Aufgabe 6

Reihe 2

Inhaltsverzeichnis

1. Tokenizer
2. Token Embedding / Positional Embedding
3. Attention Head
4. Decoder Stack
5. Classification head
6. Token prediction

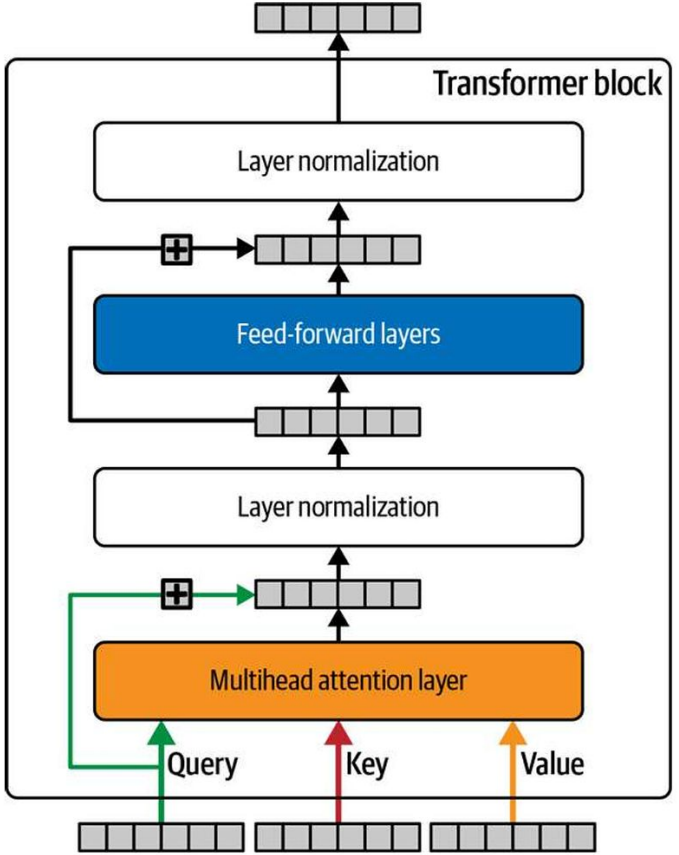
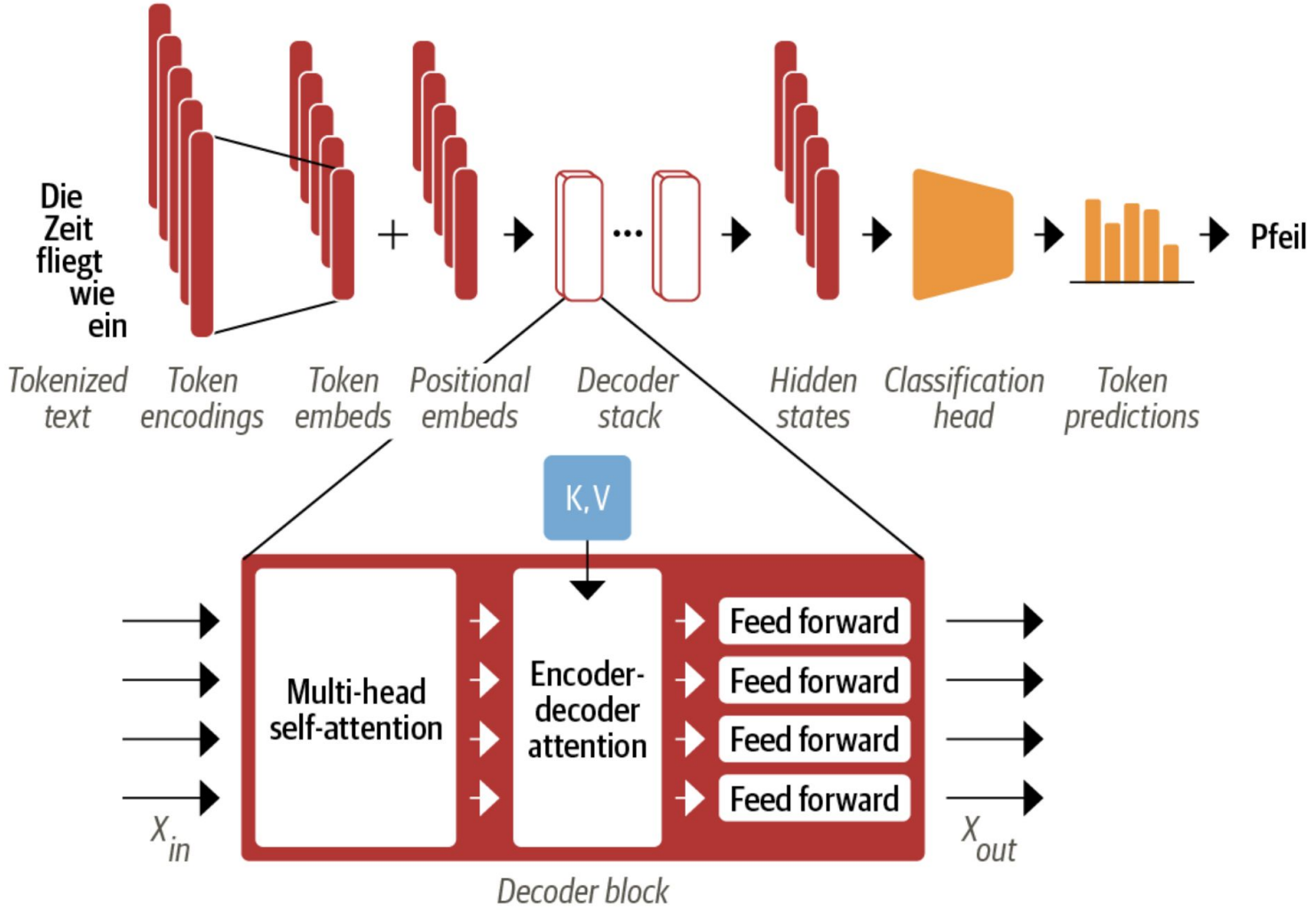


Figure 9-6. A Transformer block


Die
Zeit
fliegt
wie
ein
Pfeil
Tokenized text
Token encodings
Token embeds
Positional embeds
Decoder stack
Hidden states
Classification head
Token predictions
K, V
Multi-head self-attention
Encoder-decoder attention
Feed forward
Feed forward
Feed forward
Feed forward
X_{in}
X_{out}
Decoder block

Tokenizer

- Zeichenkette (Wort(-Teil) oder Buchstabe) wird in numerische Werte zerlegt
- Eingabe wird durch Tokenizer normalisiert (Kleinschreibung, Satzzeichen entfernen, Wörter auf Wortstamm reduzieren (Stemming))
- Token wird ein Index zugeordnet
- Satz wird verlängert, um einheitliche Länge zu erreichen (Padding)
- Verschiedene Tokenizer für verschiedene Anwendungsfälle
- Tokenizer muss die jeweilige Sprache unterstützen (auch multi-language Tokenizer verfügbar)
- Reservierte Tokens für spezielle Zeichen (Separator, Line Break)
- Herausforderungen: Umgangssprache, Noise, Mehrdeutigkeit

Token Embedding / Positional Embedding (PZ)

Token Embedding

=> capture the most nuance, connections, and semantic meanings between tokens

Warum ?

Embeddings können viel mehr Informationen über Tokens, deren Zusammenhänge, deren Ähnlichkeiten, deren Inhalt beinhalten als reine Token

Positional Embedding

=> Reflects the order of the tokens

Warum?

Der **Hund** schaut den **Jungen** an und….

Der **Junge** schaut den **Hund** an und…

Die Position der Wörter bzw. Tokens ist wichtig für das Verständniss und die Vorhersage

Token Embedding / Positional Embedding (PZ)

- Positional Embedding hat meistens den gleichen Shape wie das Token Embedding (muss es aber nicht haben)
- Wenn die beiden Embeddings (Tensoren) die gleiche Länge haben, können sie einfach addiert werden
- Embeddings sind die Wahren Inputs in LLM’s
- Embeddings tragen zur grösse des Models bei
- Embeddings sind, defacto, ein Subset der Gewichte des Models
- Embeddings können klassisch Initialisiert werden oder von einem Pre-trainierten Model übernommen werden
- Je besser die Embeddings sind, desto besser ist auch das Model

Token Embedding / Positional Embedding (PZ)

```
class TokenAndPositionEmbedding(layers.Layer):
        def __init__(self, maxlen, vocab_size, embed_dim):
                super(TokenAndPositionEmbedding, self).__init__()
                self.maxlen = maxlen
                self.vocab_size =vocab_size
                self.embed_dim = embed_dim
                self.token_emb = layers.Embedding(
                input_dim=vocab_size, output_dim=embed_dim
                )
                self.pos_emb = layers.Embedding(input_dim=maxlen,
                output_dim=embed_dim)

        def call(self, x):
                maxlen = tf.shape(x)[-1]
                positions = tf.range(start=0, limit=maxlen, delta=1)
                positions = self.pos_emb(positions)
                x = self.token_emb(x)
                return x + positions
```

Attention Head

Was ist ein Attention Head?

Ein *Attention Head* ist wie ein Detektiv, der untersucht, wie Wörter in einem Satz miteinander verbunden sind..

Beispiel-Satz:

"Der Hund jagt die Katze, weil sie schnell ist."

1. **Fragen stellen (Query):**
   Jedes Wort fragt: „Was ist für mich wichtig?“
   - Beispiel: "sie" fragt: „Worauf beziehe ich mich?“
2. **Angebote prüfen (Key):**
   Alle Wörter bieten ihre Bedeutung an: „Das bin ich, beachte mich!“
   - Beispiel: "die Katze" sagt: „Ich bin die, die schnell ist.“
3. **Antworten auswählen (Value):**
   Der Attention Head entscheidet, welches Wort am wichtigsten ist.
   - Ergebnis: "sie" → "die Katze"

Warum mehrere Attention Heads?

- Jeder Head hat eine Aufgabe:
  - Einer achtet auf Subjekte, z. B. "Der Hund".
  - Ein anderer achtet auf Verben, z. B. "jagt".
- Gemeinsam verstehen sie den Satz besser!

Ein *Attention Head* hilft dem Modell:

1. Herauszufinden, welche Wörter wichtig sind.
2. Wörter und ihre Verbindungen im Kontext zu verstehen.

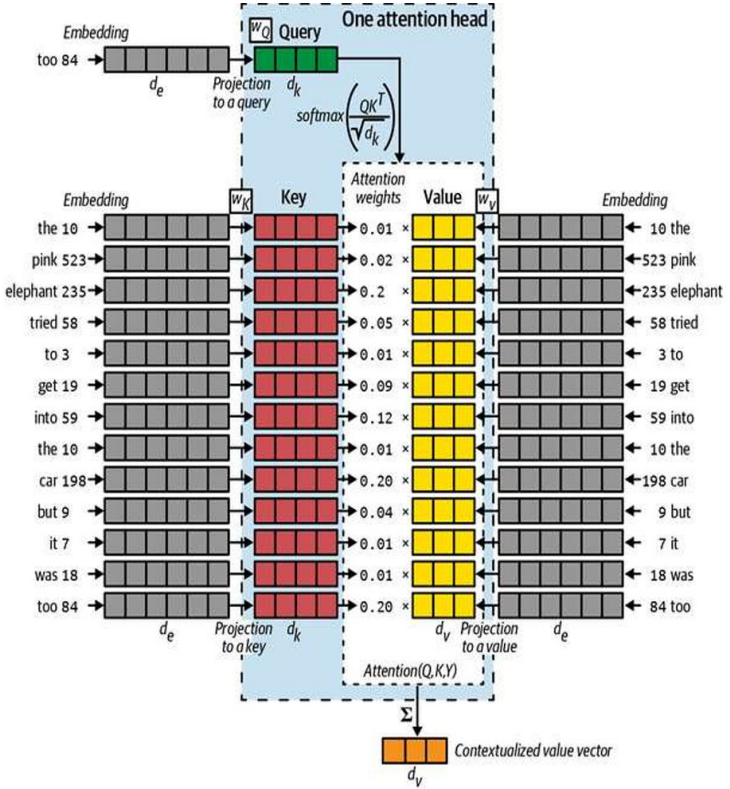


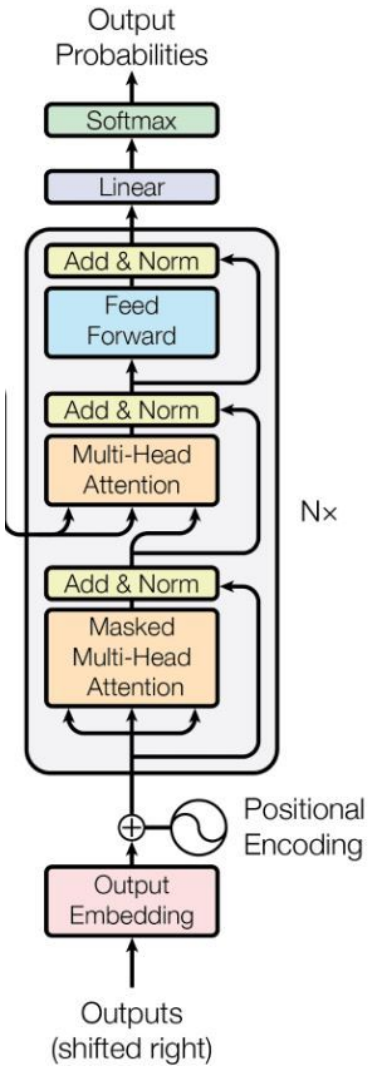
Figure 9-2. The mechanics of an attention head

To obtain the final output vector from the attention head, the attention is summed to give a vector of length d_v. This *context vector* captures a blended opinion from words in the sentence on the task of predicting what word follows *too*.

Decoder Stack

The Decoder stack contains a number of Decoders. Each Decoder contains:

- Two Multi-Head Attention layers
- Feed-forward layer



Decoder Stack - Multi-Head Attention Layer

Multiple Attention Heads: Instead of having a single attention mechanism, multi-head attention uses several attention mechanisms, called heads. Each head processes the input data in parallel but focuses on different parts of the input sequence.

Parallel Processing: Each attention head performs its self-attention calculation independently, producing different representations of the same input sequence.

Concatenation: The outputs from all attention heads are concatenated (combined) into a single representation.

Linear Transformation: The concatenated output undergoes a linear transformation to blend the information from all heads.

Decoder Stack - Multi-Head Attention Layer

Components of Multi-Head Attention

1. **Linear Layers**: Yes, linear layers are used extensively to project the input data into different subspaces.
    - **Query, Key, Value Projections**: These are linear transformations to create queries, keys, and values from the input.
    - **Output Projection**: Another linear layer to combine the outputs from different heads.
2. **Attention Mechanism**: This is the core part of the Multi-Head Attention.
    - **Scaled Dot-Product Attention**: Computes the attention scores by taking the dot product of queries and keys, scaling by the square root of the dimension, and applying a softmax function to get the weights.
3. **Concatenation and Final Linear Layer**: After calculating the attention scores and generating the weighted sum of the values, the outputs from all heads are concatenated and passed through a final linear layer.

Decoder Stack - Multi-Head Attention Layer

Benefits

- **Multiple Perspectives**: By having multiple heads, the model can learn different representations and capture various aspects of the input data. This is especially useful for understanding complex patterns in language.
- **Improved Accuracy**: Multi-head attention allows the model to focus on different parts of the sequence simultaneously, leading to more accurate and nuanced representations.

Applications

- **Machine Translation**: Helps in capturing different translations by focusing on various aspects of sentences.
- **Text Summarization**: Extracts key information from lengthy documents by attending to multiple parts simultaneously.
- **Language Models**: Used in models like BERT and GPT to improve understanding of context and relationships in text.

Decoder Stack - Feed-Forward Neural Network (FFN)

1. **Two Linear Transformations**: The feed-forward part consists of two fully connected linear layers with a ReLU activation function in between.
    - **Layer 1**: Takes the input from the attention mechanism and applies a linear transformation.
    - **ReLU Activation**: Introduces non-linearity, allowing the network to learn more complex patterns.
    - **Layer 2**: Applies another linear transformation to the output of the ReLU activation.

2. **Independence Across Positions**: Unlike the attention mechanism, which considers relationships between different positions in the input sequence, the feed-forward part processes each position independently.

3. **Output**: The transformed data is passed on to subsequent layers or returned as the final output of the model.

Decoder Stack - Why Masked Multi-Head Attention and Multi-Head Attention layers

Masked Multi-Head Attention

- **Purpose**: Used primarily in the **decoder** part of the transformer architecture, especially during tasks like language generation.
- **Function**: Masks out future tokens (words) in a sequence to prevent the model from "cheating" by looking ahead at future tokens when generating the current token. This ensures that predictions for the next word depend only on the words that have come before it.
- **Use Case**: Essential for autoregressive tasks where the model generates text one token at a time, like in language models and text generation.

Masked Multi-Head Attention: Ensures the autoregressive property by masking future tokens, which is crucial for generating coherent and sequential text without future information.

Multi-Head Attention

- **Purpose**: Used in both the **encoder** and **decoder** parts of the transformer architecture.
- **Function**: Allows the model to attend to different parts of the input sequence to gather context, improving its understanding of relationships between tokens (words).
- **Use Case**: In the encoder, it processes the entire input sequence to create a rich representation. In the decoder, it helps the model focus on relevant parts of the input sequence while generating output.

Multi-Head Attention: Provides the model with the ability to learn intricate relationships within the input data, enhancing its contextual understanding and overall performance.

Decoder Stack - Why Masked Multi-Head Attention and Multi-Head Attention layers

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


class MultiHeadAttention(nn.Module):
    def __init__(self, d_model, num_heads):
        super(MultiHeadAttention, self).__init__()
        # d_model should be divisible by num_heads
        assert d_model % num_heads == 0

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads

        self.q_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.out = nn.Linear(d_model, d_model)
```

```python
def forward(self, q, k, v, mask=None):
    batch_size = q.size(0)

    # Perform linear operation and split into num_heads
    q = self.q_linear(q).view(batch_size, -1, self.num_heads, self.d_k)
    k = self.k_linear(k).view(batch_size, -1, self.num_heads, self.d_k)
    v = self.v_linear(v).view(batch_size, -1, self.num_heads, self.d_k)

    # Transpose to get dimensions batch_size * num_heads * seq_len * d_k
    q = q.transpose(1, 2)
    k = k.transpose(1, 2)
    v = v.transpose(1, 2)

    # Apply attention
    scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(
        torch.tensor(self.d_k, dtype=torch.float32)
    )
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    attention = F.softmax(scores, dim=-1)
    x = torch.matmul(attention, v)

    # Concatenate heads and put through final linear layer
    x = x.transpose(1, 2).contiguous().view(batch_size, -1, self.d_model)
    x = self.out(x)

    return x
```

Classification head (EO)

The classification head is the final part of a neural network that takes the rich representations learned by the model (like a transformer) and converts them into specific classification outputs. Let me explain with an analogy and then get technical:

Analogy: Imagine you're a wine expert who has developed an incredibly sophisticated way of analysing wines (like the transformer backbone). But at the end, you need to make specific decisions: Is this wine red or white? What region is it from? The classification head is like your final decision-making process that takes all your detailed analysis and produces specific answers.

Basic Structure:

- Takes the output representations from the transformer
- Usually consists of one or more linear layers (matrices of weights)
- Often includes an activation function (like softmax for multi-class classification)
- Outputs probabilities or scores for each possible class

Token prediction - Eine einfache Einführung

Für den Dilettanten

Token Prediction in Transformers ist der Prozess, in dem ein KI-Modell den nächsten ***Baustein*** (z. B. ein Wort) in einer Folge “errät”. Dazu werden dem Modell eine Folge von z.B. Text-Bausteinen übergeben, welche das Modell versteht und daraus das Wort mit der höchsten Wahrscheinlichkeit hinzufügt.

Beispiel: Wenn du sagst *"Die Katze sitzt auf dem"*, wird das Modell zum Beispiel *"Baum"* oder *"Sofa"* als nächstes Wort vorschlagen.

Für den Experten

Token Prediction in Transformers basiert auf der Verarbeitung des Kontextes der bereits übergebenen Folge durch Eigen- / Kontextaufmerksamkeit (Self-Attention). Das Modell berechnet, wie stark jedes Token mit anderen in der Sequenz zusammenhängt. Im autoregressiven Modus (z. B. GPT) wird die Wahrscheinlichkeit des nächsten Tokens P(xt+1∣x1,x2,...,xt)P(x_{t+1} | x_1, x_2, ..., x_t)P(xt+1∣x1,x2,...,xt) basierend auf vorherigen Tokens maximiert.

Token prediction - Daten / Algorithmus

Linearer Klassifikator und Softmax zur Erzeugung der Ausgabewahrscheinlichkeiten

1. Zuerst fliessen die Daten aus dem Transformermodell durch eine lineare Schicht, die als Klassifikator fungiert.
   a. Die Größe des Klassifikators entspricht der Gesamtzahl der Klass
   (Anzahl der im Vokabular enthaltenen Wörter).
   In einem Szenario mit 1000 verschiedenen Klassen,
   die 1000 verschiedene Wörter repräsentieren,
   ist die Ausgabe des Klassifikators ein Array mit 1000 Elementen.

2. Der Output wird Softmax übergebe, um eine Reihe von Wahrscheinlichkeitswerten (0 und 1) zu generieren.
   Der höchste Wahrscheinlichkeitswerte kann (muss aber nicht)
   der Schlüssel zum nächsten Token bzw. Wort im Kontext NLP sein.