e-6)[12]
        self.dec_layer_norm = nn.LayerNorm(size, eps=1e-6)[13]

        self.dropout = nn.Dropout(dropout)
        self.alpha = alpha

        self._layer_norm_position = layer_norm
        assert self._layer_norm_position in {"pre", "post"}

    def forward(
        self,
        x: Tensor,
        memory: Tensor,
        src_mask: Tensor,
        trg_mask: Tensor,
        return_attention: bool = False,
```

[12] an instance of layer normalization applied to the input x before the first attention block
[13] an instance of layer normalization applied to the output of the first attention block (h1) before the second attention block

```
    **kwargs,
) -> Tensor:
    """
    Forward pass of a single Transformer decoder layer.

    First applies target-target self-attention, dropout with residual connection
    (adding the input to the result), and layer norm.

    Second computes source-target cross-attention, dropout with residual connection
    (adding the self-attention to the result), and layer norm.

    Finally goes through a position-wise feed-forward layer.

    :param x: inputs
    :param memory: source representations
    :param src_mask: source mask
    :param trg_mask: target mask (so as to not condition on future steps)
    :param return_attention: whether to return the attention weights
    :return:
        - output tensor
        - attention weights
    """
    # pylint: disable=unused-argument
    # 1. target-target self-attention
    residual = x
    if self._layer_norm_position == "pre":
        x = self.x_layer_norm(x)[14]

    h1, _ = self.trg_trg_att(x, x, x, mask=trg_mask)
    h1 = self.dropout(h1) + self.alpha * residual

    if self._layer_norm_position == "post":
        h1 = self.x_layer_norm(h1)[15]

    # 2. source-target cross-attention
    h1_residual = h1
    if self._layer_norm_position == "pre":
        h1 = self.dec_layer_norm(h1)[16]

    h2, att = self.src_trg_att(memory,
```

[14] determines the position of the layer normalization in the encoder layer. If self._layer_norm_position is set to "pre" → layer normalization is used before the attention block.
[15] Here it is set to "post" → layer normalization is applied after the first attention block
[16] Same as above: here layer normalization is applied before the cross-attention operation

```
                            memory,
                            h1,
                            mask=src_mask,
                            return_weights=return_attention)
        h2 = self.dropout(h2) + self.alpha * h1_residual

        if self._layer_norm_position == "post":
            h2 = self.dec_layer_norm(h2)[17]

        # 3. final position-wise feed-forward layer
        out = self.feed_forward(h2)

        if return_attention:
            return out, att
        return out, None
```

[17] self._layer_norm_position is set to "pre" → layer normalization is applied before the cross-attention operation

2. Implementing Pre- and Post-Normalization

Preparations

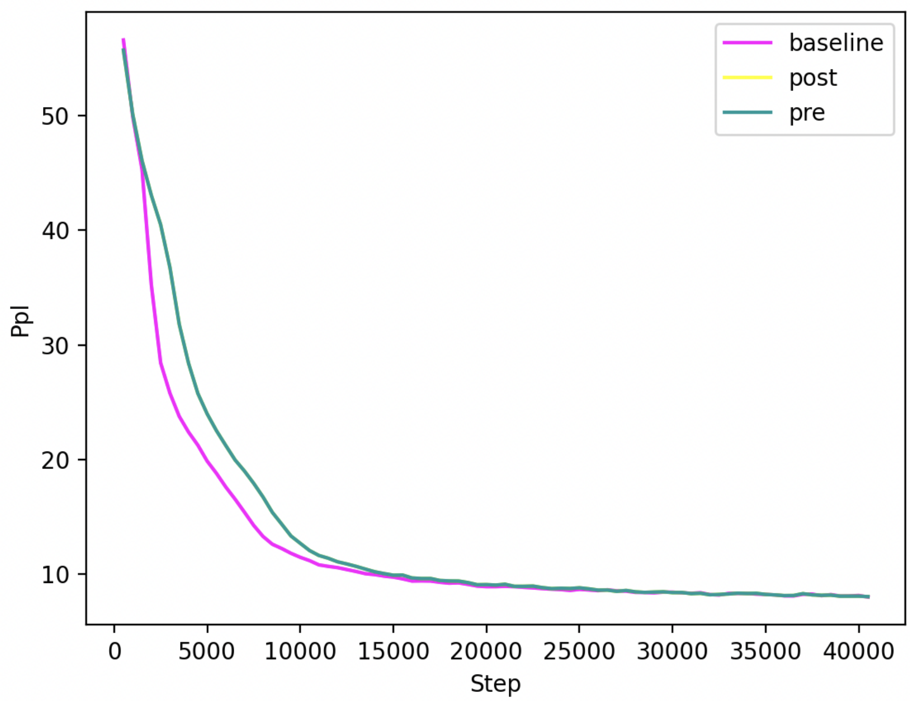
Please specify which repository you are referring to, since we are working in two here. For example, you recommend to refer to **joeynmt/scripts/training.py**, but the file does not exist. There is only **joeynmt/joeynmt/training.py**.

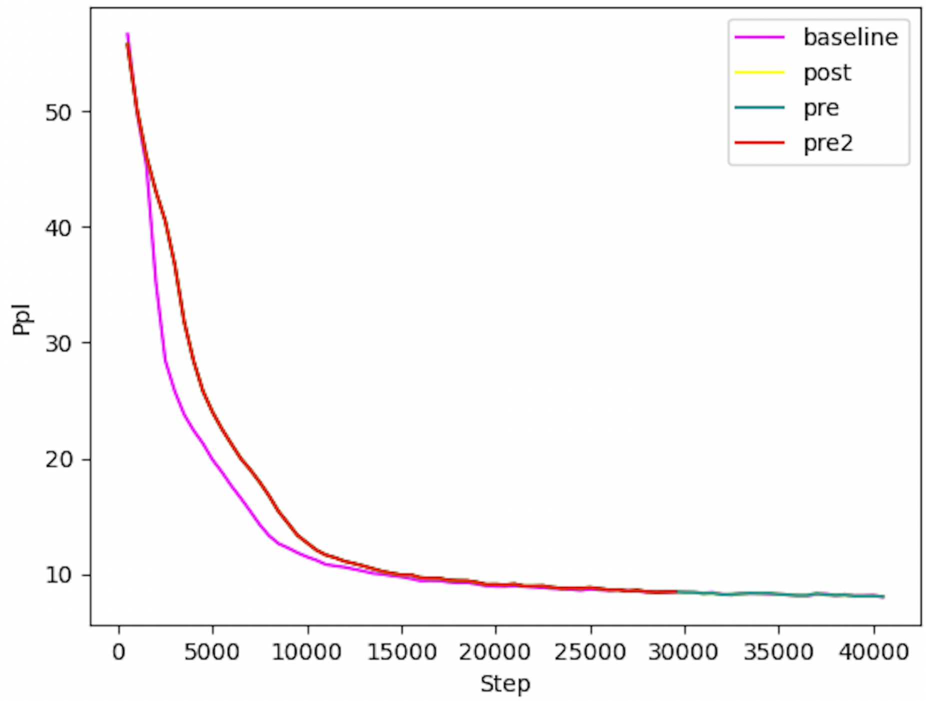
Also, how can I deal with the proposed modifications if cuda seems to no longer be supported for MacOS? I am really lost and it seems this is impossible for MacOS. I have tried to find a solution for my OS, but have not found a way to do it without spending too much time. Actually, I regretted it later, because training took ages, and a couple of hours invested in

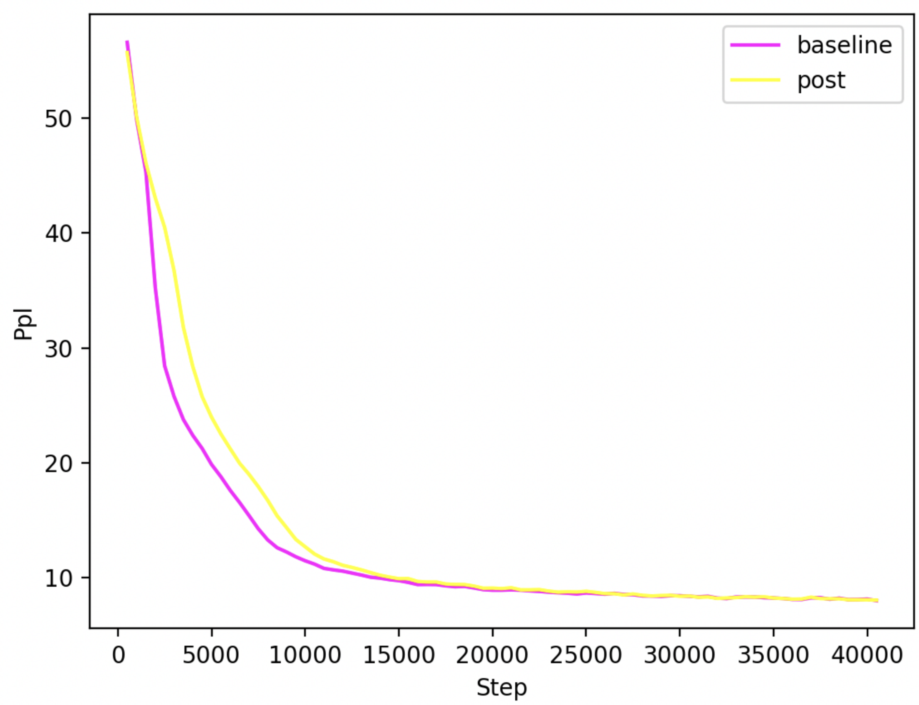
I had huge problems with incompatible packages at first, so I spent half a day trying to resolve inconsistencies manually (smth about flat namespace and lxml). In the end, the advice I found on Stack Overflow was to create a conda environment, and it finally worked and I could proceed with training.

Visualization

mt-exercise-4/perplexities/create_table.py







It is clearly visible that my “pre” and “post” results are identical, and the lines overlap completely.

Perplexity scores

When I saw that the results of post.log are identical with those achieved by training the model with “pre” layer normalization, I assumed I forgot to adapt the model accordingly in the **transformer_layers.py** file. Now that I repeated the process (partially, I interrupted the training in the middle, see the padded column pre2.log), I see that this is not the case, so the problem must lie somewhere else.

Step	baseline.log	pre.log	post.log	pre2.log
500	56.61	55.72	55.72	55.72
1000	49.93	50.13	50.13	50.13
1500	45.33	46.09	46.09	46.09
2000	35.25	43.06	43.06	43.06
2500	28.44	40.5	40.5	40.5
3000	25.79	36.75	36.75	36.75
3500	23.77	31.8	31.8	31.8
4000	22.4	28.4	28.4	28.4
4500	21.26	25.77	25.77	25.77
5000	19.87	23.99	23.99	23.99
5500	18.8	22.52	22.52	22.52
6000	17.61	21.23	21.23	21.23
6500	16.55	19.97	19.97	19.97
7000	15.42	19.01	19.01	19.01
7500	14.26	17.93	17.93	17.93
8000	13.3	16.74	16.74	16.74
8500	12.62	15.4	15.4	15.4
9000	12.25	14.38	14.38	14.38
9500	11.83	13.34	13.34	13.34
10000	11.48	12.7	12.7	12.7
10500	11.19	12.07	12.07	12.07
11000	10.82	11.64	11.64	11.64
11500	10.69	11.4	11.4	11.4
12000	10.58	11.09	11.09	11.09
12500	10.41	10.9	10.9	10.9
13000	10.24	10.69	10.69	10.69

13500	10.05	10.45	10.45	10.45
14000	9.97	10.22	10.22	10.22
14500	9.84	10.06	10.06	10.06
15000	9.75	9.92	9.92	9.92
15500	9.6	9.93	9.93	9.93
16000	9.41	9.68	9.68	9.68
16500	9.42	9.63	9.63	9.63
17000	9.41	9.63	9.63	9.63
17500	9.3	9.46	9.46	9.46
18000	9.22	9.42	9.42	9.42
18500	9.25	9.41	9.41	9.41
19000	9.12	9.28	9.28	9.28
19500	8.97	9.09	9.09	9.09
20000	8.92	9.1	9.1	9.1
20500	8.92	9.06	9.06	9.06
21000	8.95	9.13	9.13	9.13
21500	8.92	8.95	8.95	8.95
22000	8.86	8.95	8.95	8.95
22500	8.81	8.97	8.97	8.97
23000	8.75	8.84	8.84	8.84
23500	8.7	8.75	8.75	8.75
24000	8.66	8.78	8.78	8.78
24500	8.59	8.76	8.76	8.76
25000	8.68	8.82	8.82	8.82
25500	8.63	8.73	8.73	8.73
26000	8.57	8.62	8.62	8.62
26500	8.65	8.62	8.62	8.62
27000	8.55	8.51	8.51	8.51
27500	8.53	8.59	8.59	8.59
28000	8.42	8.47	8.47	8.47
28500	8.41	8.42	8.42	8.42

29000	8.38	8.45	8.45	8.45
29500	8.44	8.47	8.47	8.47
30000	8.44	8.4	8.4	
30500	8.39	8.41	8.41	
31000	8.34	8.31	8.31	
31500	8.39	8.34	8.34	
32000	8.23	8.21	8.21	
32500	8.19	8.24	8.24	
33000	8.33	8.28	8.28	
33500	8.33	8.35	8.35	
34000	8.33	8.32	8.32	
34500	8.27	8.34	8.34	
35000	8.24	8.24	8.24	
35500	8.2	8.2	8.2	
36000	8.12	8.14	8.14	
36500	8.11	8.15	8.15	
37000	8.25	8.31	8.31	
37500	8.27	8.2	8.2	
38000	8.13	8.17	8.17	
38500	8.23	8.18	8.18	
39000	8.1	8.09	8.09	
39500	8.11	8.09	8.09	
40000	8.15	8.09	8.09	
40500	8.01	8.05	8.05	

Discussion

• Given that there is a difference in the training progress for the three models, can you think of a reason for it?

As we have seen in the proposed papers, different layer normalization techniques produce different results: e.g. they cause the models to converge at different speeds, because the interactions between different layers and the flow of information are different. In our particular case

• In what way does our setup differ from Wang et. al. 2019? How could that have influenced our results?

In our model:
TransformerEncoderLayer:

- Layer normalization is applied either before or after the self-attention and feed-forward layers.

TransformerDecoderLayer:

- x_layer_norm is applied either before or after the self-attention layer
- dec_layer_norm is applied either before or after the cross-attention layer and feed-forward layer.

In the paper by Wang et al. (2019):[18]
In the encoder layers:

- Layer normalization is applied before both the self-attention and feed-forward layers → directly to the input of each sub-layer.

In the decoder layers:

- layer normalization is applied before both the self-attention and feed-forward layers.

[18] https://arxiv.org/pdf/1906.01787.pdf