Attention Is All You Need

September 30, 2023

1 Attention Is All You Need

1.1 Scaled Dot Product Attention

At its core, the self-attention mechanism revolves around the interplay of three components: **key**, **query**, and **value**. These are vital for understanding how information is weighted and propagated in attention models, such as the Transformer.

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

When $Q = K$, the term QK^T captures the self-attention, indicating how similar elements within the matrix Q are to one another.

1.1.1 Why Use $\sqrt{d_k}$?

Under the assumption that the components of q and k are independent random variables with mean 0 and variance 1 (it is quite theoretical assumption that is not realistic for most cases), their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$ has mean 0 and variance d_k.

The mean can be determined using the **linearity of expectation**:

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

$$= \sum_{i=1}^{d_k} E[q_i k_i]$$

Given the assumption that random variables are i.i.d (independently identically distributed):

$$= \sum_{i=1}^{d_k} E[q_i]E[k_i] = 0$$

Thus, the mean of $q \cdot k$ equals 0.

For variance, although variance is not strictly linear in the way that expectation is, in this context, since the random variables are independent, the variance of their sum is the sum of their variances. Hence, using a principle similar to the **linearity of expectation**:

$$\text{var}[q \cdot k] = \text{var}\left[\sum_{i=1}^{d_k} q_i k_i\right]$$

$$= \sum_{i=1}^{d_k} \text{var}[q_i k_i] = d_k$$

To make the dot product have a mean of 0 and standard deviation of 1, it's divided by $\sqrt{d_k}$. However, nowadays, this normalization is often omitted since a normal distribution is not always assumed, especially when layer normalization is not used. **Scaled Dot Product Attention** refers to the process of this calculation. Given that **Query**, **Key**, and **Value** are all 3×1 matrices:

$$Q = K = V = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

Since QK^T results in a 3×3 matrix:

$$QK^T = \begin{bmatrix} v_1 \cdot v_1 & v_1 \cdot v_2 & v_1 \cdot v_3 \\ v_2 \cdot v_1 & v_2 \cdot v_2 & v_2 \cdot v_3 \\ v_3 \cdot v_1 & v_3 \cdot v_2 & v_3 \cdot v_3 \end{bmatrix}$$

We then divide QK^T by $\sqrt{d_k}$, obtaining the **attention weight**:

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Given the value matrix, we compute:

$$\text{Softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \times V = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}$$

The attention mechanism gauges the similarity between a *query* (the word we're focusing on) and a *key* (the word we're comparing against). The resulting similarity scores are then used to weigh the importance of words in the **Value** matrix. See below example code to understand how it goes:

[1]:
```
# !python3 -m pip install torch torchtext
# !/Library/Developer/CommandLineTools/usr/bin/python3 -m pip install --upgrade␣
↪pip

import torch
```

```
import torch.nn as nn
import torch.nn.functional as F

class ScaledDotProductAttention(nn.Module):
    def __init__(self, temperature):
        super(ScaledDotProductAttention, self).__init__()
        self.temperature = temperature
        self.softmax = nn.Softmax(dim=2)

    def forward(self, q, k, v, mask=None):
        attn = torch.bmm(q, k.transpose(1, 2))
        attn = attn / self.temperature

        if mask is not None:
            attn = attn.masked_fill(mask, -float('inf'))

        attn = self.softmax(attn)
        output = torch.bmm(attn, v)
        return output, attn
```

1.2 MultiHeadAttention & PositionWiseFeedForward

Now, it is the moment to introduce **MultiHeadAttention** – where multiple **ScaledDotProductAttention** modules are applied.

[2]:
```
class MultiHeadAttention(nn.Module):
    def __init__(self, n_head, d_model, d_k, d_v, dropout=0.1):
        super(MultiHeadAttention, self).__init__()
        self.n_head = n_head
        self.d_k = d_k
        self.d_v = d_v

        self.w_qs = nn.Linear(d_model, d_k * n_head, bias=False)
        self.w_ks = nn.Linear(d_model, d_k * n_head, bias=False)
        self.w_vs = nn.Linear(d_model, d_v * n_head, bias=False)
        self.fc = nn.Linear(n_head * d_v, d_model)
        self.attention = ScaledDotProductAttention(temperature=torch.sqrt(torch.
 ↪tensor(d_k).float()))

        self.dropout = nn.Dropout(dropout)
        self.layer_norm = nn.LayerNorm(d_model)

    def forward(self, q, k, v, mask=None):
        d_k, d_v, n_head = self.d_k, self.d_v, self.n_head
        sz_b, len_q = q.size(0), q.size(1)

        residual = q  # Store the residual connection
```

```
        q = self.w_qs(q).view(sz_b, len_q, n_head, d_k)
        k = self.w_ks(k).view(sz_b, len_q, n_head, d_k)
        v = self.w_vs(v).view(sz_b, len_q, n_head, d_v)

        q = q.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k)
        k = k.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k)
        v = v.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_v)

        if mask is not None:
            mask = mask.repeat(n_head, 1, 1)
        output, attn = self.attention(q, k, v, mask=mask)

        output = output.view(n_head, sz_b, len_q, d_v)
        output = output.permute(1, 2, 0, 3).contiguous().view(sz_b, len_q, -1)
        output = self.dropout(self.fc(output))
        output = self.layer_norm(output + residual)
        return output, attn


class PositionwiseFeedForward(nn.Module):
    def __init__(self, d_in, d_hid, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_in, d_hid)
        self.w_2 = nn.Linear(d_hid, d_in)
        self.layer_norm = nn.LayerNorm(d_in)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        residual = x
        x = self.w_2(F.relu(self.w_1(x)))
        x = self.dropout(x)
        x = self.layer_norm(x + residual)
        return x
```

1.2.1 Transformer Encoder

[3]:
```
class TransformerEncoderLayer(nn.Module):
    def __init__(self, d_model, d_k, d_v, d_ff, n_head, dropout=0.1):
        super(TransformerEncoderLayer, self).__init__()
        self.self_attn = MultiHeadAttention(n_head, d_model, d_k, d_v,␣
 ↪dropout=dropout)
        self.pos_ffn = PositionwiseFeedForward(d_model, d_ff, dropout=dropout)

    def forward(self, src, mask=None):
        enc_output, attn = self.self_attn(src, src, src, mask=mask)
        enc_output = self.pos_ffn(enc_output)
```

```
        return enc_output, attn

class TransformerEncoder(nn.Module):
    def __init__(self, d_model, d_k, d_v, d_ff, n_head, n_layers, dropout=0.1):
        super(TransformerEncoder, self).__init__()
        self.layers = nn.ModuleList([
            TransformerEncoderLayer(d_model, d_k, d_v, d_ff, n_head,␣
 ↪dropout=dropout)
            for _ in range(n_layers)
        ])

    def forward(self, src, mask=None):
        attns = []
        for layer in self.layers:
            src, attn = layer(src, mask=mask)
            attns.append(attn)
        return src, attns
```

1.2.2 Sample Code to Run

[4]:
```
# Assuming you have all the previous classes defined as provided
import torch
from torchtext.vocab import build_vocab_from_iterator
from torchtext.data.utils import get_tokenizer

# 1. Tokenization and Vocabulary
tokenizer = get_tokenizer('basic_english')

def yield_tokens(data_iter):
    for sentence in data_iter:
        yield tokenizer(sentence)

sentences = ["Hello World"]
vocab = build_vocab_from_iterator(yield_tokens(sentences), specials=['<unk>',␣
 ↪'<pad>', '<bos>', '<eos>'])
vocab.set_default_index(vocab["<unk>"])

tokenized_sentence = [vocab[token] for token in tokenizer("Hello World")]
sentence_tensor = torch.tensor(tokenized_sentence, dtype=torch.long).
 ↪unsqueeze(0)  # (1, len(sentence))

# 2. Embedding
embedding_dim = 512
embedding = torch.nn.Embedding(len(vocab), embedding_dim)
embedded_sentence = embedding(sentence_tensor)

# 3. Encoding with TransformerEncoder
```

```
# Assuming d_model=512, d_k=d_v=64, d_ff=2048, n_head=8, n_layers=6
encoder = TransformerEncoder(d_model=512, d_k=64, d_v=64, d_ff=2048, n_head=8,␣
↪n_layers=6)
enc_output, attns = encoder(embedded_sentence)

print("Encoder Output Shape:", enc_output.shape)
```

```
Encoder Output Shape: torch.Size([1, 2, 512])
```

[5]:
```
embedded_sentence.shape, enc_output.shape
```

[5]:
```
(torch.Size([1, 2, 512]), torch.Size([1, 2, 512]))
```

1.2.3 Into PDF

[6]:
```
from base_lib import convert_ipynb_to_pdf

convert_ipynb_to_pdf('./Attention Is All You Need.ipynb')
```