NeuralNets

April 25, 2025

0.0.1 Transformers Architecture

Transformers is the standard architecture upon which most LLMs are based—the decoder-only transformer architecture. This architecture is a modified version of the encoder-decoder transformer architecture that was popularized by GPT.

While explaining the architecture, we will rely on Andrej Karpathy's nanoGPT—a minimal and functional implementation of decoder-only transformers—as a reference.

[1] https://github.com/karpathy/nanoGPT

[2] https://github.com/wolfecameron/nanoMoE , https://cameronrwolfe.substack.com/p/nano-moe

[3] https://jalammar.github.io/illustrated-transformer/

[4] https://poloclub.github.io/transformer-explainer/

Encoder The encoder's inputs first flow through a self-attention layer – a layer that helps the encoder look at other words in the input sentence as it encodes a specific word. The outputs of the self-attention layer are fed to a feed-forward neural network.

We begin by turning each input word in the sequence into a vector using an embedding algorithm.

The embedding only happens in the bottom-most encoder. The abstraction that is common to all the encoders is that they receive a list of vectors each of the size 512 – In the bottom encoder that would be the word embeddings, but in other encoders, it would be the output of the encoder that's directly below. The size of this list is hyperparameter we can set – basically it would be the length of the longest sentence in our training dataset.

After embedding the words in our input sequence, each of them flows through each of the two layers of the encoder.

Self-Attention Say the following sentence is an input sentence we want to translate:

"The animal didn't cross the street because it was too tired"

What does "it" in this sentence refer to? Is it referring to the street or to the animal? It's a simple question to a human, but not as simple to an algorithm.

When the model is processing the word "it", self-attention allows it to associate "it" with "animal".

As the model processes each word (each position in the input sequence), self attention allows it to look at other positions in the input sequence for clues that can help lead to a better encoding for this word.

Self-Attention in Detail The **first step** in calculating self-attention is to create three vectors from each of the encoder's input vectors (in this case, the embedding of each word). So for each word, we create a Query vector, a Key vector, and a Value vector. These vectors are created by multiplying the embedding by three matrices that we trained during the training process.

Notice that these new vectors are smaller in dimension than the embedding vector. Their dimensionality is 64, while the embedding and encoder input/output vectors have dimensionality of 512. They don't HAVE to be smaller, this is an architecture choice to make the computation of multiheaded attention (mostly) constant.

Multiplying `x1` by the `WQ` weight matrix produces `q1`, the "query" vector associated with that word. We end up creating a "query", a "key", and a "value" projection of each word in the input sentence.

The **second step** in calculating self-attention is to calculate a score. Say we're calculating the self-attention for the first word in this example, "Thinking". We need to score each word of the input sentence against this word. The score determines how much focus to place on other parts of the input sentence as we encode a word at a certain position.

The score is calculated by taking the dot product of the query vector with the key vector of the respective word we're scoring. So if we're processing the self-attention for the word in position #1, the first score would be the dot product of `q1` and `k1`. The second score would be the dot product of `q1` and `k2`.

The **third and fourth steps** are to divide the scores by 8 (the square root of the dimension of the key vectors used in the paper – 64. This leads to having more stable gradients. There could be other possible values here, but this is the default), then pass the result through a softmax operation. Softmax normalizes the scores so they're all positive and add up to 1.

This softmax score determines how much each word will be expressed at this position. Clearly the word at this position will have the highest softmax score, but sometimes it's useful to attend to another word that is relevant to the current word.

The **fifth step** is to multiply each value vector by the softmax score (in preparation to sum them up). The intuition here is to keep intact the values of the word(s) we want to focus on, and drown-out irrelevant words (by multiplying them by tiny numbers like 0.001, for example).

The **sixth step** is to sum up the weighted value vectors. This produces the output of the self-attention layer at this position (for the first word).

That concludes the self-attention calculation. The resulting vector is one we can send along to the feed-forward neural network. In the actual implementation, however, this calculation is done in matrix form for faster processing. So let's look at that now that we've seen the intuition of the calculation on the word level.

Matrix Calculation of Self-Attention The first step is to calculate the Query, Key, and Value matrices. We do that by packing our embeddings into a matrix `X`, and multiplying it by the weight matrices we've trained `(WQ, WK, WV)`.

Every row in the `X` matrix corresponds to a word in the input sentence. We again see the difference in size of the embedding vector `(512, or 4 boxes in the figure)`, and the `q/k/v vectors (64, or 3 boxes in the figure)`

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

[ ]:
```

```

Decoder-only The decoder-only transformer, which is more commonly-used for modern LLMs, simply removes the encoder from this architecture and uses only the decoder, as indicated by the name. Practically, this means that every layer of the decoder-only transformer architecture contains the following:

- A masked self-attention layer.
- A feed-forward layer.

To form the full decoder-only transformer architecture, we just stack `L` of these layers, which are identical in structure but have independent weights, on top of each other. A depiction of this structure is provided in the figure below.

Let's now discuss each component of the architecture in isolation to gain a better understanding. We will start with the input structure for the model, followed by the components of each layer (i.e., self-attention and feed-forward layers) and how they are combined to form the full model architecture.

0.0.2 From Text to Tokens

the input to an LLM is just a sequence of text (i.e., the prompt). However, the input that we see in the figure above is not a sequence of text! Rather, the model's input is a list of token vectors. If we are passing text to the model as input, how do we produce these vectors from our textual input?

Tokenization. The first step of constructing the input for an LLM is breaking the raw textual input—a sequence of characters—into discrete tokens. This process, called tokenization, is handled by the model's tokenizer. There are many kinds of tokenizers, but Byte-Pair Encoding (BPE) tokenizers [2] are the most common; see here for more details. These tokenizers take a sequence of raw text as input and break this text into a sequence of discrete tokens as shown in the figure above.

https://www.youtube.com/watch?v=zduSFxRajkE

[2]:
```
from huggingface_hub import login

login(token="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")
```

[8]:
```
import torch
from transformers import AutoTokenizer

# load the llama-3.2 tokenizer
```

```
tokenizer = AutoTokenizer.from_pretrained('meta-llama/Llama-3.1-8B')

# raw text
text = "This raw text will be tokenized"

# create tokens using tokenizer
tokens = tokenizer.tokenize(text)
token_ids = tokenizer.convert_tokens_to_ids(tokens)
# token_ids = tokenizer.encode(text)  # directly create token ids

# view the results
print("Original Text:", text)
print("Tokens:", tokens)
print("Token IDs:", token_ids)

# create token embedding layer
VOCABULARY_SIZE: int = tokenizer.vocab_size
EMBEDDING_DIM: int = 768

token_embedding_layer = torch.nn.Embedding(
    num_embeddings=VOCABULARY_SIZE,
    embedding_dim=EMBEDDING_DIM,
)

# get token embeddings (IDs must be passed as a tensor, not a list)
token_emb = token_embedding_layer(torch.tensor(token_ids))
print(f'Token Embeddings Shape: {token_emb.shape}')
```

```
Original Text: This raw text will be tokenized
Tokens: ['This', 'Ġraw', 'Ġtext', 'Ġwill', 'Ġbe', 'Ġtoken', 'ized']
Token IDs: [2028, 7257, 1495, 690, 387, 4037, 1534]
Token Embeddings Shape: torch.Size([7, 768])
```

Token IDs and Embeddings Each token in the LLM's vocabulary is associated with a unique integer ID. For example, the prior code yields this sequence of IDs when tokenizing our text: `[2028, 7257, 1495, 690, 387, 4037, 1534]`. Each of these IDs is associated with a vector, known as a token embedding, in an embedding layer. An embedding layer is just a large matrix that stores many rows of vector embeddings. To retrieve the embedding for a token, we just lookup the corresponding row—given by the token ID—in the embedding layer; see above.

[15]:
```
print(f'Token embedding matrix:')
print(token_emb)
```

```
Token embedding matrix:
tensor([[-0.7294, -0.2191, -0.0981,  …,  0.3523, -0.1128, -0.8054],
        [ 0.6078,  0.7273,  1.0119,  …,  0.4998,  0.2290,  1.0440],
        [-1.6154,  1.4670, -0.2871,  …,  0.5157, -0.2414, -0.4173],
        …,
 [-0.3272,  1.4531,  0.4898,  …, -0.3586, -0.6169,  0.7317],
 [-1.2175,  0.2563, -0.7830,  …, -0.4600, -0.5081,  0.6184],
 [-2.5497,  0.9171,  1.9708,  …, -0.7857, -0.2745, -1.2188]],
grad_fn=<EmbeddingBackward0>)
```

The token embedding matrix is of size `[C, d]`, where `C` is the number of tokens in our input and `d` is the dimension of token embeddings that is adopted by the LLM. We usually have a batch of `B` input sequences instead of a single input sequence, forming an input matrix of size `[B, C, d]`. The dimension d impacts the sizes of all layers or activations within the transformer, which makes `d` an important hyperparameter choice. Prior to passing this matrix to the transformer as input, we also add a positional embedding to each token in the input, which communicates the position of each token within its sequence to the transformer.

(Masked and Multi-Headed) Self-Attention Now, we are ready to pass our input—a token embedding matrix—to the decoder-only transformer to begin processing. As previously outlined, the transformer contains repeated blocks with self-attention and a feed-forward transformation, each followed by normalization operations. Let's look at self-attention first.

What is self-attention? Put simply, self-attention transforms the representation of each token in a sequence based upon its relationship to other tokens in the sequence. Intuitively, self-attention bases the representation of each token on the other tokens in the sequence (including itself) that are most relevant to that token. In other words, we learn which tokens to "pay attention" to when trying to understand the meaning of a token in our sequence. For example, we see above that the representation for the word making is heavily influenced by the words more and difficult, which help to convey the overall meaning of the sentence.

An attention function maps a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Scaled Dot Product Attention. Given our input token matrix of size [C, d] where `C` is # of tokens in input sequence and `d` is embedding dimensions (i.e., we will assume that we are processing a single input sequence instead of a batch for simplicity), we begin by projecting our input using three separate linear projections, forming three separate sets of (transformed) token vectors. These projections are referred to as the key, query and value projections; see below.

The intuitive reasoning for the name of each projection is as follows:

A `query` is what you use to search for information. It represents the current token for which we want to find other relevant tokens in the sequence.

The `key` represents each other token in the sequence and acts as an index to match the query with other relevant tokens in the sequence.

The `value` is the actual information that is retrieved once a query matches a key. The value is used to compute each token's output in self-attention.

The weight matrix `W_q` learns to transform the token embeddings into a space where they encode 'questions' about the input. Extract and emphasize specific dimensions of the token's representation that are most useful for determining relationships with other tokens. Create a representation that determines "what do I need to focus on (or attend) to understand this input?

The weight matrix `W_k` is a transformed representation of each token into a form that indicates: what this token has to offer" to other tokens in the sequence.

Despite having different names and serving different functions, `Q`, `K`, and `V` initially all come from the same word input embeddings but are transformed differently to serve their specific purposes in the attention mechanism.

Computing attention scores. After projecting the input, we compute an attention score `a[i, j]` for each pair of tokens `[i, j]` in our input sequence. Intuitively, this attention score, which lies in the `[0, 1]` range, captures how much a given token should "pay attention" to another token in the sequence—higher attention scores indicate that a pair of tokens are very relevant to each other. As hinted at above, attention scores are generated using the key and query vectors. We compute `a[i, j]` by taking the dot product of the query vector for token `i` with the key vector for token `j`; see below for a depiction of this process.

Attention scores = `W_q . W_k`. Large values represent close alignment in embedding space.

We can efficiently compute all pairwise attention scores in a sequence by:

- Stacking the query and key vectors into two matrices.
- Multiplying the query matrix with the transposed key matrix.

This operation forms a matrix of size `[C, C]` called the attention matrix—that contains all pairwise attention scores over the entire sequence. From here, we divide each value in the attention matrix by the square root of `d` an approach that has been found to improve training stability and apply a softmax operation to each row of the attention matrix; see below. After softmax has been applied, each row of the attention matrix forms a valid probability distribution—each row contains positive values that sum to one. The `i-th` row of the attention matrix stores probabilities between the `i-th` token and each other token in our sequence.

Computing output.

Once we have the attention scores, deriving the output of self-attention is easy. The output for each token is simply a weighted combination of value vectors, where the weights are given by the attention scores. To compute this output, we simply multiply the attention matrix by the value matrix as shown above. Notably, self-attention preserves the size of its input—a transformed, `d-dimensional` output vector is produced for each token vector within the input.

Masked self-attention. So far, the formulation we have learned is for vanilla (or bidirectional self-attention). As mentioned previously, however, decoder-only transformers use masked self-attention, which modifies the underlying attention pattern by "masking out" tokens that come after each token in the sequence. Each token can only consider tokens that come before it—following tokens are masked.

Let's consider a token sequence `["LLM", "#s", "are", "cool", "."]` and compute masked attention scores for the token "are". So far, we have learned that self-attention will compute an attention score between "are" and every other token in the sequence. With masked self-attention, however, we only compute attention scores for "LLM", "#s", and "are". Masked self-attention prohibits us from looking forward in the sequence! Practically, this is achieved by simply setting all attention scores for these tokens to negative infinity, yielding a pairwise probability of zero for masked tokens after the application of softmax.

Attention heads. The attention operation we have described so far uses softmax to normalize attention scores that are computed across the sequence. Although this approach forms a valid probability distribution, it also limits the ability of self-attention to focus on multiple positions within the sequence—the probability distribution can easily be dominated by one (or a few) words. To solve this issue, we typically compute attention across multiple "heads" in parallel.

Within each head, the masked attention operation is identical. However, we:

Use separate key, query, and value projections for each attention head.

Reduce the dimensionality of the key, query, and value vectors (i.e., this can be done by modifying the linear projection) to reduce computational costs.

More specifically, we will change the dimensionality of vectors in each attention head from `d` to `d // H`, where `H` is the number of attention heads, to keep the computational costs of multi-headed self-attention (relatively) fixed.

Now, we have several attention heads that compute self-attention in parallel. However, we still need to produce a single output representation from the multiple heads of our self-attention module. We have several options for combining the output of each attention head; e.g., concatenation, averaging, projecting, and more. However, the vanilla implementation of multi-headed self-attention does the following (depicted above):

- Concatenates the output of each head.
- Linearly projects the concatenated output.

Because each attention head outputs token vectors of dimension `d // H`, the concatenated output of all attention heads has dimension `d`. Thus, the multi-headed self-attention operation still preserves the original size of the input.

https://gist.github.com/wolfecameron/26863dbbc322b15d2e224a2569868256#file-causal_self_attention-py

[2]:
```python
"""
Source: https://github.com/karpathy/nanoGPT/blob/master/model.py
"""

import math
import torch
from torch import nn
import torch.nn.functional as F

class CausalSelfAttention(nn.Module):

    def __init__(
        self,
        d,
        H,
        T,
        bias=False,
        dropout=0.2,
    ):
        """
        Arguments:
        d: size of embedding dimension
        H: number of attention heads
        T: maximum length of input sequences (in tokens)
        bias: whether or not to use bias in linear layers
        dropout: probability of dropout
        """
        super().__init__()
        assert d % H == 0

        # key, query, value projections for all heads, but in a batch
        # output is 3X the dimension because it includes key, query and value
        self.c_attn = nn.Linear(d, 3*d, bias=bias)

        # projection of concatenated attention head outputs
        self.c_proj = nn.Linear(d, d, bias=bias)

        # dropout modules
        self.attn_dropout = nn.Dropout(dropout)
        self.resid_dropout = nn.Dropout(dropout)
        self.H = H
        self.d = d

        # causal mask to ensure that attention is only applied to
        # the left in the input sequence
        self.register_buffer("mask", torch.tril(torch.ones(T, T))
                                     .view(1, 1, T, T))

    def forward(self, x):
        B, T, _ = x.size() # batch size, sequence length, embedding␣
↪dimensionality

        # compute query, key, and value vectors for all heads in batch
        # split the output into separate query, key, and value tensors
        q, k, v  = self.c_attn(x).split(self.d, dim=2) # [B, T, d]

        # reshape tensor into sequences of smaller token vectors for each head
        k = k.view(B, T, self.H, self.d // self.H).transpose(1, 2) # [B, H, T,␣
↪d // H]
        q = q.view(B, T, self.H, self.d // self.H).transpose(1, 2)
        v = v.view(B, T, self.H, self.d // self.H).transpose(1, 2)

        # compute the attention matrix, perform masking, and apply dropout
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1))) # [B,␣
↪H, T, T]
```

```
        att = att.masked_fill(self.mask[:,:,:T,:T] == 0, float('-inf'))
        att = F.softmax(att, dim=-1)
        att = self.attn_dropout(att)

        # compute output vectors for each token
        y = att @ v # [B, H, T, d // H]

        # concatenate outputs from each attention head and linearly project
        y = y.transpose(1, 2).contiguous().view(B, T, self.d)
        y = self.resid_dropout(self.c_proj(y))
        return y
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[2]:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Define a simple 2-layer Neural Network
class TwoLayerNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(TwoLayerNN, self).__init__()
        # First fully connected layer (Input -> Hidden)
        self.fc1 = nn.Linear(input_size, hidden_size)

        # Activation function (ReLU introduces non-linearity)
        self.relu = nn.ReLU()

        # Second fully connected layer (Hidden -> Output)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        """
        Forward pass of the network.
        :param x: Input tensor
        :return: Output tensor (predictions)
        """
        x = self.fc1(x)      # First layer transformation
        x = self.relu(x)     # Apply ReLU activation
        x = self.fc2(x)      # Second layer transformation
        return x  # Output layer (logits for classification)
```

[ ]:
```
# Example usage
input_size = 3    # Number of input features
hidden_size = 5   # Number of neurons in the hidden layer
output_size = 2   # Number of output classes

# Create the model
model = TwoLayerNN(input_size, hidden_size, output_size)

# Generate some random input (batch size = 4)
x_sample = torch.rand(4, input_size)

# Perform a forward pass
output = model(x_sample)

# Print the output
print("Model Output:\n", output)
```