Explainable AI - Lecture 9

Transformers

norwegian open ai lab
NTNU

XAI for transformers

We'll look at three approaches to XAI'ing transformers; two we already know and one new:

- Grad-SAM (new)
- SHAP (known)
- TCAV (known)

First, we'll do a crash course on the transformer architecture.

Disclaimer: I'm no transformer expert, so if you feel confused after this lecture, I'm sorry. In that case, I recommend the following resources:

➔ Attention is all you need, original paper by Wasvani et al (2017)

➔ Attention in transformers, step-by-step by 3Blue1Brown on YouTube.
Note that 3Blue1Brown uses different convention than the original paper when calculating the attention (K^TQ, instead of QK^T)

Transformer architecture basics

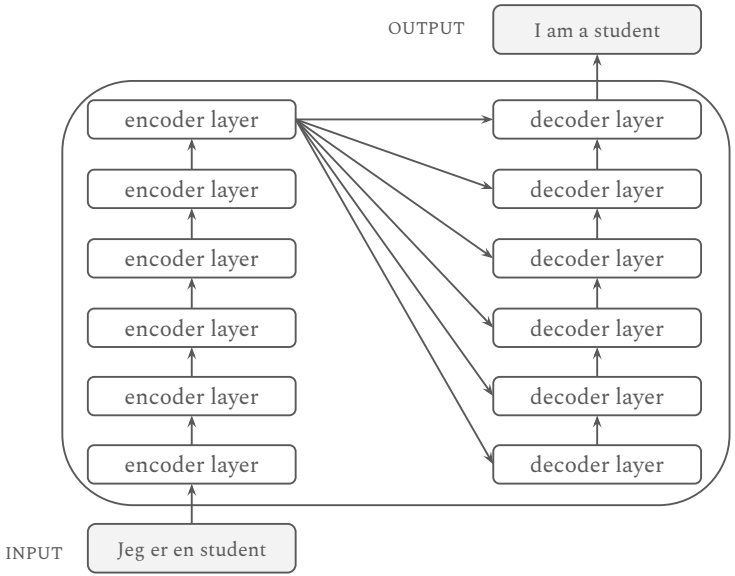
Attention is all you need, 2017

The transformer (for machine translation)

The transformer is a neural network architecture for **sequential processing** tasks.

The original transformer architecture was proposed for machine **translation** in the paper *Attention Is All You Need* by Vaswani et al (2017).

Overview of structure:
An **encoder** maps an input sequence to a sequence of continuous representations from which a **decoder** generates an output sequence, one element at a time.



Transformer architecture

Main components

- **Input**:
  - Embedding
  - Positional encoding
- **Encoder**, a stack of identical layers, each containing:
  - Multi-head self-attention
  - Feed-forward network
- **Decoder**, a stack of identical layers, each containing:
  - Masked multi-head self-attention
  - Multi-head cross-attention
  - Feed-forward network
- **Output**:
  - Linear layer with softmax

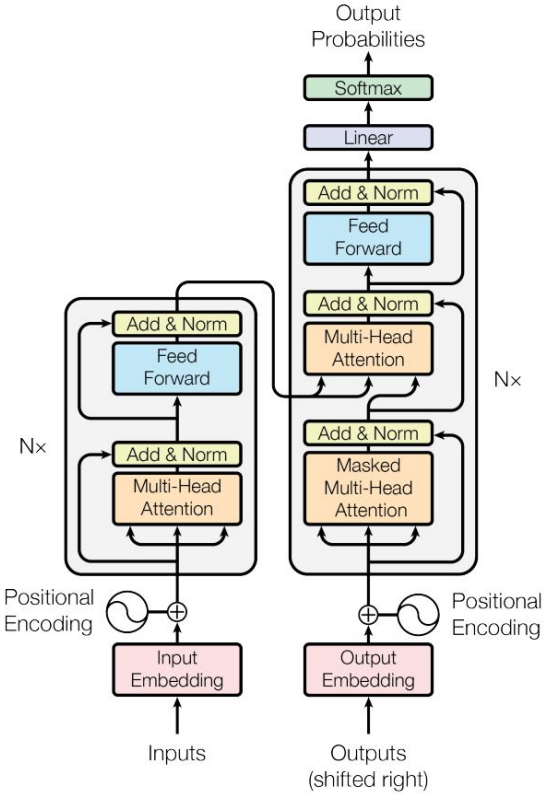
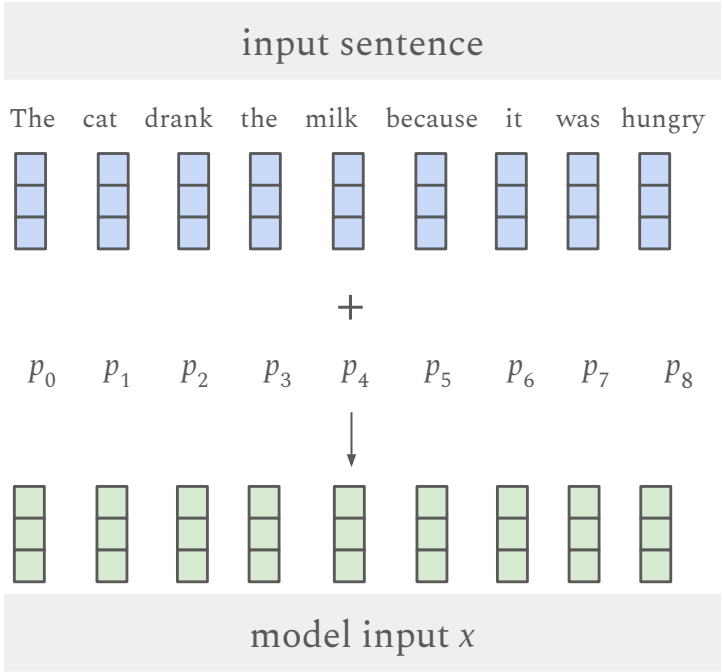


Figure from Vaswani et al (2017).

Tokens and embedding

A **token** is a unit of the input data. In NLP, this is typically a single word or a subword. In images, it is a small, fixed-size image patch, consisting of $n \times n$ pixels.

The transformer input is a sequence of tokens.



Tokens and embedding

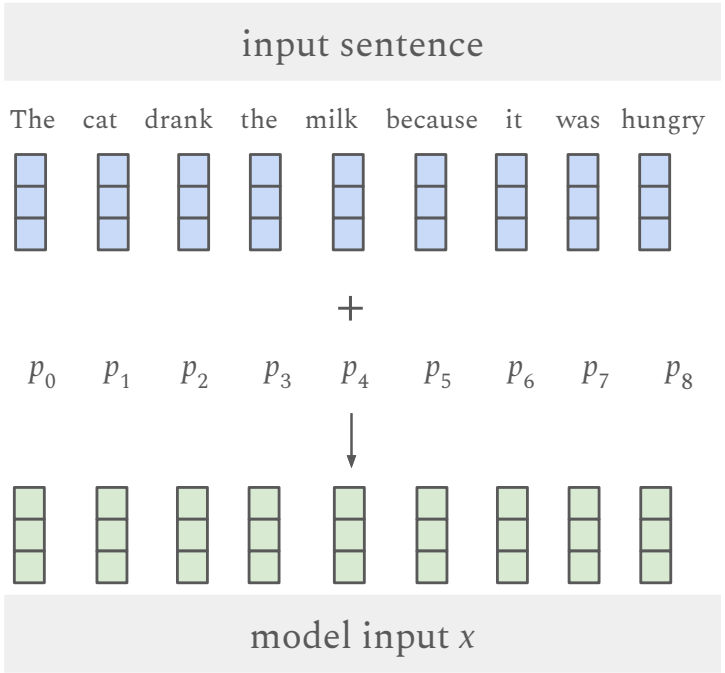
A **token** is a unit of the input data. In NLP, this is typically a single word or a subword. In images, it is a small, fixed-size image patch, consisting of $n \times n$ pixels.
The transformer input is a sequence of tokens.

The available tokens constitute the model's vocabulary.
Tokens can be

- whole words ("apple")
- subword units ("ing", "tion", "un-")
- characters (rare in modern LLMs)
- special symbols (<SEP>, <EOS>, <UNK>, <CLS>)

Each token corresponds to an integer ID. For example:

```
"playing" --> ["play", "ing"] --> [1023, 817]
```



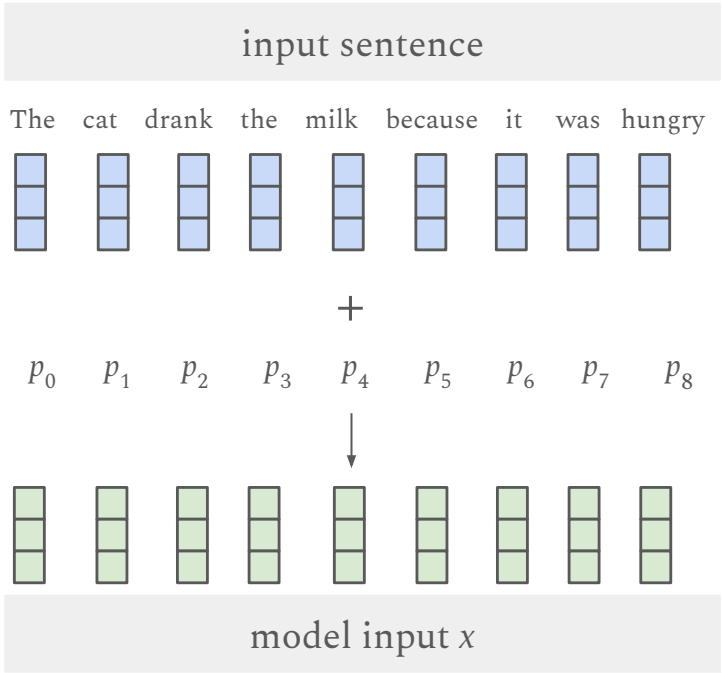
Tokens and embedding

Tokens are mapped into a vector representation, called the token **embedding.** This representation is *learned* during model training, and its dimensionality depends on the model architecture.

Intuitively, the embedding is a lookup table $E \in \mathbb{R}^{Vxd}$
V = size of the vocabulary, d_{model}= embedding dimensionality, and each row in E is the embedding vector for a token ID

```
"play" (ID=1023) --> E[1023]=[0.12, -0.77,...,0.03] # dimension d
```

For example, BERT has embedding dimension 768, and 30,000 tokens in its vocabulary.



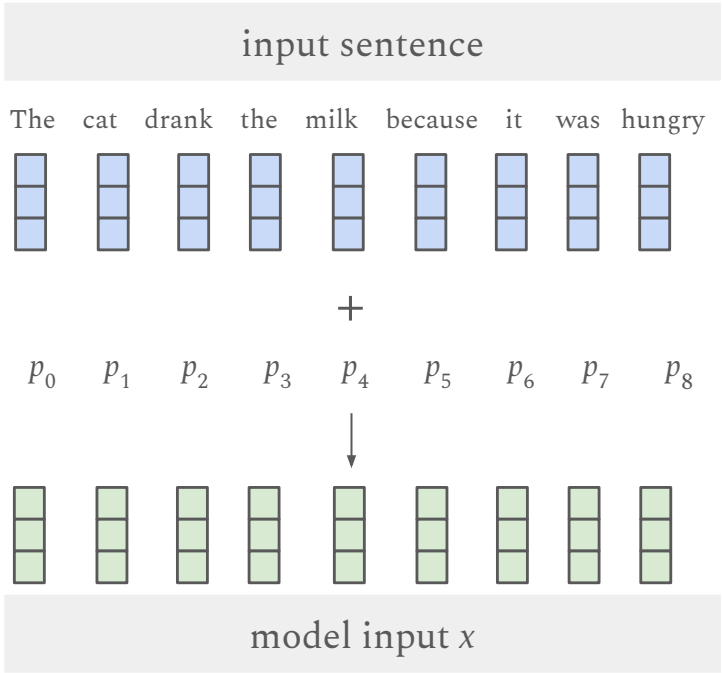
Tokens and embedding

Tokens are mapped into a vector representation, called the token **embedding**. This representation is *learned* during model training, and its dimensionality depends on the model architecture.

Token embeddings have the right dimensionality, but contain no positional information: the token "play" has the same embedding vector regardless of where it appears in the sequence.

To preserve positional information, positional encoding is added element-wise to create the actual input vector:

```
x_t = token_embedding(token_t) + positional_embedding(t)
```

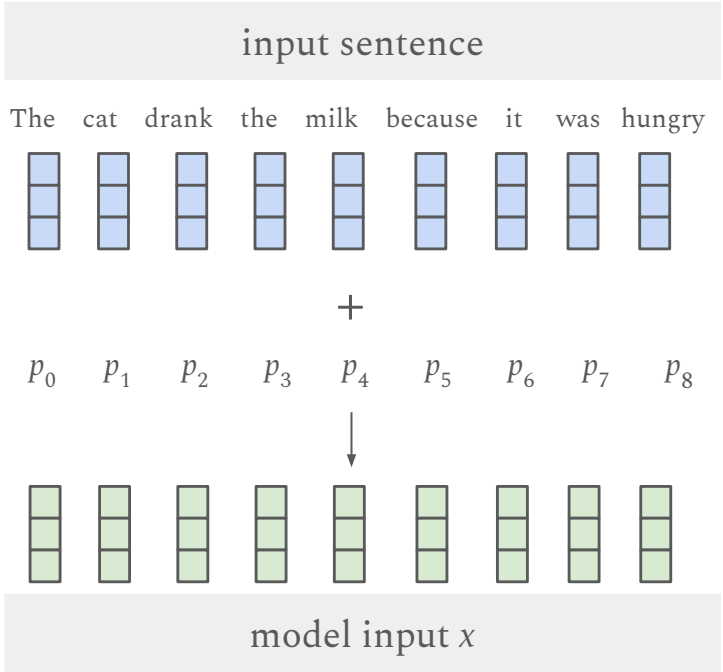


Tokens, embedding and positional encoding

A **token** is a unit of the input data. In NLP, this is typically a single word or a subword. In images, it is a small, fixed-size image patch, consisting of $n \times n$ pixels.
The transformer input is a sequence of tokens.

The token **embedding** is the d-dimensional vector representation of a token, *learned* during model training, with d matching the dimensionality of the model (d_{model})

To preserve information of a token's position in the sequence, a **positional encoding** vector (of the same dimension as the embedding) is added to the token embedding. The positional encoding can be either *fixed* or *learned*.



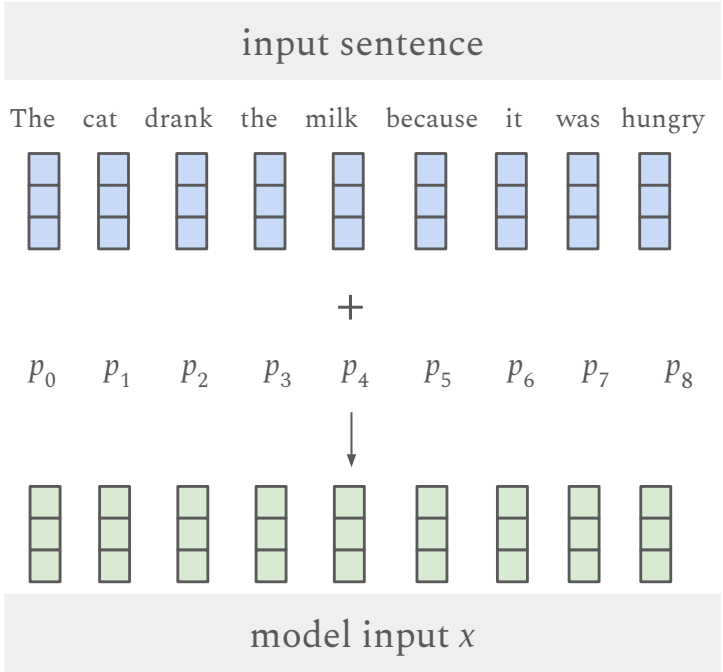
Tokens, embedding and positional encoding

Intuitively:

A **token** = an input piece

The **embedding** = token in a meaningful vector representation

The **positional encoding** = where in the input the token is



The positional encoding

For example, BERT and GPT-2 use learned positional encodings. This is learned as the model is trained.

GPT-2 has a context window (maximum input sequence length) of 1024, so the positional encoding matrix is $P \in \mathbb{R}^{1024 \text{x} d}$.

Vaswani et al. (2017) use a fixed positional encoding, obtained from sine and cosine functions.

Fixed positional encoding

The fixed positional encoding is obtained from sine and cosine functions of different frequencies:

$$\mathrm{PE}_{(\mathrm{pos},2i)} = \sin(\mathrm{pos}/10000^{2i/d_{\mathrm{model}}})$$

$$\mathrm{PE}_{(\mathrm{pos},2i+1)} = \cos(\mathrm{pos}/10000^{2i/d_{\mathrm{model}}})$$

Here:
pos is the actual position of the token in the sentence (its index),
i is the dimension index, running from 0,1,2,..., to $d_{\mathrm{model}}/2–1$
d_{model} is the embedding dimension, like before.

These equations gives the *2i*-th and *(2i + 1)*-th components of the positional encoding vector (i.e., all encodings in total).

Fixed positional encoding

Even dimensions use the sine function, while odd dimensions use the cosine function. Each pair gives a unique 2D representation of position for a particular frequency. The denominator sets a frequency that varies across dimensions, with wavelengths ranging from small to very large.

Intuitively, since each vector dimension (or pair of dimensions) uses a distinct sinusoidal frequency, the encoding function produces a **unique d_{model}-dimensional vector for every token position** in the sequence. Moreover, since sine and cosine vary smoothly with respect to their input, **adjacent token positions have similar positional encodings** – nearby tokens yield nearby vectors in the embedding space.

Learned positional encoding

While Vaswani et al (2017) used sine and cosine functions to get a **fixed** positional encoding, it can also be **learned** by the model, similar to the token embeddings.

A **learned positional encoding** can be implemented as a learnable lookup table (a matrix), where each row corresponds to a position index in the sequence and stores a trainable vector of dimension d_{model} representing that position.

Tokens, embedding and positional encoding

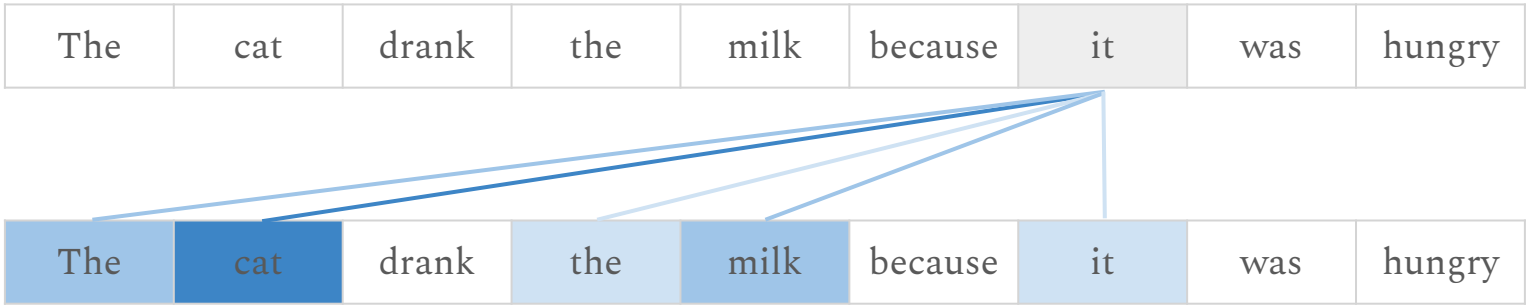
With the tokenization, embedding and positional encoding in place, the model input is modified to be useful to the transformer. Pseudocode:

```
#   tokens: list of token IDs, length = seq_len
#   E: token embedding matrix, shape [vocab_size, d_model]
#   P: positional embedding matrix, shape [max_seq_len, d_model]
#   X: embedded sequence, shape [seq_len, d_model]

seq_len = length(tokens)
X = zeros(seq_len, d_model)
for t in range(0, seq_len):
     token_id = tokens[t]      # integer
     token_vec = E[token_id]   # shape [d_model]
     pos_vec   = P[t]          # shape [d_model]

     X[t] = token_vec + pos_vec # element-wise addition
```

Attention mechanism

(Self-)Attention enables the model to determine which tokens are relevant to each other. This allows it to build context-dependent representations.

The example shows how the token “it” *attends* to every other token in the sequence. When the model processes the token “it”, self-attention provides additional context so the model can associate this token with the correct token “cat”.



Scaled dot-product self-attention

The goal of self-attention is to identify and attend to the most important features in the sequence
(tokens attend within the same sentence ← "self")

Each token is associated with three vectors: **query**, **key** and **value**.

> **Query**: *Request for information, representing what the token is looking for in the sequence*
>
> **Key**: *Label or identifier used to determine the token's relevance to a given query*
>
> **Value**: *The actual information associated with the token*

Think of this as looking for information: The query is the searcher, the key is the searchable entry, and the value is the the retrieved content.

Scaled dot-product self-attention

The goal of self-attention is to identify and attend to the most important features in the sequence
(tokens attend within the same sentence ← "self")

Each token is associated with three vectors: **query**, **key** and **value**.

> **Query**: *Request for information, representing what the token is looking for in the sequence*
>
> **Key**: *Label or identifier used to determine the token's relevance to a given query*
>
> **Value**: *The actual information associated with the token*

In a transformer, the query asks: "Which other tokens in the sequence contain information relevant to me?". The key describes what a token contains, and the value holds the actual information.

Scaled dot-product self-attention

The goal of self-attention is to identify and attend to the most important features in the sequence
(tokens attend within the same sentence ← "self")

Each token is associated with three vectors: **query**, **key** and **value**.

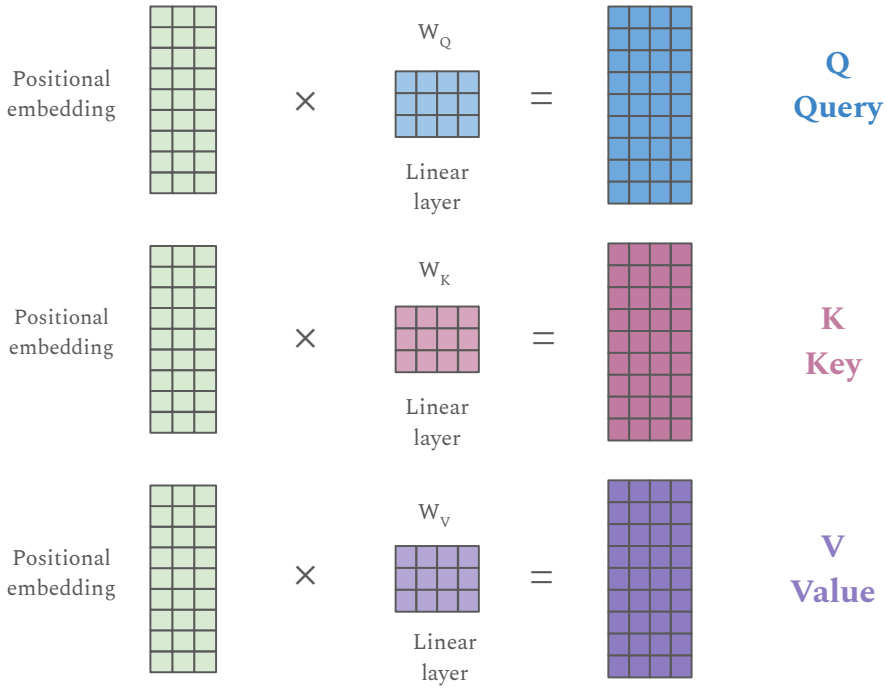
> **Query**: *Request for information, representing what the token is looking for in the sequence*
>
> **Key**: *Label or identifier used to determine the token's relevance to a given query*
>
> **Value**: *The actual information associated with the token*

The weights of these are all *learned* during model training.

The query, key and value vectors for all tokens are stacked into matrices **Q**, **K**, and **V**, respectively.

Scaled dot-product self-attention

Queries, keys and values are obtained by separate linear projections of the input.



Scaled dot-product self-attention

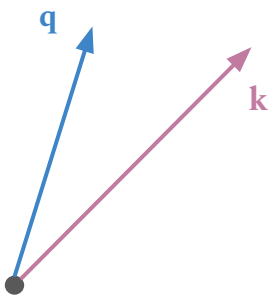
How to find out how relevant / similar the key is to the query?

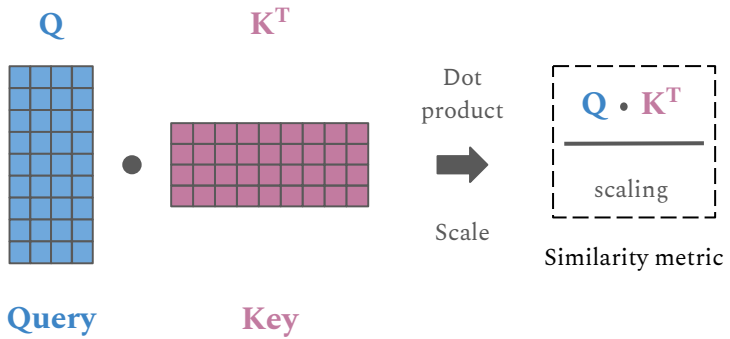
Scaled dot-product self-attention

How to find out how relevant / similar the key is to the query?

Pairwise similarity between each **query** and **key** is calculated through the dot product.

The dot product is multiplied by a scaling factor.



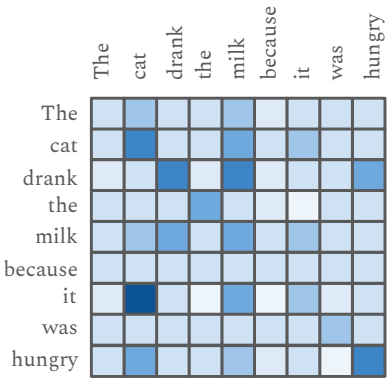


Scaled dot-product self-attention

Apply softmax to turn the scaled dot product into a weight. These are the **attention weights**.

Interpretation: For each token in the sequence, the associated attention weights tells how much a token attends to all other tokens.

High attention weight = the token attends more to that token



$$= \text{softmax}\left(\frac{Q \cdot K^T}{\text{scaling}}\right)$$

Matrix columns are K and rows Q. Entries are (Q_{id} K_{dj})

The softmax is taken so that the queries sum to 1.

Scaled dot-product self-attention

The key, query, value stuff is often confusing the first time you hear about it. As an *analogy*, consider that:

The query asks *"are there any adjectives in front of me?"*
["funny person" and "angry person" or "large tower" and "tiny tower" have different meanings]

$$E \times W_Q = Q$$

Learnable weights ⇒ the model finds out what to query for

Embedding with positional encoding: what and where the word is, but no contextual information

Scaled dot-product self-attention

The key, query stuff is often confusing the first time you hear about it. As an *analogy*, consider that:

The query asks *"are there any adjectives in front of me?"*
["funny person" and "angry person" or "large tower" and "tiny tower" carry different meanings]

$E \times W_Q = Q$

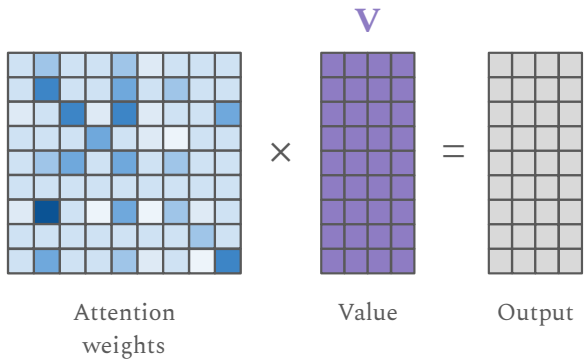
The key represents each token's relevance to the query. When Q and K align closely, the key is relevant for answering the query. In this analogy, the key value of adjectives will be large.

$E \times W_K = K$

Together, Q and K yield the attention weights. These determine who attends to whom.

Scaled dot-product self-attention

Why do we call them "attention weights"?



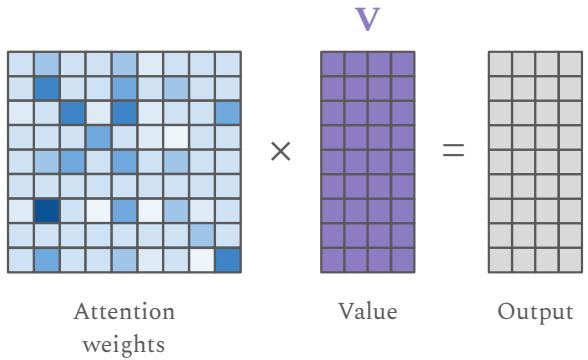
Scaled dot-product self-attention

The attention weights are the weights of the **value** vectors.

The **attention values** are calculated as a weighted sum of each value vector by the corresponding attention weights.

Value vectors associated with (multiplied by) higher weights contribute more to the final attention values.

While the query and key vectors determine who attends to whom, values determine what is actually passed forward, weighted by the attention weights.



Scaled dot-product self-attention

The scaled dot-product attention is computed as,

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

d_k denotes the dimension of the query and key vectors (this is d_{model}, if we ignore multiple heads)

Detail: The scaling is done for normalisation: As d_k increases, the magnitude of the dot product can grow significantly (the variance of a dot product of random vectors grows proportionally to the dimensionality), causing the softmax to saturate and potentially vanishing gradients.

Multi-head attention

In **multi-head attention**, the attention function is computed multiple times in parallel, on a lower-dimensional representation of the same input.

Each head is an independent part of the network, with independent Q, K and V matrices. The outputs from all heads are later combined. *This is the classic ensemble mindset.*

The outputs from all heads are combined by concatenation, and linearly projected to the model's dimensionality (d_{model}).

Multi-head attention allows the model to focus on different relationships in the sequence simultaneously.

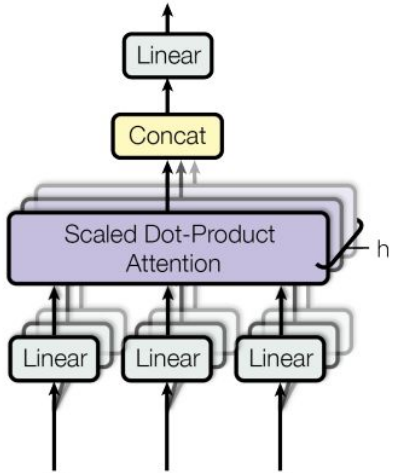
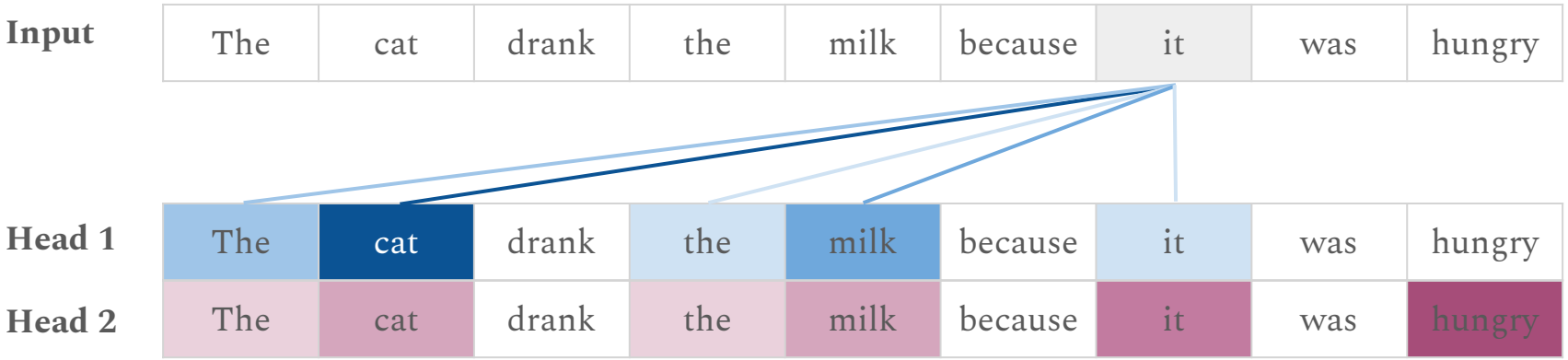


Figure from Vaswani et al (2017).

Multi-head attention, illustration:

When the model processes the token “it”, the first attention head highlights the token “cat”, while the second focuses on “hungry”. This way, both the relationships between “it” and “cat”, and “it” and “hungry” are modelled.



Code implementation of self-attention

```python
class Head(nn.Module):
    def __init__(self, head_size, n_embd, block_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)

    def forward(self, x):
        k = self.key(x)
        q = self.query(x)

        attn_wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5
        attn_wei = F.softmax(attn_wei, dim=-1)
        v = self.value(x)
        out = attn_wei @ v

        return out
```

Define the linear layers W_Q, W_K and W_V

The first dimension is the dimension of the embedding space

The second dimension is conventionally* *head_size = n_embd // n_head,*

$E \times W_Q = Q$; $E \times W_K = K$; $E \times W_V = V$

$A = Q \times K^T / \sqrt{d_k}$

A = softmax(A) dot V

Return the computed attention

*Attention Is All You Need, Vaswani et al (2017)

Code implementation of multi-head self-attention

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, n_heads, head_size, n_embd, block_size, dropout):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size, n_embd, block_size) for _ in range(n_heads)])
        self.proj = nn.Linear(head_size * n_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
```

Compute the attention function for all *n_heads* heads independently (*list comprehension using the Head class*).

Concatenate the output from each head.

Project the result through a linear layer to get back to the embedding dimension *n_embd.*

Before returning the output, apply dropout to avoid over-fitting.

Adaptations of the transformer architecture

GPT - Transformers for text generation

Radford et al (2018) proposed **GPT (Generative Pre-trained Transformer)** as an adaption of the original transformer architecture by Vaswani et al (2017) for text generation.

GPT utilizes only the decoder-part of the original architecture, originally made to take the encoded input and predict a new sequence. GPT iteratively predicts the next token given an input sequence.

The model also uses an extended version of the self-attention mechanism we looked at; so-called *masked* multi-head self-attention. This *masked attention* prevents the model from attending to future tokens (which isn't possible, so this is central for generative models).

The model undergoes an unsupervised pre-training on unlabeled text data followed by supervised fine-tuning for a specific task.

We will not be XAI-ing GPT models in this lecture.

BERT - Transformers for text understanding

Devlin et al (2019) proposed **BERT (Bidirectional Encoder Representations from Transformers)** which is a adaption of the transformer architecture by Vaswani et al (2017).

BERT utilizes only the the encoder part of the original transformer architecture and uses no masking.

The model is pre-trained on large-scale text data and can be fine-tuned for downstream tasks such as text classification.

Example: Classify the sentences as either expressing a positive or a negative sentiment

Input	**Label**
I love this movie!	1
The service was terrible.	0

We will use this sentiment classification in the XAI part.

At a glance, difference between original, GPT and BERT:

Original transformer (Vaswani et al)

Encoder
Processes the input sequence in parallel
Bidirectional self-attention (look left and right)

Decoder
Generates output tokens one by one
Uses causal (left-to-right) self-attention
Can also attend to the encoder output (cross-attention)

→ This design is well-suited for sequence-to-sequence tasks like translation.

GPT

No encoder

Decoder
Generates output tokens one by one
Uses causal (left-to-right) self-attention
No encoder ⇒ no cross-attention

→ This design is well-suited for generation tasks, as it looks only at past tokens, and generates new tokens sequentially

BERT

Encoder
Processes the entire sequence in parallel
Bidirectional self-attention

No decoder
No causal mask
No autoregressive generation

→ This design is well-suited for classification, sentiment, question answering, … *not* generation.

Exactly which transformers we'll be X'ing:

BERT

To every tokenized sequence, we add a special **classification token** (**[CLS]**) as the first element. The final hidden state of the last hidden layer corresponding to this token is used for classification tasks as an aggregate representation of the entire sequence.

A special **separator token** (**[SEP]**) is used to separate distinct text segments, and appears at the end of sequences and between sentence pairs (e.g. for question-answering tasks).

The special tokens are also associated with embedding vectors and positional encodings - exactly like other tokens.

Example: [CLS] tokenized input sentence [SEP]

ViT - Transformers for image classification

Dosovitskiy et al (2021) proposed **ViT (Vision Transformer)**, an adaptation of the original transformer by Vaswani et al (2017) for image classification.

The model architecture is a transformer encoder with an MLP classification head.

The model is pre-trained on large image datasets and fine-tuned to (smaller) downstream tasks.

The input image is divided into patches which are converted to embedding vectors through a linear projection combined with positional information.

ViT uses a classification embedding at the beginning of the sequence, similar as BERT.

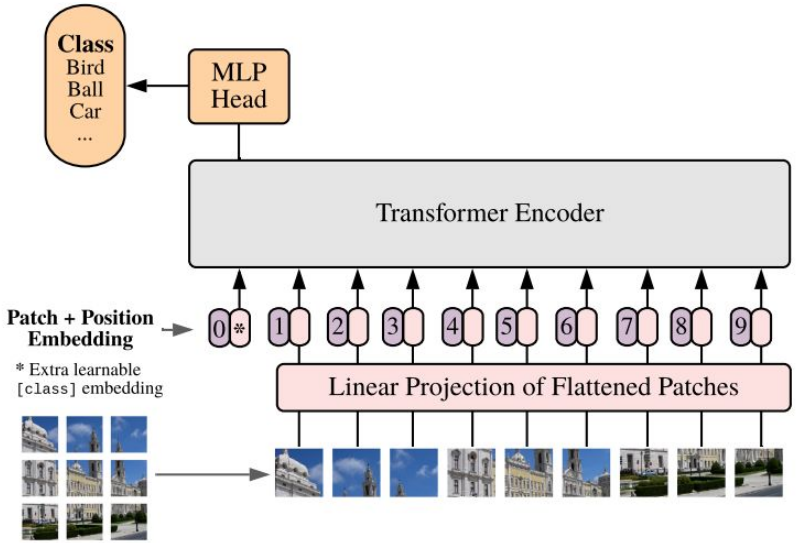


Figure from Dosovitskiy et al (2021).

Explanation methods for transformers

We will look at three XAI methods for transformers:

1. Grad-SAM (this one is new to us) - for sentiment classification with BERT
2. SHAP (we already know this one) - for sentiment classification with BERT
3. TCAV (we already know this one) - for image classification with a ViT

Grad-SAM

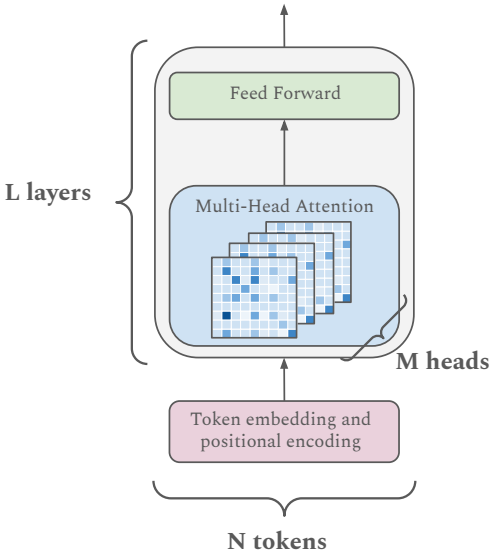
Barkan et al (2022) proposed **Grad-SAM**, which combines attention and gradient information to obtain token-level attributions.

The Grad-SAM score of the i'th input token x_i is

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

The sum is over
N: number of tokens in x
M: number of attention heads
L: number of encoder layers

And H is a Hadamard product we have to calculate…



Grad-SAM

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

The Grad-SAM score r of the i'th input token x_i is the sum over all sequence elements j, attention heads m, and encoder layers l, of

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

: the Hadamard product between the attention values and the ReLU of the gradient.

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

Grad-SAM

The Grad-SAM score *r* of the *i*'th input token x_i is the sum over all sequence elements *j*, attention heads *m*, and encoder layers *l*, of

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

: the Hadamard product between the attention values and the ReLU of the gradient.

The gradient is computed of the model prediction s_x with respect to the attention values,

$$G_x^{lm} := \frac{\partial s_x}{\partial A_x^{lm}}$$

In one sentence, what does this gradient tell us?

Grad-SAM

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

The Grad-SAM score r of the i'th input token x_i is the sum over all sequence elements j, attention heads m, and encoder layers l, of

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

: the Hadamard product between the attention values and the ReLU of the gradient.

The gradient is computed of the model prediction s_x with respect to the attention values,

$$G_x^{lm} := \frac{\partial s_x}{\partial A_x^{lm}}$$

Intuitively, this gradient tells us how sensitive the model prediction is to the attention values.

As before, we get gradients from torch :)

Get the gradients for Grad-SAM

```python
def get_grads_gradsam(model, attentions, logits):
    """Return the gradient of the output with respect to the attention weights for all heads in all layers."""

    n_layers = model.config.num_hidden_layers

    pred = get_model_pred(logits)
    logits = logits.squeeze()

    attention_grads = {layer_idx : None for layer_idx in range(n_layers)}

    # Retain gradients for attention matrices
    for att in attentions:
        att.retain_grad()

    model.zero_grad()

    # Backward from the model prediction
    logits[pred].backward(retain_graph=True)

    # Collect gradients for each layer
    for layer_idx, att in enumerate(attentions):
        grad = att.grad.squeeze().clone() # Shape (n_heads, seq_len, seq_len)
        attention_grads[layer_idx] = grad

    return attention_grads
```

attentions: list of attention values (torch tensors) for each head and encoder layer

logits: activations of one instance (sentence), right before the pred softmax

Initialize gradient dictionary with None

Tell torch to retain gradients

Clear gradient before backward pass

Do a backward pass on the prediction of the data point

Collect and store gradients from each encoder layer in a dictionary

Grad-SAM on BERT predictions for sentiment classification

```python
def grad_sam(model, attentions, logits):
    """Returns the Grad-SAM score for all tokens in an input sequence."""

    model = model.to(device)

    n_heads = model.config.num_attention_heads
    n_layers = model.config.num_hidden_layers

    attention_grads = get_grads_gradsam(model, attentions, logits)

    seq_len = attentions[0].size(-1) # CLS and SEP tokens are included

    layer_vectors = []

    for layer_idx, grad in attention_grads.items():
        relu_grad = F.relu(grad.to(device)) # Shape (n_heads, seq_len, seq_len)

        attention = attentions[layer_idx].squeeze().clone().to(device) # Shape (n_heads, seq_len, seq_len)

        hadamard_product = attention * relu_grad

        # Sum attention weights for each head (sum over columns)
        summed_columns = hadamard_product.sum(dim=-1, keepdim=False) # Shape (n_heads, seq_len)

        # Sum over all heads
        summed_heads = summed_columns.sum(dim=0, keepdim=False) # Shape (seq_len)

        layer_vectors.append(summed_heads)

    grad_sam_scores = sum(layer_vectors)
    grad_sam_scores = grad_sam_scores / (n_layers*n_heads*seq_len)

    return grad_sam_scores
```

Compute Grad-SAM scores (see equation)

Get gradients of the model output with respect to all attention values.

Compute the Hadamard product of attention values and the gradient (with ReLU).

Sum over N
Sum over M

(end for)

Sum over L, normalize and return

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

Grad-SAM function pseudocode

```
move model to device
seq_len = total sequence length N, including CLS and SEP tokens
n_heads = number of  attention heads M
n_layers = number of hidden layers L
attention_gradients = get_grads_gradsam(model, attentions, logits)
initialize empty layer_vectors list

for each layer index and gradient matrix in attention_gradients:
      relu_grad = ReLU(gradient matrix)
      attention_matrix = attentions for the layer index
      Hadamard product of attention_matric and relu_grad

      sum Hadamard product over attention columns j (to N)
      sum result over all heads m (to M)
      append sum to layer_vectors

grad_sam_scores = sum over layer_vectors l (to L)
normalise grad_sam_scores

return grad_sam_scores
```

$$H_x^{lm} = A_x^{lm} \circ \mathrm{ReLU}(G_x^{lm})$$

$$r_{x_i} = \frac{1}{LMN} \sum_{l=1}^{L} \sum_{m=1}^{M} \sum_{j=1}^{N} [H_x^{lm}]_{ij}$$

BERT predictions for sentiment classification

```
tokenizer = AutoTokenizer.from_pretrained("textattack/bert-base-uncased-SST-2")
model = AutoModelForSequenceClassification.from_pretrained("textattack/bert-base-uncased-SST-2").to(device)
model.eval()
```

Load tokenizer and BERT model (fine-tuned for binary sentiment classification) from Hugging Face.

```
def tokenize_input(tokenizer, input_sentence):
    """Tokenize the input sentence."""
    tokenizer.truncation_side = "right"
    inputs = tokenizer(input_sentence, return_tensors="pt", truncation=True, padding=True)
    return inputs
```

Tokenize input sentence (string) and add special tokens [CLS] and [SEP].

```
sentence = "I like the movie."
input = tokenize_input(tokenizer, sentence)
output = model(**input, output_attentions=True)
logits = output.logits
attentions = output.attentions
```

Tokenize input and run forward pass to get output logits and attention values from all heads and layers.

BERT predictions for sentiment classification

```python
for sentence in ["I like the movie",
                 "She hates the book",
                 "This is so boring",
                 "I cannot wait for next time",
                 "You look extremely confused"]:
    # Return the tokenizer output for the input sentence, which includes the tokenized sentences with
    # added CLS and SEP tokens and the attention masks
    input = tokenize_input(tokenizer, sentence).to(device)
    output = model(**input, output_attentions=True) # Forward pass
    logits = output.logits
    pred = get_model_pred(logits)

    # Get attention matrices for all heads and layers
    attentions = output.attentions
    # attentions is a tuple with tensor elements of shape
    # (batch size, num attention heads, sequence length, sequence length),
    # one tensor per layer (i.e., encoder block)
    print(f"The sentiment of the sentence '{sentence}' is {pred}")
```

```
The sentiment of the sentence 'I like the movie' is 1
The sentiment of the sentence 'She hates the book' is 0
The sentiment of the sentence 'This is so boring' is 0
The sentiment of the sentence 'I cannot wait for next time' is 1
The sentiment of the sentence 'You look extremely confused' is 0
```

These predictions all seem reasonable, but how did the various tokens contribute?

Let's calculate the Grad-SAM scores

Grad-SAM scores for BERT sentiment classification

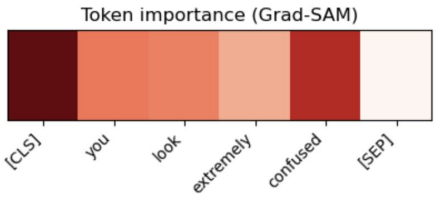
```
The sentiment of 'You look extremely confused' is 0
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00099815
2. 'confused': 0.00080487
3. 'you': 0.00054431
4. 'look': 0.00051781
5. 'extremely': 0.00039444
6. '[SEP]': 0.00013036
```

```
plot_token_importance(tokens, grad_sam_scores)
```



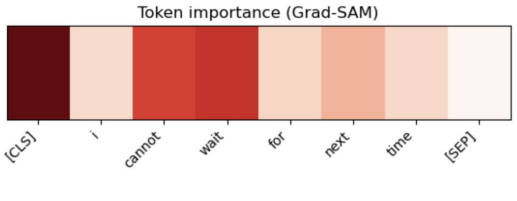

```
The sentiment of 'I cannot wait for next time' is 1
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00215395
2. 'wait': 0.00163126
3. 'cannot': 0.00151768
4. 'next': 0.00080023
5. 'for': 0.00057652
6. 'time': 0.00056583
7. 'i': 0.00055223
8. '[SEP]': 0.00027473
```

```
plot_token_importance(tokens, grad_sam_scores)
```



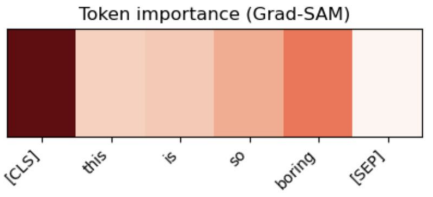

```
The sentiment of 'This is so boring' is 0
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00047818
2. 'boring': 0.00025618
3. 'so': 0.00017947
4. 'is': 0.00013792
5. 'this': 0.00012859
6. '[SEP]': 0.00005025
```

```
plot_token_importance(tokens, grad_sam_scores)
```



What do we see?

Which token is always important?

What does "token importance" mean here?

Grad-SAM scores for BERT sentiment classification

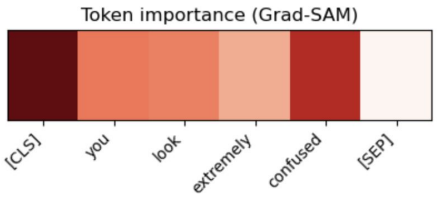
```
The sentiment of 'You look extremely confused' is 0
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00099815
2. 'confused': 0.00080487
3. 'you': 0.00054431
4. 'look': 0.00051781
5. 'extremely': 0.00039444
6. '[SEP]': 0.00013036
```

```
plot_token_importance(tokens, grad_sam_scores)
```



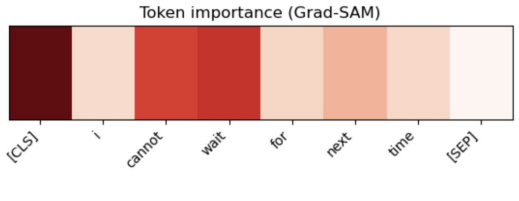

```
The sentiment of 'I cannot wait for next time' is 1
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00215395
2. 'wait': 0.00163126
3. 'cannot': 0.00151768
4. 'next': 0.00080023
5. 'for': 0.00057652
6. 'time': 0.00056583
7. 'i': 0.00055223
8. '[SEP]': 0.00027473
```

```
plot_token_importance(tokens, grad_sam_scores)
```



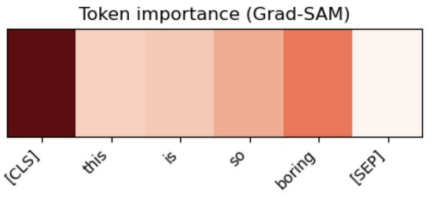

```
The sentiment of 'This is so boring' is 0
```

```
grad_sam_scores = grad_sam(model, attentions, logits).detach().cpu().numpy()
```

```
tokens = tokenizer.convert_ids_to_tokens(input["input_ids"][0])
display_token_importance(tokens, grad_sam_scores)
```

```
1. '[CLS]': 0.00047818
2. 'boring': 0.00025618
3. 'so': 0.00017947
4. 'is': 0.00013792
5. 'this': 0.00012859
6. '[SEP]': 0.00005025
```

```
plot_token_importance(tokens, grad_sam_scores)
```



The classification token is always important in this classification task (not too surprising)

The Grad-SAM score is the "attention times gradient".

Token importance according to Grad-SAM means that the token *receives much attention* from other tokens, *and* that the model prediction is sensitive to the token's attention.

Grad-SAM

Where do we place this in the taxonomy?

Post-hoc?

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Grad-SAM

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour.*

Model-agnostic or model-specific?

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Grad-SAM

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour.*

Model-specific: *We use the attention values and gradients of the specific model. We can only use this method on transformer models.*

Local or global?

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Grad-SAM

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour.*

Model-specific: *We use the attention values and gradients of the specific model. We can only use this method on transformer models.*

Local: *The explanation is given in terms of importance attribution per input token.*

Post-hoc methods	Model-agnostic	Model-specific
Local		
Global		

Grad-SAM

Where do we place this in the taxonomy?

Post-hoc: *We got a model and want to explain its behaviour.*

Model-specific: *We use the attention values and gradients of the specific model. We can only use this method on transformer models.*

Local: *The explanation is given in terms of importance attribution per input token.*

Post-hoc methods	Model-agnostic	Model-specific
Local		Grad-SAM
Global		

We will look at three XAI methods for transformers:

1. Grad-SAM (this one is new to us) - for sentiment classification with BERT
2. SHAP (we already know this one) - for sentiment classification with BERT
3. TCAV (we already know this one) - for image classification with a ViT

SHAP for BERT sentiment classification

Do you remember SHAP?

What does the SHAP value of a feature tell us?

SHAP for BERT sentiment classification

SHAP values are perturbation-based input feature importances that quantify the contributions to the model prediction's deviation from the expected prediction across a background dataset.

In short: A feature's SHAP value tells us how much the feature contributed to drive the model prediction away from the average prediction, on some background dataset.

A large (absolute) SHAP value indicates an important feature.

SHAP for BERT sentiment classification

For text, SHAP expects a prediction function that returns the logits of the positive class. For our model and tokenizer, we can do this as follows

```
"""Implementation following https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/text.html"""

def f(x):
    # Tokenize all strings in input
    tv = torch.tensor([tokenizer(v, truncation=True,
                                 max_length=tokenizer.model_max_length,
                                 padding="max_length")["input_ids"] for v in x], device=device)

    # Fix padding stuff when input consist of several sentences
    attention_masks = torch.tensor([tokenizer(v, truncation=True, max_length=tokenizer.model_max_length,
                                              padding="max_length")["attention_mask"] for v in x], device=device)

    # Get model prediction (logits) and return the positive class
    outputs = model(input_ids=tv, attention_mask=attention_masks)[0].detach().cpu().numpy()

    return np.array(outputs[:,1]) # Positive class logits
```

Please don't let this confuse you if you're not familiar with transformers.

The important part is: `f` is needed by SHAP to get the logits from the model prediction - and this function must do the tokenization of the inputs, or SHAP gets confused.

SHAP for BERT sentiment classification

With the function `f`, we can now calculate SHAP values using the shap library (almost) as before

```
explainer = shap.Explainer(f, tokenizer)
shap_values = explainer(sentences, fixed_context=1)
```

```
shap_values
```

```
.values =
array([array([-0.1295938 ,  0.56973486,  2.80113666, -0.14384789,  0.32744559,
              0.17642172])                                                    ,
       array([ 0.04401805,  0.04553582, -3.53780551, -0.05805362,  0.00513234,
              0.29313125])                                                    ,
       array([ 0.30843912,  0.25726233,  0.28962884,  0.28991709, -4.94576939,
              0.3164889 ])                                                    ,
       array([-0.10001853,  0.30019138,  1.07011396,  1.28322518,  0.11099461,
              0.42985061,  0.22122911, -0.22297784])                          ,
       array([ 0.59630368,  0.48589848,  0.27857004, -0.05426301, -4.83736869,
              0.56557564])                                                   ],
      dtype=object)

.base_values =
array([ 0.06226253,  0.06226253,  0.06226253, -0.42705753,  0.06226253])

.data =
(array(['', 'I ', 'like ', 'the ', 'movie', ''], dtype=object), array(['', 'She ', 'hates ', 'the ', 'book', ''], dtype=object), array
(['', 'This ', 'is ', 'so ', 'boring', ''], dtype=object), array(['', 'I ', 'cannot ', 'wait ', 'for ', 'next ', 'time', ''],
      dtype=object), array(['', 'You ', 'look ', 'extremely ', 'confused', ''], dtype=object))
```

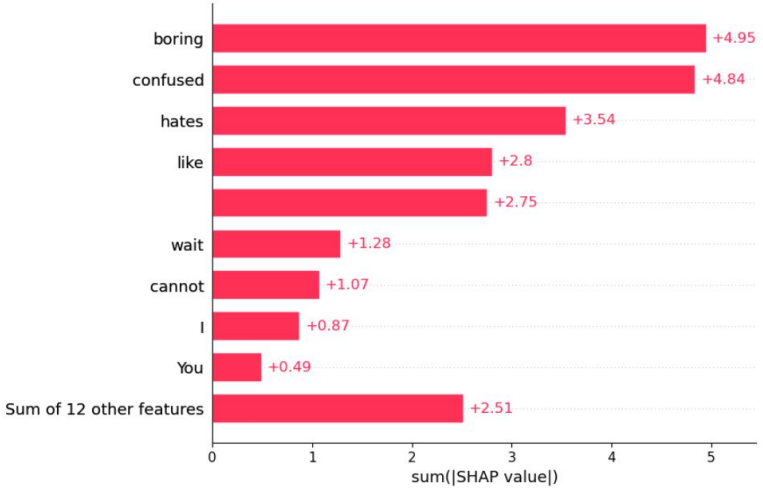
We can study these values together by making a bar plot of the absolute SHAP values, for example

SHAP for BERT sentiment classification

Which words got the highest SHAP values for our five sentences?

```
(array(['', 'I ', 'like ', 'the ', 'movie', ''], dtype=object), array(['', 'She ', 'hates ', 'the ', 'book', ''], dtype=object), array
(['', 'This ', 'is ', 'so ', 'boring', ''], dtype=object), array(['', 'I ', 'cannot ', 'wait ', 'for ', 'next ', 'time', ''],
      dtype=object), array(['', 'You ', 'look ', 'extremely ', 'confused', ''], dtype=object))
```

```
shap.plots.bar(shap_values.abs.sum(0))
```



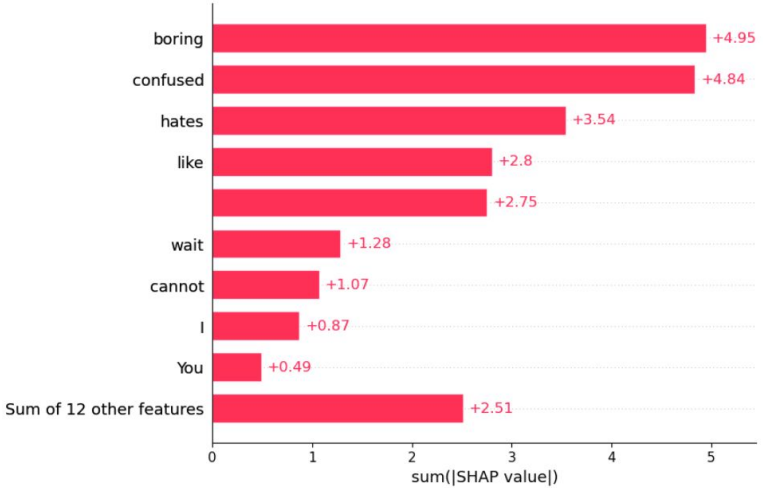
Does this make sense?

SHAP for BERT sentiment classification

Which words got the highest SHAP values for our five sentences?

```
(array(['', 'I ', 'like ', 'the ', 'movie', ''], dtype=object), array(['', 'She ', 'hates ', 'the ', 'book', ''], dtype=object), array
(['', 'This ', 'is ', 'so ', 'boring', ''], dtype=object), array(['', 'I ', 'cannot ', 'wait ', 'for ', 'next ', 'time', ''],
      dtype=object), array(['', 'You ', 'look ', 'extremely ', 'confused', ''], dtype=object))
```

```
shap.plots.bar(shap_values.abs.sum(0))
```



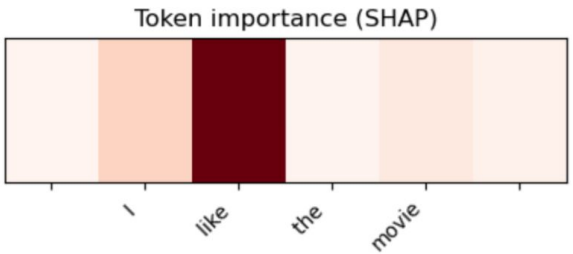
'boring' , 'confused' and 'hates'.

It does make sense that these are important in a sentiment classification setting.

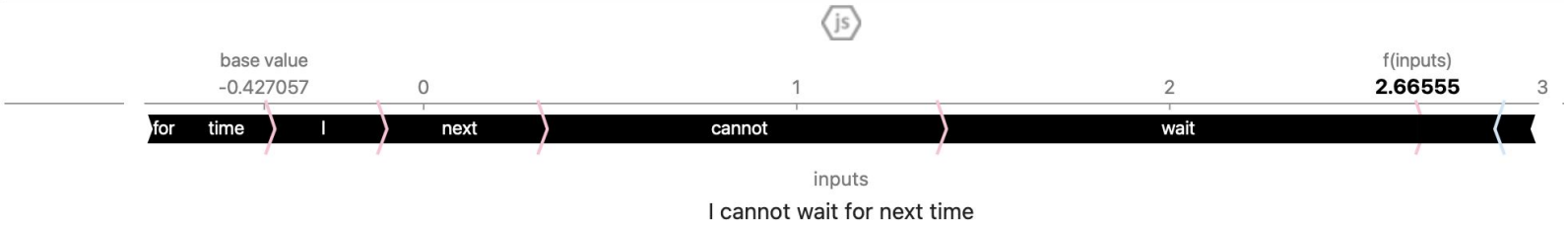
SHAP force plot and absolute values for single data points

```
i = 0
shap.plots.initjs()
shap.plots.text(shap_values[i])
```


```
plot_token_importance(shap_values[i].data, np.abs(shap_values[i].values), title="Token importance (SHAP)")
```



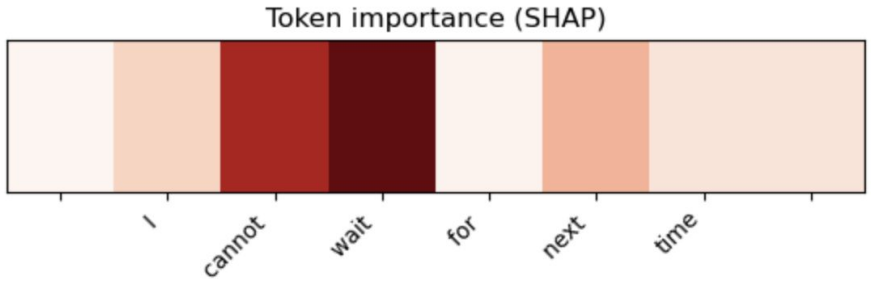
```
i = 3
shap.plots.initjs()
shap.plots.text(shap_values[i])
```



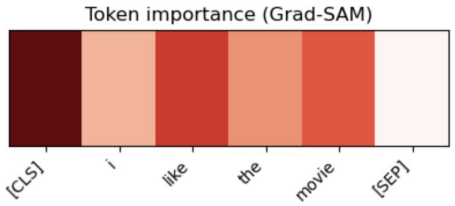

```
print(shap_values[i])
```

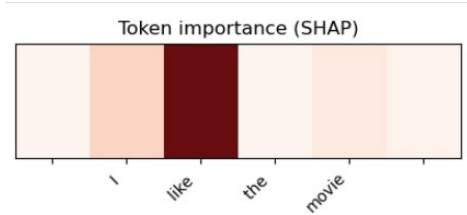
```
.values =
array([-0.10001764,  0.30019084,  1.07011516,  1.28322519,  0.11099668,
        0.42984768,  0.2212288 , -0.22297816])

.base_values =
np.float64(-0.4270571172237396)

.data =
array(['', 'I ', 'cannot ', 'wait ', 'for ', 'next ', 'time', ''],
      dtype=object)
```

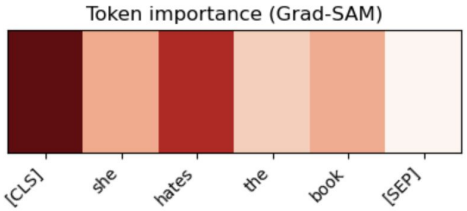
```
plot_token_importance(shap_values[i].data, np.abs(shap_values[i].values), title="Token importance (SHAP)")
```

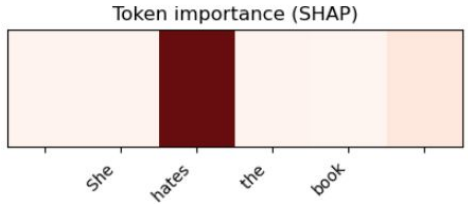


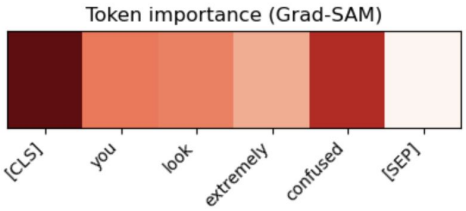
SHAP vs Grad-SAM for BERT sentiment classification

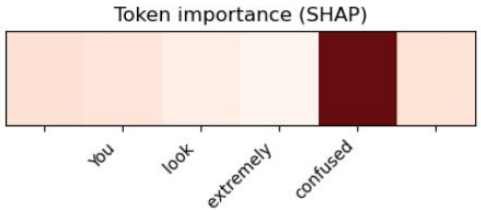












Note that these are absolute SHAP values.

Do the Grad-SAM and SHAP values agree?

What's the largest discrepancy?

Why do you think this is?

SHAP vs Grad-SAM for BERT sentiment classification

Grad-SAM and SHAP measure completely different notions of importance.
The CLS token almost always ends up “important” in this gradient-based method, but not in SHAP.

Part of the homework is trying to explain why this is as expected :)

We will look at three XAI methods for transformers:

1. Grad-SAM (this one is new to us) - for sentiment classification with BERT
2. SHAP (we already know this one) - for sentiment classification with BERT
3. TCAV (we already know this one) - for image classification with a ViT

ViT - Transformers for image classification

Dosovitskiy et al (2021) proposed **ViT (Vision Transformer)**, an adaptation of the original transformer by Vaswani et al (2017) for image classification.

The model architecture is a transformer encoder with an MLP (=linear layer) classification head.

The model is pre-trained on large image datasets and fine-tuned to (smaller) downstream tasks.

The input image is divided into patches which are converted to embedding vectors through a linear projection combined with positional information.

ViT uses a classification embedding at the beginning of the sequence, similar as BERT.

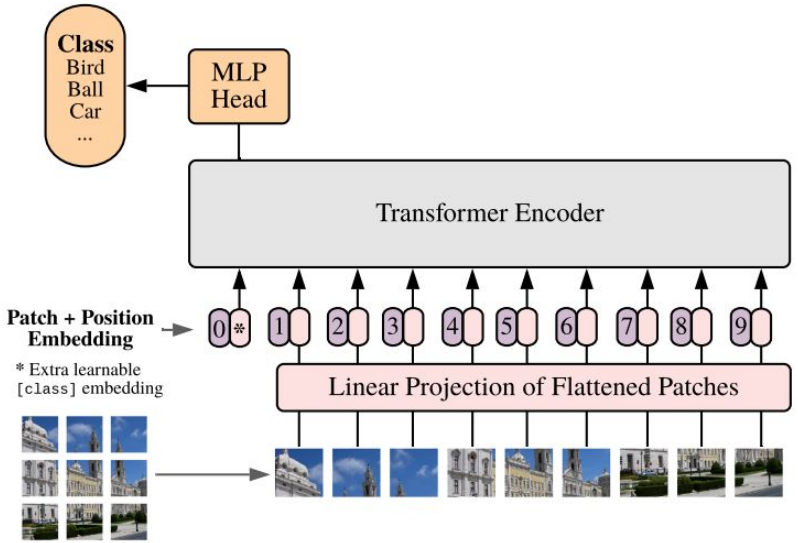


Figure from Dosovitskiy et al (2021).

TCAV for explaining ViT

Do you remember concept detection? What is a concept activation vector?

What does the TCAV score of a concept tell us?

TCAV for explaining ViT

A concept activation vector is the normal vector to the decision surface separating a user-defined concept from another concept in the activation space of a model's internal layer.

TCAV uses directional derivatives along the CAV to quantify the degree to which the corresponding concept affects the model's prediction (classification).

In short: TCAV counts how often increased presence of a specific concept contributes to more instances being predicted to a given class.

TCAV for explaining ViT - in short

```
processor = ViTImageProcessor.from_pretrained('google/vit-base-patch16-224')
vit = ViTForImageClassification.from_pretrained('google/vit-base-patch16-224').to(device)
vit.eval()
```

Load processor and model from Hugging Face

```
label2id = vit.config.label2id
class_idx = label2id.get(target_class)
```

Get class index of the target class (e.g., zebra)

```
positive_loader = get_vit_dataloader(p_concept_paths, processor, batch_size=batch_size)
negative_loader = get_vit_dataloader(n_concept_paths, processor, batch_size=batch_size)
target_loader = get_vit_dataloader(target_paths, processor, batch_size=batch_size)
```

Get data loaders for concept and target images

```
layer = vit.vit.encoder.layer[10]
```

Choose model layer

```
positive_activations = flatten(get_activations(vit, positive_loader, device, layer))
negative_activations = flatten(get_activations(vit, negative_loader, device, layer))
```

Get activations for positive and negative concept images

```
cav = train_cav(positive_activations, negative_activations)
```

Compute the CAV using a logistic regressor

```
grads = flatten(get_grads(vit, target_loader, device, layer, target_idx=class_idx))
tcav_score = tcav(cav, grads)
```

Get gradients for the target images
Compute TCAV score

TCAV for explaining ViT - in short

The functionality for generating CAVs and calculating the TCAV score is implemented in the file `tcav.py`

Conceptually, this is exactly the same as we learned in the lecture on concept based explanations. There are small adaptations of the code to the `huggingface` model.

You can play around with the code, and verify that the TCAV score for

- "striped" vs "dotted" is ~0.95,
- "bubbly" vs "striped" is ~0.1
- "dotted" vs "bubbly" is ~0.05
- "striped" vs "bubbly" is ~0.95

Do these make sense for zebra classification?

Interpreting the attention mechanism

Interpreting attention

Transformers work thanks to the attention mechanism, and its behaviour is largely determined by the attention *weights*.

As always, the model's extracted knowledge is represented in its weights, but attention weights have an explicit interpretation, namely how much each token attends to all tokens.

Interpreting attention

Transformers work thanks to the attention mechanism, and its behaviour is largely determined by the attention *weights*.

As always, the model's extracted knowledge is represented in its weights, but attention weights have an explicit interpretation, namely how much each token attends to all tokens.

Do you think there is any local or global information for interpretation or explanation here?

Interpreting attention

Transformers work thanks to the attention mechanism, and its behaviour is largely determined by the attention *weights*.

As always, the model's extracted knowledge is represented in its weights, but attention weights have an explicit interpretation, namely how much each token attends to all tokens.

- The interpretation thus works at a fine-grained (token-to-token) level, providing a local interpretation via the activations (attention weights) for one data point.

- The attention weights also contain global information (about the entire model), as they are obtained from Q and K, which are based on the embeddings, learned from the whole training data set.

Interpreting attention

Transformers work thanks to the attention mechanism, and its behaviour is largely determined by the attention *weights*.

As always, the model's extracted knowledge is represented in its weights, but attention weights have an explicit interpretation, namely how much each token attends to all tokens.

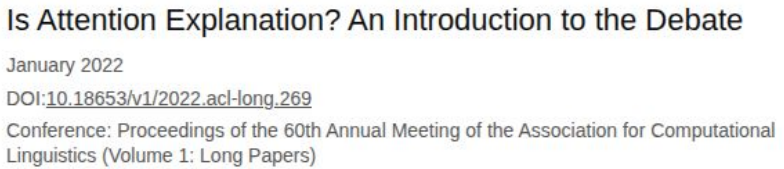
- The interpretation thus works at a fine-grained (token-to-token) level, providing a local interpretation via the activations (attention weights) for one data point.

- The attention weights also contain global information (about the entire model), as they are obtained from Q and K, which are based on the embeddings, learned from the whole training data set.

However, the explanatory power of the attention mechanism is debated.

Interpreting attention

Yes, there is a *debate*.


Is Attention Explanation? An Introduction to the Debate

January 2022

DOI:10.18653/v1/2022.acl-long.269

Conference: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)

Briefly summarized, the two sides say:

- "Inputs accorded high attention weights are not necessarily responsible for model outputs" (attention is not attribution)

- "Attention weights can be used to predict the model output, meaning they encode information" (attention contains information)

Let's try to understand the two sides.

“Attention is not Explanation”

Attention is not Explanation is a paper published by Jain et al (2019), arguing that “**standard attention modules do not provide meaningful explanations and should not be treated as though they do**”.

More nuanced: The ability of attention modules to provide transparency for model predictions is (*in the sense of pointing to inputs responsible for outputs*) questionable.

They evaluate whether “input units (e.g., words) accorded high attention weights are responsible for model outputs”.

“Attention is not Explanation”

Attention is not Explanation is a paper published by Jain et al (2019), showing that “**standard attention modules do not provide meaningful explanations and should not be treated as though they do**”.

More nuanced: The ability of attention modules to provide transparency for model predictions is (*in the sense of pointing to inputs responsible for outputs*) questionable.

They evaluate whether “input units (e.g., words) accorded high attention weights are responsible for model outputs”.

They state that, assuming attention provides an explanation for model prediction, the following two properties should hold:

1) Attention weights should **correlate** with other measures of feature importance.
2) Alternative or **counterfactual** attention weight configurations should yield corresponding changes in the model’s prediction.

They report that neither property is consistently observed in the context of several NLP tasks when **RNN encoders** are used.

“Attention is not Explanation”

Property 1: Attention weights should **correlate** with other measures of feature importance.

They analyze the correlation between learned attention weights and 1) gradient-based feature importance and 2) ‘leave-one-out’ (LOO) measures.

Results: Attention weights do not strongly or consistently agree with such feature importance scores when using a BiRNN encoder.

Do you see any weaknesses in demanding property 1, or in the analysis?

“Attention is not Explanation”

Property 1: Attention weights should **correlate** with other measures of feature importance.

They analyze the correlation between learned attention weights and 1) gradient-based feature importance and 2) ‘leave-one-out’ (LOO) measures.

Results: Attention weights do not strongly or consistently agree with such feature importance scores when using a BiRNN encoder.

Limitations:

1. Feature importance measures do not necessarily represent the ground truth.
2. The analysis is done using an RNN encoder, so we cannot know that it applies to transformers.
3. …

“Attention is not Explanation”

Property 2: Alternative or **counterfactual** attention weight configurations should yield corresponding changes in the model’s prediction.

Idea: The prediction should change if the model attends to different input tokens.

Counterfactual distributions are constructed by

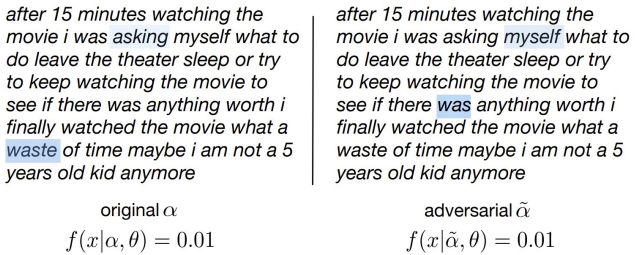
1) randomly permuting observed attention weights, or
2) adversarial attention weights that maximally differ from the observed attention weights but nonetheless yield an equivalent prediction (within some ε).

“Attention is not Explanation”

Property 2: Alternative or **counterfactual** attention weight configurations should yield corresponding changes in the model’s prediction.

Idea: The prediction should change if the model attends to different input tokens.

Counterfactual example:



Highlighted words correspond to large attention weights.

This shows that it is possible to achieve the same prediction (0.01 here) when the model attention weights (left) are very different from an adversarially constructed set of attention weights (right), for the same sample.

“Attention is not Explanation”

Property 2: Alternative or **counterfactual** attention weight configurations should yield corresponding changes in the model’s prediction.

Idea: The prediction should change if the model attends to different input tokens.

Result: “There exist many cases in which despite a high attention weight, an alternative and different attention configuration over inputs yields effectively the same output”.

“Attention is not not Explanation”

Then came a counter paper: *Attention is not not Explanation,* by Wiegreffe et al (2019)

“Attention is not not Explanation”

Attention is not not Explanation by Wiegreffe et al (2019) challenges the assumptions of Jain et al (2019). They accept property 1, but not property 2, and through experiments conclude that “**the prior work does not disprove the usefulness of attention mechanisms for explainability**”.

Jain et al argue that the violation of property 2, i.e., the fact that

it is possible to find adversarial counterfactual attention weights that maximally differ from the observed attention weights but nonetheless yield an equivalent prediction (within some ε)

does not make sense as an argument *against* attention weights for explanation / interpretation.

Can you think of why?

“Attention is not not Explanation”

Jain et al (2019):
If alternative attention distributions produce similar results to those obtained by the original model, then the original model’s attention scores cannot be reliably used to faithfully explain the model’s prediction.

Wiegreffe et al (2019):
Adversarial attentions cannot be treated as equally plausible or faithful explanations for model prediction.

The original attention weights are not assigned arbitrarily but rather trained alongside the rest of the model; the model components depend on each other. The attentions weights cannot just be changed independently.

The motivation for using attention weights for explanation is that they are a result of the model’s training. Finding alternative distributions resulting in similar predictions invalidates this motivation.

“Attention is not not Explanation”

Idea: Instead, examine the predictive power of attention distributions in a model with no access to neighbouring tokens of the instance.

Create diagnostic model which has an embedding (like the original model), but has an MLP instead of an attention mechanism.

Use different weights between embedding and output;

1) **Uniform**: outputs are equally likely for each instance;
2) **Trained MLP**: the MLP learns its own parameters;
3) **Base LSTM**: weights learned by the original model’s attention layer
4) **Adversary**: weights based on distributions found adversarially.

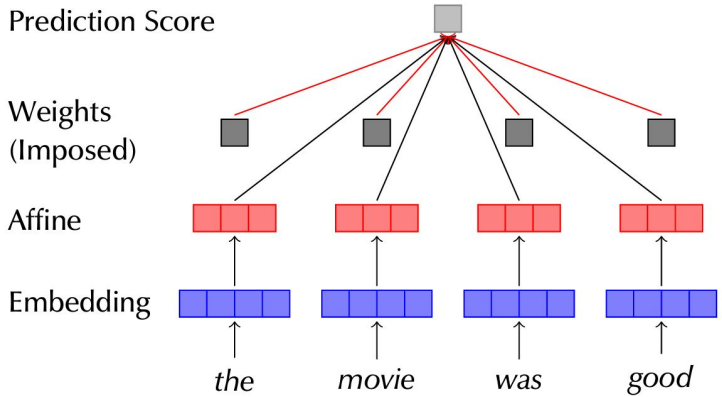


Figure from Wiegreffe et al (2019).

“Attention is not not Explanation”

Idea: Instead, examine the predictive power of attention distributions in a model with no access to neighbouring tokens of the instance.

Create diagnostic model which has an embedding (like the original model), but has an MLP instead of an attention mechanism.

If pre-trained scores from an attention model perform well, this suggest that they are helpful and consistent, fulfilling a certain sense of explainability.

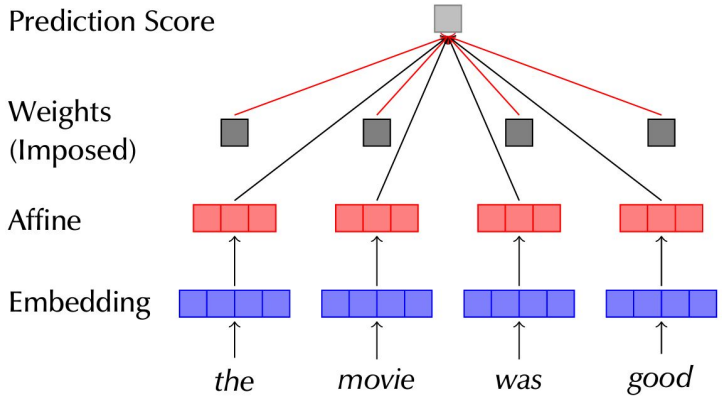


Figure from Wiegreffe et al (2019).

“Attention is not not Explanation”

Results: The pre-trained attention weights (the base LSTM) are more useful than the MLP learned weights, which are again more useful than the baseline.

Independent token-level models that have no access to contextual information find attention weights useful, indicating that they encode some measure of token importance which is not model-dependent.

Adversarially-trained attention distributions in the diagnostic setup perform poorly compared to traditional attention mechanisms, indicating that trained attention mechanisms (in RNNs on the tested datasets) learn meaningful relationships between tokens and prediction.

⇒ “Whether or not attention is explanation depends on the definition of explainability one is looking for.”

Whether or not (something) is an explanation depends on the definition of explainability one is looking for.

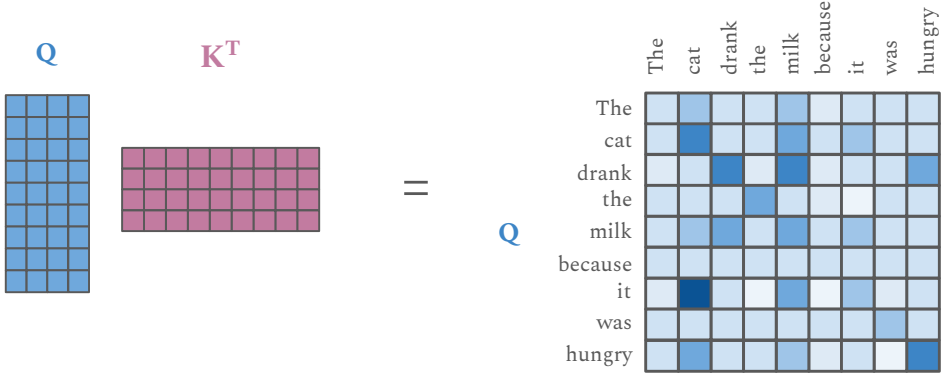
This holds in all of XAI :)

Homework :)

1) Implement Grad-SAM based on the equations and pseudocode.
2) Choose a few sentences, calculate SHAP values for the BERT classification and interpret the result. Compare to the Grad-SAM values of the same sentences and discuss why the two differ. In particular, what happens to the CLS token?
3) Choose two layers and plot the attention weights for one of the heads in each layer. Suggest an interpretation of each of the heatmaps.

Technicality regarding Grad-SAM implementation

Recall the attention equation: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$



Matrix columns are K and rows Q.
Entries are (Q_{id} K_{dj})

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

Technicality regarding Grad-SAM implementation

In Grad-SAM, we do three sums: $\frac{1}{LMN}\sum_{l=1}^{L}\sum_{m=1}^{M}\sum_{j=1}^{N}[H_x^{lm}]_{ij}$

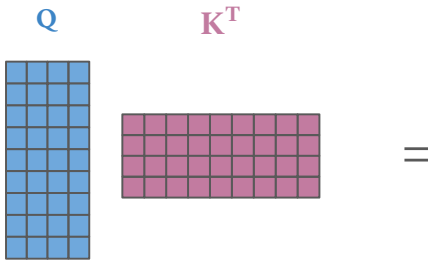
Where the innermost is over the sequence elements, meaning the columns, of $H_x^{lm} = A_x^{lm} \circ \text{ReLU}(G_x^{lm})$

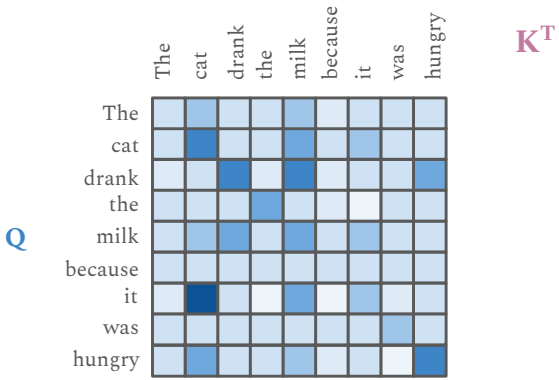
However, Grad-SAM defines attention as

$$A_x^{lm} = \text{softmax}\left(\frac{(W_q^{lm}U_x^{l-1})^T W_k^{lm}U_x^{l-1}}{\sqrt{d_a}}\right)$$

This, Q^TK, is different from Vaswani / BERT who use QK^T.

It's a result of the definition of the input dimensions. What's important is that A has dimension (seq_len, seq_len), and that the inner sum over *j* is over the row corresponding to the token's query.





Ktnx.

SHAP for explaining BERT predictions for sentiment classification

```python
def shap_values(model, tokenizer, sentence):
    """Implementation from https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/text.html"""
    sentence = [sentence]

    def f(x):
        tv = torch.tensor([tokenizer(v, truncation=True, max_length=tokenizer.model_max_length, padding="max_length")["input_ids"]
                           for v in x], device=device)
        attention_masks = torch.tensor([tokenizer(v, truncation=True, max_length=tokenizer.model_max_length, padding="max_length")["attention_mask"]
                                        for v in x], device=device)
        outputs = model(input_ids=tv, attention_mask=attention_masks)[0].detach().cpu().numpy()
        scores = (np.exp(outputs).T / np.exp(outputs).sum(-1)).T
        val = sp.special.logit(scores[:, 1])
        return val

    explainer = shap.Explainer(f, tokenizer)
    shap_values = explainer(sentence, fixed_context=1)

    return shap_values
```

Input: tokenizer, model and sentence to explain.

Function f (required by SHAP): tokenize the input, (fix padding stuff), get model output. Convert the output logits to probabilities (softmax), map the probabilities (0, 1) to the whole real line $(-\infty,+\infty)$. Return the corresponding value for the *positive* class.

Create explainer instance on the function f and the tokenizer, and run it on the specific sentence.