View code on Github

Transformer Encoder and Decoder Models


```
import math

import torch
import torch.nn as nn

from labml_nn.utils import clone_module_list
from .feed_forward import FeedForward
from .mha import MultiHeadAttention
from .positional_encoding import get_positional_encoding
```

Embed tokens and add fixed positional encoding

```
class EmbeddingsWithPositionalEncoding(nn.Module):
```

```
    def __init__(self, d_model: int, n_vocab: int, max_len: int = 5000):
        super().__init__()
        self.linear = nn.Embedding(n_vocab, d_model)
        self.d_model = d_model
        self.register_buffer('positional_encodings', get_positional_encoding(d_model, max_len))
```

```
    def forward(self, x: torch.Tensor):
        pe = self.positional_encodings[:x.shape[0]].requires_grad_(False)
        return self.linear(x) * math.sqrt(self.d_model) + pe
```

Embed tokens and add parameterized positional encodings

```
class EmbeddingsWithLearnedPositionalEncoding(nn.Module):
```

```
    def __init__(self, d_model: int, n_vocab: int, max_len: int = 5000):
        super().__init__()
        self.linear = nn.Embedding(n_vocab, d_model)
        self.d_model = d_model
        self.positional_encodings = nn.Parameter(torch.zeros(max_len, 1, d_model), requires_grad=True)
```

```
    def forward(self, x: torch.Tensor):
        pe = self.positional_encodings[:x.shape[0]]
        return self.linear(x) * math.sqrt(self.d_model) + pe
```

Transformer Layer

This can act as an encoder layer or a decoder layer.

Some implementations, including the paper seem to have differences in where the layer-normalization is done. Here we do a layer normalization before attention and feed-forward networks, and add the original residual vectors. Alternative is to do a layer normalization after adding the residuals. But we found this to be less stable when training. We found a detailed discussion about this in the paper On Layer Normalization in the Transformer Architecture.

```
class TransformerLayer(nn.Module):
```

- `d_model` is the token embedding size
- `self_attn` is the self attention module
- `src_attn` is the source attention module (when this is used in a decoder)
- `feed_forward` is the feed forward module
- `dropout_prob` is the probability of dropping out after self attention and FFN

```
    def __init__(self, *,
                 d_model: int,
                 self_attn: MultiHeadAttention,
                 src_attn: MultiHeadAttention = None,
                 feed_forward: FeedForward,
                 dropout_prob: float):
```

```python
        super().__init__()
        self.size = d_model
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.dropout = nn.Dropout(dropout_prob)
        self.norm_self_attn = nn.LayerNorm([d_model])
        if self.src_attn is not None:
            self.norm_src_attn = nn.LayerNorm([d_model])
        self.norm_ff = nn.LayerNorm([d_model])
```

Whether to save input to the feed forward layer

```python
        self.is_save_ff_input = False
```

```python
    def forward(self, *,
                x: torch.Tensor,
                mask: torch.Tensor,
                src: torch.Tensor = None,
                src_mask: torch.Tensor = None):
```

Normalize the vectors before doing self attention

```python
        z = self.norm_self_attn(x)
```

Run through self attention, i.e. keys and values are from self

```python
        self_attn = self.self_attn(query=z, key=z, value=z, mask=mask)
```

Add the self attention results

```python
        x = x + self.dropout(self_attn)
```

If a source is provided, get results from attention to source. This is when you have a decoder layer that pays attention to encoder outputs

```python
        if src is not None:
```

Normalize vectors

```python
            z = self.norm_src_attn(x)
```

Attention to source. i.e. keys and values are from source

```python
            attn_src = self.src_attn(query=z, key=src, value=src, mask=src_mask)
```

Add the source attention results

```python
            x = x + self.dropout(attn_src)
```

Normalize for feed-forward

```python
        z = self.norm_ff(x)
```

Save the input to the feed forward layer if specified

```python
        if self.is_save_ff_input:
            self.ff_input = z.clone()
```

Pass through the feed-forward network

```python
        ff = self.feed_forward(z)
```

Add the feed-forward results back

```python
        x = x + self.dropout(ff)

        return x
```

Transformer Encoder

```python
class Encoder(nn.Module):
```

```python
    def __init__(self, layer: TransformerLayer, n_layers: int):
        super().__init__()
```

Make copies of the transformer layer

```python
        self.layers = clone_module_list(layer, n_layers)
```

Final normalization layer

```python
        self.norm = nn.LayerNorm([layer.size])
```

```python
    def forward(self, x: torch.Tensor, mask: torch.Tensor):
```

Run through each transformer layer

```python
        for layer in self.layers:
            x = layer(x=x, mask=mask)
```

Finally, normalize the vectors

```python
        return self.norm(x)
```

Transformer Decoder

```python
class Decoder(nn.Module):
```

```python
    def __init__(self, layer: TransformerLayer, n_layers: int):
        super().__init__()
```

Make copies of the transformer layer

```python
        self.layers = clone_module_list(layer, n_layers)
```

Final normalization layer

```python
        self.norm = nn.LayerNorm([layer.size])
```

```python
    def forward(self, x: torch.Tensor, memory: torch.Tensor, src_mask: torch.Tensor, tgt_mask: torch.Tensor):
```

Run through each transformer layer

```python
        for layer in self.layers:
            x = layer(x=x, mask=tgt_mask, src=memory, src_mask=src_mask)
```

Finally, normalize the vectors

```python
        return self.norm(x)
```

Generator

This predicts the tokens and gives the lof softmax of those. You don't need this if you are using `nn.CrossEntropyLoss`.

```python
class Generator(nn.Module):
```

```python
    def __init__(self, n_vocab: int, d_model: int):
        super().__init__()
        self.projection = nn.Linear(d_model, n_vocab)
```

```python
    def forward(self, x):
        return self.projection(x)
```

Combined Encoder-Decoder

```python
class EncoderDecoder(nn.Module):
```

```python
    def __init__(self, encoder: Encoder, decoder: Decoder, src_embed: nn.Module, tgt_embed: nn.Module, generator: nn.Module):
        super().__init__()
```

```python
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator
```

This was important from their code. Initialize parameters with Glorot / fan_avg.

```python
        for p in self.parameters():
            if p.dim() > 1:
                nn.init.xavier_uniform_(p)
```

```python
    def forward(self, src: torch.Tensor, tgt: torch.Tensor, src_mask: torch.Tensor, tgt_mask: torch.Tensor):
```

Run the source through encoder

```python
        enc = self.encode(src, src_mask)
```

Run encodings and targets through decoder

```python
        return self.decode(enc, src_mask, tgt, tgt_mask)
```

```python
    def encode(self, src: torch.Tensor, src_mask: torch.Tensor):
        return self.encoder(self.src_embed(src), src_mask)
```

```python
    def decode(self, memory: torch.Tensor, src_mask: torch.Tensor, tgt: torch.Tensor, tgt_mask: torch.Tensor):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)
```