# nlp-practical-four

April 11, 2025

[ ]:
```
# This Python 3 environment comes with many helpful analytics libraries␣
↪installed
# It is defined by the kaggle/python Docker image: https://github.com/kaggle/
↪docker-python
# For example, here's several helpful packages to load

import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

# Input data files are available in the read-only "../input/" directory
# For example, running this (by clicking run or pressing Shift+Enter) will list␣
↪all files under the input directory

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

# You can write up to 20GB to the current directory (/kaggle/working/) that␣
↪gets preserved as output when you create a version using "Save & Run All"
# You can also write temporary files to /kaggle/temp/, but they won't be saved␣
↪outside of the current session
```

[1]:
```
import torch
import torch.nn as nn
import math
import copy
```

[4]:
```
class InputEmbeddings(nn.Module):
    def __init__(self, d_model: int, vocab_size: int):
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x):
        # Multiply by sqrt(d_model) as per the paper
```

```
        return self.embedding(x) * math.sqrt(self.d_model)
```

[5]:
```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model: int, dropout: float, max_len: int = 5000):
        super().__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, dtype=torch.float).unsqueeze(1) #␣
↪Shape: (max_len, 1)
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.
↪log(10000.0) / d_model)) # Shape: (d_model/2)

        pe[:, 0::2] = torch.sin(position * div_term) # Apply sin to even indices
        pe[:, 1::2] = torch.cos(position * div_term) # Apply cos to odd indices

        pe = pe.unsqueeze(0) # Shape: (1, max_len, d_model) - Add batch␣
↪dimension

        # Register buffer makes it part of the model's state, but not a␣
↪parameter
        self.register_buffer('pe', pe)

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        # Add positional encoding to the input embeddings
        # self.pe[:, :x.size(1)] selects encodings up to the sequence length of␣
↪x
        x = x + self.pe[:, :x.size(1)].requires_grad_(False) # Don't train␣
↪positional encodings
        return self.dropout(x)
```

[6]:
```
class LayerNorm(nn.Module):
    def __init__(self, features: int, eps: float = 1e-6):
        super().__init__()
        self.eps = eps
        # Learnable parameters
        self.gamma = nn.Parameter(torch.ones(features)) # scale
        self.beta = nn.Parameter(torch.zeros(features)) # offset

    def forward(self, x):
        # x shape: (batch_size, seq_len, features)
        mean = x.mean(-1, keepdim=True) # Calculate mean over the last␣
↪dimension (features)
```

```
        std = x.std(-1, keepdim=True)   # Calculate std dev over the last␣
↪dimension
        # Normalize
        return self.gamma * (x - mean) / (std + self.eps) + self.beta
```

[7]:
```
class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_model: int, d_ff: int, dropout: float = 0.1):
        super().__init__()
        self.linear_1 = nn.Linear(d_model, d_ff) # W1
        self.dropout = nn.Dropout(dropout)
        self.linear_2 = nn.Linear(d_ff, d_model) # W2
        self.relu = nn.ReLU()

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        # FFN(x) = max(0, xW1 + b1)W2 + b2
        return self.linear_2(self.dropout(self.relu(self.linear_1(x))))
```

[8]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, d_model: int, num_heads: int, dropout: float = 0.1):
        super().__init__()
        assert d_model % num_heads == 0, "d_model must be divisible by␣
↪num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads # Dimension of keys/queries/values per␣
↪head

        # Linear layers for Query, Key, Value, and final output
        self.w_q = nn.Linear(d_model, d_model)
        self.w_k = nn.Linear(d_model, d_model)
        self.w_v = nn.Linear(d_model, d_model)
        self.w_o = nn.Linear(d_model, d_model) # Output transformation

        self.dropout = nn.Dropout(dropout)
        self.attention_scores = None # To store attention scores for␣
↪visualization if needed

    @staticmethod
    def attention(query, key, value, mask=None, dropout=None):
        d_k = query.size(-1)
        # scores shape: (batch_size, num_heads, seq_len_q, seq_len_k)
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)

        if mask is not None:
            # Apply mask (typically for padding or causal attention)
```

```
            # Mask should be broadcastable to scores shape
            scores = scores.masked_fill(mask == 0, -1e9) # Fill with very␣
↪small value

        # p_attn shape: (batch_size, num_heads, seq_len_q, seq_len_k)
        p_attn = torch.softmax(scores, dim=-1)

        if dropout is not None:
            p_attn = dropout(p_attn)

        # output shape: (batch_size, num_heads, seq_len_q, d_k)
        output = torch.matmul(p_attn, value)
        return output, p_attn # Return context vector and attention weights

    def forward(self, query, key, value, mask=None):
        # query, key, value shape: (batch_size, seq_len, d_model)
        batch_size = query.size(0)

        # 1) Apply linear projections and split into heads
        # q, k, v shape: (batch_size, num_heads, seq_len, d_k)
        q = self.w_q(query).view(batch_size, -1, self.num_heads, self.d_k).
↪transpose(1, 2)
        k = self.w_k(key).view(batch_size, -1, self.num_heads, self.d_k).
↪transpose(1, 2)
        v = self.w_v(value).view(batch_size, -1, self.num_heads, self.d_k).
↪transpose(1, 2)

        # 2) Apply attention on all heads in parallel
        # x shape: (batch_size, num_heads, seq_len_q, d_k)
        # self.attention_scores shape: (batch_size, num_heads, seq_len_q,␣
↪seq_len_k)
        x, self.attention_scores = MultiHeadAttention.attention(q, k, v,␣
↪mask=mask, dropout=self.dropout)

        # 3) Concatenate heads and apply final linear layer
        # x shape: (batch_size, seq_len_q, d_model)
        x = x.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
        x = self.w_o(x)

        return x
```

[9]:

```
class ResidualConnection(nn.Module):
    def __init__(self, d_model: int, dropout: float):
        super().__init__()
        self.norm = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)
```

```python
    def forward(self, x, sublayer):
        # Apply residual connection to the output of the sublayer
        # The sublayer is a function (like multi-head attention or feed-forward)
        # return x + self.dropout(sublayer(self.norm(x))) # Original paper: Norm is applied before sublayer
        # Post-LN variation (often performs well or better): Apply norm after adding residual
        return self.norm(x + self.dropout(sublayer(x)))
```

[10]:

```python
class EncoderLayer(nn.Module):
    def __init__(self, d_model: int, self_attn: MultiHeadAttention, feed_forward: PositionwiseFeedForward, dropout: float):
        super().__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.res_connect_1 = ResidualConnection(d_model, dropout)
        self.res_connect_2 = ResidualConnection(d_model, dropout)
        self.d_model = d_model

    def forward(self, x, mask):
        # x shape: (batch_size, seq_len, d_model)
        # mask shape: (batch_size, 1, 1, seq_len) or similar broadcastable shape

        # Sublayer 1: Multi-Head Self-Attention + Add & Norm
        # Pass x as query, key, and value for self-attention
        x = self.res_connect_1(x, lambda x: self.self_attn(x, x, x, mask))

        # Sublayer 2: Feed Forward + Add & Norm
        x = self.res_connect_2(x, self.feed_forward)
        return x
```

[11]:

```python
class Encoder(nn.Module):
    def __init__(self, layer: EncoderLayer, num_layers: int):
        super().__init__()
        # Create N identical layers
        self.layers = nn.ModuleList([copy.deepcopy(layer) for _ in range(num_layers)])
        self.norm = LayerNorm(layer.d_model) # Final normalization layer

    def forward(self, x, mask):
        # x shape: (batch_size, seq_len, d_model)
        # mask shape: Broadcastable to attention scores
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x) # Apply final normalization
```

[12]:
```
class DecoderLayer(nn.Module):
    def __init__(self, d_model: int, self_attn: MultiHeadAttention, cross_attn:␣
↪MultiHeadAttention, feed_forward: PositionwiseFeedForward, dropout: float):
        super().__init__()
        self.d_model = d_model
        self.self_attn = self_attn # Masked self-attention
        self.cross_attn = cross_attn # Cross-attention (query=decoder, key/
↪value=encoder output)
        self.feed_forward = feed_forward
        self.res_connect_1 = ResidualConnection(d_model, dropout)
        self.res_connect_2 = ResidualConnection(d_model, dropout)
        self.res_connect_3 = ResidualConnection(d_model, dropout)

    def forward(self, x, memory, src_mask, tgt_mask):
        # x shape (decoder input): (batch_size, tgt_seq_len, d_model)
        # memory shape (encoder output): (batch_size, src_seq_len, d_model)
        # src_mask: Masks encoder padding. Shape: (batch_size, 1, 1,␣
↪src_seq_len)
        # tgt_mask: Masks decoder padding and future tokens. Shape:␣
↪(batch_size, 1, tgt_seq_len, tgt_seq_len)

        # Sublayer 1: Masked Multi-Head Self-Attention + Add & Norm
        # Pass x as query, key, value; use target mask
        x = self.res_connect_1(x, lambda x: self.self_attn(x, x, x, tgt_mask))

        # Sublayer 2: Multi-Head Cross-Attention + Add & Norm
        # Query comes from decoder (x), Key/Value come from encoder output␣
↪(memory)
        # Use source mask here
        x = self.res_connect_2(x, lambda x: self.cross_attn(x, memory, memory,␣
↪src_mask))

        # Sublayer 3: Feed Forward + Add & Norm
        x = self.res_connect_3(x, self.feed_forward)
        return x
```

[13]:
```
class Decoder(nn.Module):
    def __init__(self, layer: DecoderLayer, num_layers: int):
        super().__init__()
        self.layers = nn.ModuleList([copy.deepcopy(layer) for _ in␣
↪range(num_layers)])
        self.norm = LayerNorm(layer.d_model) # Final normalization layer

    def forward(self, x, memory, src_mask, tgt_mask):
        # x shape (decoder input): (batch_size, tgt_seq_len, d_model)
        # memory shape (encoder output): (batch_size, src_seq_len, d_model)
```

```
        # src_mask/tgt_mask: Appropriate masks
        for layer in self.layers:
            x = layer(x, memory, src_mask, tgt_mask)
        return self.norm(x) # Apply final normalization
```

[14]:
```
class ProjectionLayer(nn.Module):
    def __init__(self, d_model: int, vocab_size: int):
        super().__init__()
        self.proj = nn.Linear(d_model, vocab_size)

    def forward(self, x):
        # x shape: (batch_size, seq_len, d_model)
        # Output shape: (batch_size, seq_len, vocab_size)
        # Log softmax is often used with NLLLoss for training
        return torch.log_softmax(self.proj(x), dim=-1)
```

[15]:
```
class Transformer(nn.Module):
    def __init__(self,
                 encoder: Encoder,
                 decoder: Decoder,
                 src_embed: InputEmbeddings,
                 tgt_embed: InputEmbeddings,
                 pos_enc: PositionalEncoding,
                 projection: ProjectionLayer):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.pos_enc = pos_enc # Use the same PE for src and tgt embeddings
        self.projection = projection

    def encode(self, src, src_mask):
        # Embed source sequence, add positional encoding, pass through encoder
        return self.encoder(self.pos_enc(self.src_embed(src)), src_mask)

    def decode(self, tgt, memory, src_mask, tgt_mask):
        # Embed target sequence, add positional encoding, pass through decoder
        return self.decoder(self.pos_enc(self.tgt_embed(tgt)), memory,␣
↪src_mask, tgt_mask)

    def project(self, x):
        # Project decoder output to vocabulary space
        return self.projection(x)

    def forward(self, src, tgt, src_mask, tgt_mask):
        # src: (batch_size, src_seq_len)
```

```
        # tgt: (batch_size, tgt_seq_len)
        # src_mask: (batch_size, 1, 1, src_seq_len) - Masks padding in src
        # tgt_mask: (batch_size, 1, tgt_seq_len, tgt_seq_len) - Masks padding and future tokens in tgt

        encoder_output = self.encode(src, src_mask)
        decoder_output = self.decode(tgt, encoder_output, src_mask, tgt_mask)
        output = self.project(decoder_output)
        return output # Shape: (batch_size, tgt_seq_len, tgt_vocab_size)
```

[16]:
```
def build_transformer(src_vocab_size: int, tgt_vocab_size: int,
                      d_model: int = 512, num_layers: int = 6, num_heads: int = 8,
                      d_ff: int = 2048, dropout: float = 0.1, max_len: int = 5000) -> Transformer:

    # Create embedding layers
    src_embed = InputEmbeddings(d_model, src_vocab_size)
    tgt_embed = InputEmbeddings(d_model, tgt_vocab_size)

    # Create positional encoding layer
    pos_enc = PositionalEncoding(d_model, dropout, max_len)

    # Create Multi-Head Attention, Feed Forward, and Encoder/Decoder layers
    attn = MultiHeadAttention(d_model, num_heads, dropout)
    ff = PositionwiseFeedForward(d_model, d_ff, dropout)
    encoder_layer = EncoderLayer(d_model, copy.deepcopy(attn), copy.deepcopy(ff), dropout)
    decoder_layer = DecoderLayer(d_model, copy.deepcopy(attn), copy.deepcopy(attn), copy.deepcopy(ff), dropout)

    # Create Encoder and Decoder stacks
    encoder = Encoder(encoder_layer, num_layers)
    decoder = Decoder(decoder_layer, num_layers)

    # Create projection layer
    projection = ProjectionLayer(d_model, tgt_vocab_size)

    # Assemble the Transformer model
    model = Transformer(encoder, decoder, src_embed, tgt_embed, pos_enc, projection)

    # Initialize parameters (Xavier/Glorot recommended in the paper)
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
```

```
    return model
```

[17]:
```
if __name__ == '__main__':
    print("--- Building and Testing Transformer ---")

    # Define parameters
    SRC_VOCAB_SIZE = 10000
    TGT_VOCAB_SIZE = 11000
    D_MODEL = 512
    NUM_LAYERS = 6
    NUM_HEADS = 8
    D_FF = 2048
    DROPOUT = 0.1
    MAX_LEN = 100 # Max sequence length for example

    # Build the model
    transformer_model = build_transformer(
        SRC_VOCAB_SIZE, TGT_VOCAB_SIZE, D_MODEL, NUM_LAYERS, NUM_HEADS, D_FF,␣
↪DROPOUT, MAX_LEN
    )
    print(f"Model built successfully. Parameter count: {sum(p.numel() for p in␣
↪transformer_model.parameters() if p.requires_grad):,}")

    # Create dummy input data
    BATCH_SIZE = 4
    SRC_SEQ_LEN = 10
    TGT_SEQ_LEN = 12

    # Dummy source and target sequences (indices)
    src_tokens = torch.randint(1, SRC_VOCAB_SIZE, (BATCH_SIZE, SRC_SEQ_LEN)) #␣
↪Avoid 0 for padding usually
    tgt_tokens = torch.randint(1, TGT_VOCAB_SIZE, (BATCH_SIZE, TGT_SEQ_LEN))

    # Create dummy masks
    # Source padding mask (mask where src_tokens == 0, assuming 0 is padding)
    src_padding_mask = (src_tokens != 0).unsqueeze(1).unsqueeze(2) # Shape: (B,␣
↪1, 1, S)
    # Target padding mask (mask where tgt_tokens == 0)
    tgt_padding_mask = (tgt_tokens != 0).unsqueeze(1).unsqueeze(2) # Shape: (B,␣
↪1, 1, T)
    # Target subsequent/causal mask (prevents attending to future tokens)
    tgt_seq_len = tgt_tokens.size(1)
    tgt_subsequent_mask = torch.tril(torch.ones((tgt_seq_len, tgt_seq_len),␣
↪dtype=torch.bool)).unsqueeze(0).unsqueeze(0) # Shape: (1, 1, T, T)
    # Combine target padding and subsequent masks
```

```
    tgt_combined_mask = tgt_padding_mask & tgt_subsequent_mask # Shape: (B, 1,␣
↪T, T)

    print(f"\nInput shapes:")
    print(f"src_tokens:        {src_tokens.shape}")
    print(f"tgt_tokens:        {tgt_tokens.shape}")
    print(f"src_padding_mask:  {src_padding_mask.shape}")
    print(f"tgt_combined_mask: {tgt_combined_mask.shape}")


    # Forward pass
    transformer_model.eval() # Set model to evaluation mode (disables dropout)
    with torch.no_grad(): # Disable gradient calculation for inference
        output = transformer_model(src_tokens, tgt_tokens, src_padding_mask,␣
↪tgt_combined_mask)

    print(f"\nOutput shape (log probabilities): {output.shape}") # Should be␣
↪(BATCH_SIZE, TGT_SEQ_LEN, TGT_VOCAB_SIZE)
    print("--- Example Completed ---")
```

```
--- Building and Testing Transformer ---
Model built successfully. Parameter count: 60,535,544

Input shapes:
src_tokens:        torch.Size([4, 10])
tgt_tokens:        torch.Size([4, 12])
src_padding_mask:  torch.Size([4, 1, 1, 10])
tgt_combined_mask: torch.Size([4, 1, 12, 12])

Output shape (log probabilities): torch.Size([4, 12, 11000])
--- Example Completed ---
```