Million-to-trillion-parameter transformer models such as ChatGPT and GPT-4 appear to be impenetrable black boxes that nobody can interpret. As a result, many developers and users have sometimes been discouraged when dealing with these mind-blowing models. However, recent research has begun to solve the problem with cutting-edge tools to interpret the inner workings of a transformer model. Shattering the transformer black boxes will build trust between those who design them and those who implement and use them. It is beyond the scope of this book to describe all the interpretable AI methods and algorithms. Many systems exist, but our goal is not to explore them all. Instead, this chapter will focus on ready-to-use visual interfaces that provide insights for transformer model developers and users. The chapter begins by installing and running `BertViz` by *Jesse Vig*. Jesse Vig did quite an excellent job of building a visual interface that shows the activity in the attention heads of a BERT transformer model. BertViz interacts with the BERT models and provides a well-designed interactive interface we will implement. We will then interpret Hugging Face transformers with SHAP by building an interactive interface in Python.We will continue our journey through the layers of a BERT model with dictionary learning. The **Local Interpretable Model-agnostic Explanations** (**LIME**) approach provides practical functions to visualize how a transformer learns to understand language. The method shows that transformers often begin by learning a word, then the word in the context of the sentence, and finally, long-range dependencies.Finally, we will introduce other tools to investigate transformers' inner workings, such as Google's Language **Interpretability Tool** (**LIT**). LIT is a non-probing tool using PCA(Principal Component Analysis) or UMAP(Uniform Manifold Approximation and Projection) to represent transformer model predictions. OpenAI Large Language Models (LLMs) will take us

deeper and visualize the activity of a neuron in a transformer with an interactive interface. This approach opens the door to GPT-4 explaining a transformer, for example.By the end of the chapter, you will be able to interact with users to show visualizations of the activity of transformer models. Interpretation tools for transformers still have a long way to go. However, these nascent tools will help developers and users understand how transformer models work and increase their transparency.This chapter covers the following topics:

- Installing and running BertViz
- Running BertViz's interactive interface
- Interpreting Hugging Face transformer models with SHAP
- The difference between probing and non-probing methods
- A Principal Component Analysis (PCA) reminder
- Introducing LIME
- Running transformer visualization through dictionary learning
- Word-level polysemy disambiguation
- Visualizing low-level, mid-level, and high-level dependencies
- Visualizing key transformer factors
- Language Interpretability Tool (LIT)
- Visualizing the activity of transformer models with OpenAI LLMs

Our first step will begin by installing and running BertViz.

Transformer visualization with BertViz

Jesse Vig's article, *A Multiscale Visualization of Attention in the Transformer Model*, 2019, recognizes the effectiveness of transformer models. However, *Jesse Vig* explains that deciphering the attention mechanism is challenging. The paper describes the process of BertViz, a visualization tool.BertViz can visualize attention head activity and interpret a transformer model's behavior.BertViz was first designed to visualize BERT and GPT models. In this section, we will visualize the activity of a BERT model.Some tools mention the term "interpretable," stressing the "why" of an output.

Others use the term "explainable" to describe "how" an output is reached. Finally, some don't apply the nuance and use the terms loosely because "why" can sometimes mean "how" to explain why! We will use the terms loosely, as the tools in this chapter most often do.Let's now install and run BertViz.

Running BertViz

It only takes six steps to visualize and interact with transformer attention heads.Open the `BertViz.ipynb` notebook in the directory of this chapter in the GitHub repository of this book.The first step is to install `BertViz` and the requirements.

Step 1: Installing BertViz and importing the modules

The notebook installs `BertViz`, Hugging Face transformers, and the other basic requirements to implement the program:

```
!pip install bertViz
from bertViz import head_view, model_view
from transformers import BertTokenizer, BertModel
```

The **head_view** and **model_view** libraries are now imported. We will now load the BERT model and tokenizer.

Step 2: Load the models and retrieve attention

BertViz supports BERT, GPT, RoBERTa, and other models. You can consult BertViz on GitHub for more information: https://github.com/jessevig/BertViz.We fine-tuned a `'bert-base-uncased'` in *Chapter 5, Diving into Fine-Tuning through BERT.* In this section, we will also run a `bert-base-uncased` model and a pretrained tokenizer to deepen our understanding of its inner mechanisms:

```
# Load model and retrieve attention
model_version = 'bert-base-uncased'
do_lower_case = True
model = BertModel.from_pretrained(model_version, output_attentions=True)
tokenizer = BertTokenizer.from_pretrained(model_version, do_lower_case=do_lower_case)
```

We now enter our two sentences. You can try different sequences to analyze the behavior of the model. `sentence_b_start` will be necessary for *Step: 5 Model view*:

```
sentence_a = "A lot of people like animals so they adopt cats"
sentence_b = "A lot of people like animals so they adopt dogs"
inputs = tokenizer.encode_plus(sentence_a, sentence_b, return_tensors='pt',
add_special_tokens=True)
token_type_ids = inputs['token_type_ids']
input_ids = inputs['input_ids']
attention = model(input_ids, token_type_ids=token_type_ids)[-1]
sentence_b_start = token_type_ids[0].tolist().index(1)
input_id_list = input_ids[0].tolist() # Batch index 0
tokens = tokenizer.convert_ids_to_tokens(input_id_list)
```

Copy Explain

And that's it! We are ready to interact with the visualization interface.

Step 3: Head view

We have one final line to add to activate the visualization of the attention heads:

```
head_view(attention, tokens)
```

Copy Explain

The words of the first layer (layer 0) are not the actual tokens, but the interface is educational. The 12 attention heads of each layer are displayed in different colors. The default view is set to layer 0, as shown in *Figure 9.1*:

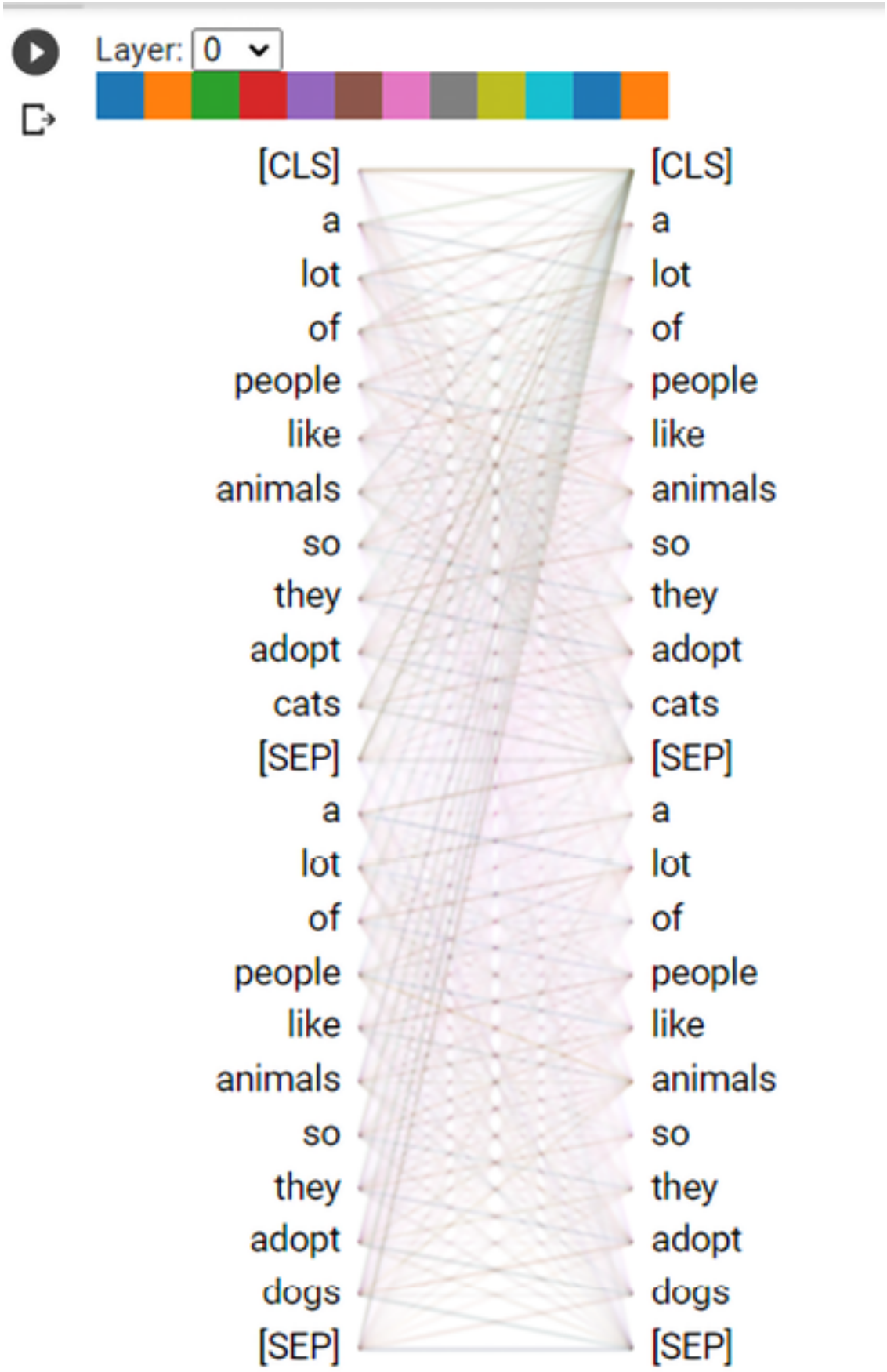

Figure 9.1: The visualization of attention heads

We are now ready to explore attention heads.

Step 4: Processing and displaying attention heads

Each color above the two columns of tokens represents an attention head of the layer number. Choose a layer number and click on an attention head (color).The words in the sentences are broken down into tokens in the attention. However, in this section, the word `tokens` loosely refers to `words` to help us understand how the transformer heads work.I focused on the word `animals` in *Figure 9.2*:

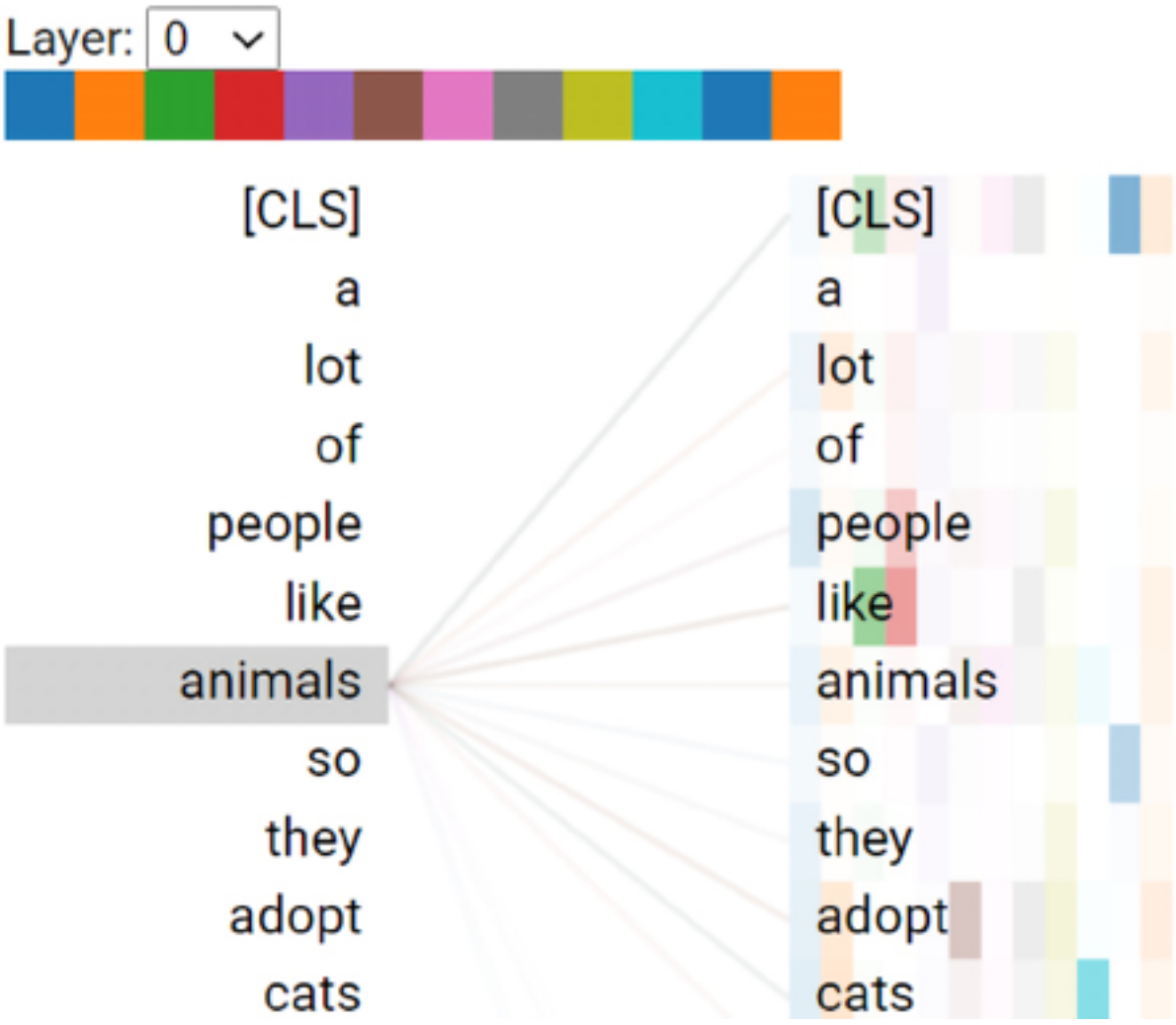

Figure 9.2: Selecting a layer, an attention head, and a token

`BertViz` shows that the model made a connection between `animals` and several words. This is normal since we are only at layer 0.Layer 1 begins to isolate words `animals` is related to, as shown in *Figure 9.3*:

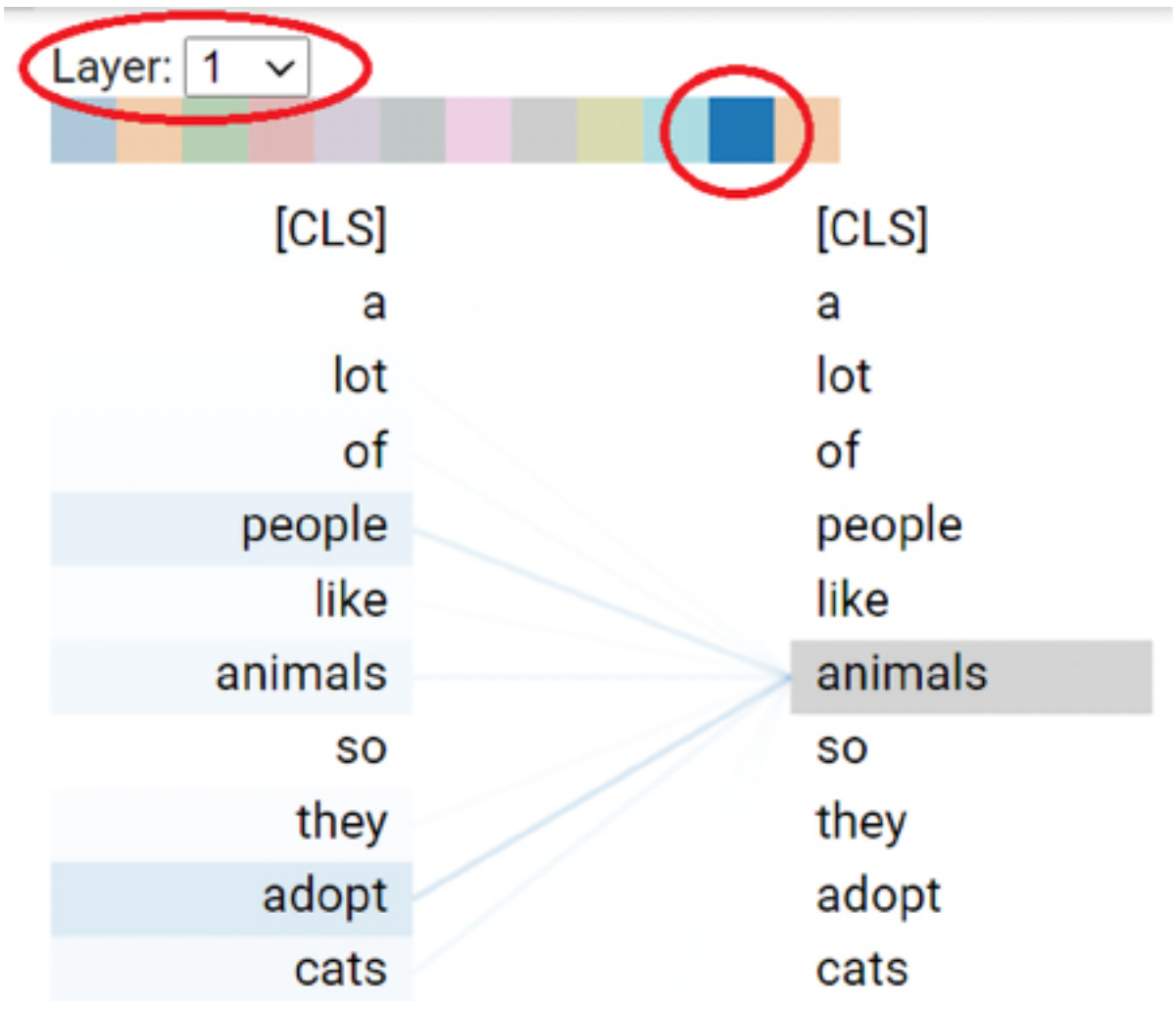

Figure 9.3: Visualizing the activity of attention head 11 of layer 1

Attention head 11 makes a connection between `animals`, `people`, and `adopt`.If we click on `cats`, some interesting connections are shown in *Figure 9.4*:

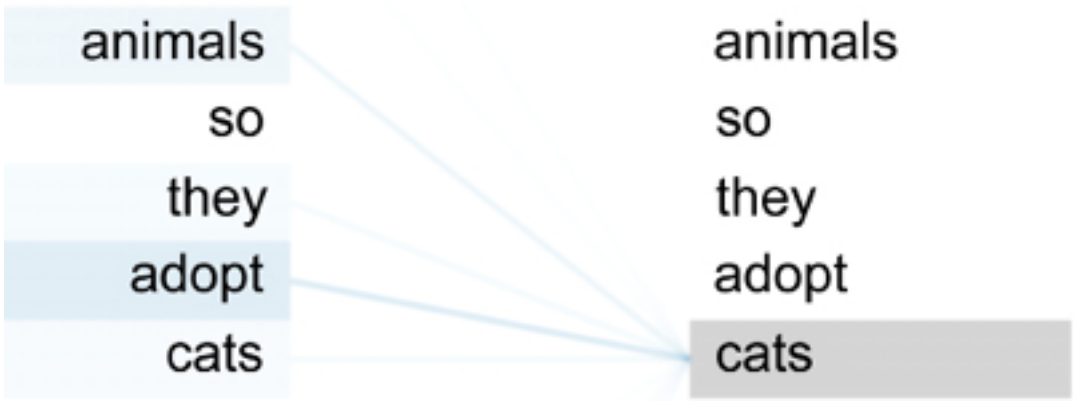

Figure 9.4: Visualizing the connections between cats and other tokens

The word `cats` is now associated with `animals`. This connection shows that the model is learning that cats are animals.You can change the sentences and then click on the layers and attention heads to visualize how the transformer makes connections. You will find limits, of course. The good and bad connections will show you how transformers work and fail. Both cases are valuable for explaining how transformers behave and why they require more layers, parameters, and data.Let's see how `BertViz` displays the model view.

Step 5: Model view

It only takes one line to obtain the model view of a transformer with `BertViz`:

```
model_view(attention, tokens, sentence_b_start)
```

`BertViz` displays all of the layers and heads in one view, as shown in the view excerpt in *Figure 9.5*:

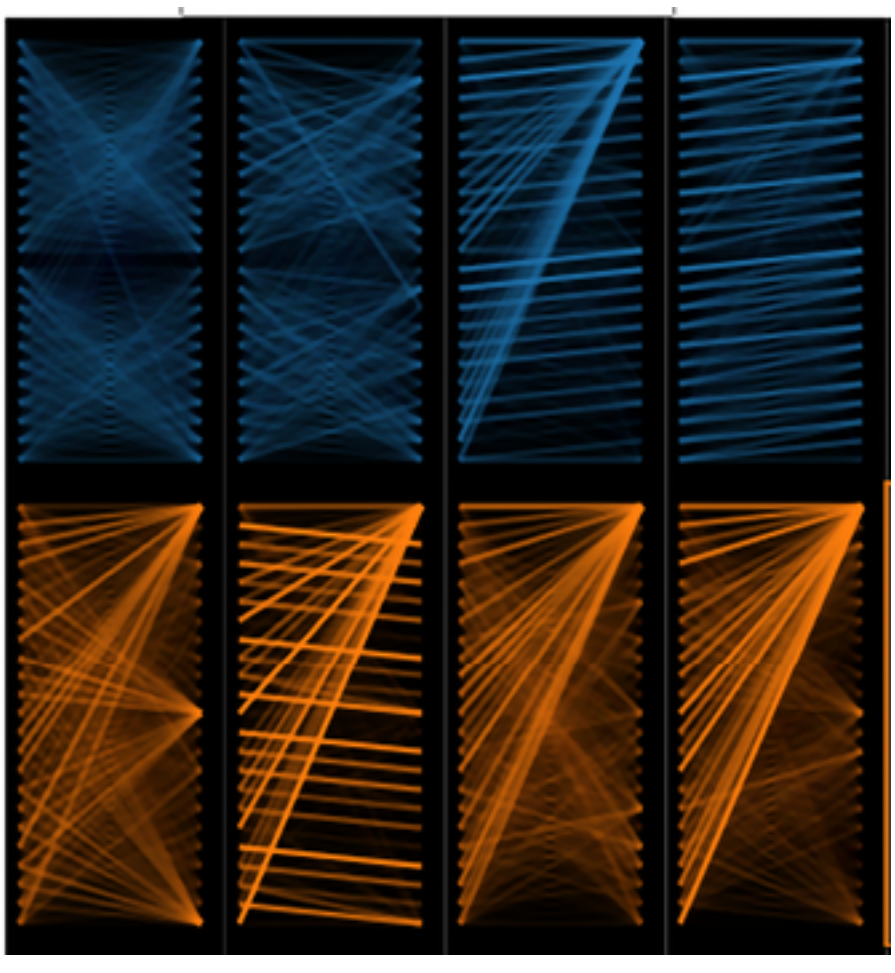

Figure 9.5: Model view mode of BertViz

If you click on one of the heads, you will obtain a head view with word-to-word and sentence-to-sentence options. You can then go through the attention heads to see how the transformer model makes better representations as it progresses through the layers. For example, *Figure 9.6* shows the activity of an attention head in the first layers:

Figure 9.6: Activity of an attention head in the lower layers of the model

This interactive interface offers insights into the activity of the attention heads. We will now dive deeper into the analysis of these attention heads.

Step 6: Displaying the output probabilities of attention heads

In this section, we will implement an interactive interface to analyze the scores of the output of the attention heads begun in *Chapter 3, Emergent vs Downstream Tasks: the Unseen Depths of Transformers*. The program first installs the Hugging Face library:

Copy Explain

```
#Installing Hugging Face transformers
!pip install transformers
We will be using  BertTokenizer and BertModel
from transformers import BertTokenizer, BertModel
```

Now, we add the interactive form and enter a sentence:

```
%%capture
input_text = "The output shows the attention values" #@param {type:"string"}
print(input_text)
```

The output is the string we entered. Once you have run through the cells, you can return to this form and enter any sequence you wish to explore.The code now defines the tokenizer and the model and begins by tokenizing the input.

```
# Load the BERT model and tokenizer
model_name = 'bert-base-uncased'
tokenizer = BertTokenizer.from_pretrained(model_name)
model = BertModel.from_pretrained(model_name, output_attentions=True)
# Tokenize the input text
tokens = tokenizer.tokenize(input_text)
input_ids = tokenizer.convert_tokens_to_ids(tokens)
```

We now access the attention outputs:

```
# Get the attention matrix
inputs = tokenizer.encode_plus(input_text, return_tensors='pt')
input_ids = inputs['input_ids']
attention_mask = inputs['attention_mask']
outputs = model(input_ids, attention_mask=attention_mask)
attentions = outputs.attentions
```

We can now stream the output of the attention heads.

Streaming the output of the attention heads

Now that we have retrieved the attention outputs, we can extract the content of each of the 12 layers and 12 heads of the model:

```
# Visualize the attention head activity
for layer, attention in enumerate(attentions):
    print(f"Layer {layer+1}:")
    for head, head_attention in enumerate(attention[0]):
        print(f"Head {head+1}:")
        for source_token, target_tokens in enumerate(head_attention[:len(tokens)]):
            print(f"Source token '{tokens[source_token]}' (index {source_token+1}):")
            for target_token, attention_value in
enumerate(target_tokens[:len(tokens)]):
                print(f"Target token '{tokens[target_token]}' (index
{target_token+1}): {attention_value}")
```

The program displays the output as shown in the following excerpt:

```
Layer 4:
Head 1:
Source token 'the' (index 1):
Target token 'the' (index 1): 0.35965585708618164
Target token 'output' (index 2): 0.016658220440149307
Target token 'shows' (index 3): 0.0013477668398991227
Target token 'the' (index 4): 0.002952627604827285
Target token 'attention' (index 5): 0.0156584158539772O3
Target token 'values' (index 6): 0.0007451129495166242
Source token 'output' (index 2):
Target token 'the' (index 1): 0.12423665076494217
Target token 'output' (index 2): 0.1629541665315628
Target token 'shows' (index 3): 0.0036389527376741 17
Target token 'the' (index 4): 0.0013890396803617477
Target token 'attention' (index 5): 0.467225402593612 67
Target token 'values' (index 6): 0.0005048955790698528
…
```

The output is interesting to examine. Take some time to look at each element of the output before moving to the next cell:

- `Layer`: the layer number 0 to n that is being analyzed
- `Head`: the layer number 0 to n in a layer
- `Source token`: the token for which the attention values are computed. The token is each word or subword.
- `Target token`: the token that the source token is applying attention to.

`Index` is the position from 0 to n in the input sequence

The values displayed are the attention values based on the architecture of an attention head, as explained in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model.* Take the time to review the chapter if necessary. The values represent the process of the output we examined in Chapter 2:

$$Attention(\boldsymbol{Q},\boldsymbol{K},\boldsymbol{V}) = softmax\left(\frac{\boldsymbol{QK^T}}{\sqrt{\boldsymbol{d_k}}}\right)V$$ However, we extracted the output of the attention head before applying `V`. So, in this case, we are looking at the following:

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

The softmax function is handy in this case because now the attention scores sum up to 1, thus adding up to 100%. We can thus see the "attention" the source token pays to each target function. The higher the score, the stronger the relationship is.In this case, we have 12 layers, each containing 12 heads. The dimensionality of the input embedding is 768. Since we have 12 heads, the scaling parameter is *dk* =768/12=64. This description can be found in the *Additional Information* section of the notebook.We excluded `V` from the interpretation function because if `V` were applied, it would make the output more difficult to analyze. Furthermore, adding additional operations blurs the initial values.Please take a moment to look at the values streamed to see how the heads produce their stochastic values.We will now create a score matrix interface to clarify the relationships between the words.

Visualizing Word Relationships Using Attention Scores with Pandas

Streaming the outputs provides valuable information on attention head outputs. However, viewing the relationships in a word x word matrix will make interpreting the model much more accessible.The program first begins by importing Pandas so that we can load the data in a DataFrame. Then it imports ipywidgets to create an interactive interface with drop-down menus and filters, among other functions:

Copy Explain

```
#Displaying the outputs in Pandas
import pandas as pd
import ipywidgets as widgets
```

The program can now create a DataFrame for each layer and each head attention matrix:

Copy Explain

```
# Create a DataFrame for each layer and head's attention matrix
df_layers_heads = []
for layer, attention in enumerate(attentions):
    for head, head_attention in enumerate(attention[0]):
        attention_matrix = head_attention[:len(tokens), :len(tokens)].detach().numpy()
 # detach the tensor from gradients and convert to numpy
        df_attention = pd.DataFrame(attention_matrix, index=tokens, columns=tokens)
        df_layers_heads.append((layer, head, df_attention))
```

The program sets the display options:

Copy Explain

```
# Set the DataFrame display options for better visualization
pd.set_option('display.max_columns', None)
pd.set_option('display.expand_frame_repr', False)
pd.set_option('max_colwidth', None)
```

We need a function to display the attention matrix:

Copy Explain

```
# Function to display the attention matrix
def display_attention(selected_layer, selected_head):
    _, _, df_to_display = next(df for df in df_layers_heads if df[0] == selected_layer
and df[1] == selected_head)
    display(df_to_display)
```

We continue by creating an interactive widget for the layers and head:

Copy Explain

```
# Create interactive widgets for the layer and head
layer_widget = widgets.IntSlider(min=0, max=len(attentions)-1, step=1,
description='Layer:')
head_widget = widgets.IntSlider(min=0, max=len(attentions[0][0])-1, step=1,
description='Head:')
```

Our interactive interface is ready to be displayed:

Copy Explain

```
# Use the widgets to interact with the function
widgets.interact(display_attention, selected_layer=layer_widget,
selected_head=head_widget)
```

The output is an interpretable interface that displays the values of relationships between the words in each head(0 to 11) as they progress through the layers(0 to 12):

Layer: 2

Head: 4

	the	output	shows	the	attention	values
the	0.707479	0.033644	0.017860	0.034750	0.034133	0.029946
output	0.540537	0.133895	0.005742	0.030299	0.086131	0.016524
shows	0.534301	0.069996	0.009407	0.070419	0.054985	0.048486
the	0.603289	0.015633	0.031514	0.108724	0.015514	0.078233
attention	0.598435	0.098799	0.003900	0.024784	0.083625	0.014138
values	0.722781	0.038620	0.008522	0.023299	0.030720	0.020654

Figure 9.xxx:

You can now select a layer and a head to analyze the outputs of the attention heads.For example, let's look at the relationship between the word "attention" and "values" at layer 0 for head 0:The relationship between the source word "values" and the target word "attention" is not the highest for "values." If you look carefully, you will see that "the" seems to attract "values" more than "attention."

Layer: 0

Head: 0

	the	output	shows	the	attention	values
the	0.070714	0.160558	0.049482	0.050669	0.166155	0.088148
output	0.148749	0.159173	0.087599	0.087984	0.200537	0.060334
shows	0.087292	0.048361	0.188823	0.130096	0.058543	0.108047
the	0.103095	0.054673	0.270750	0.110395	0.057181	0.072417
attention	0.122744	0.166162	0.108632	0.088309	0.186993	0.066312
values	0.217896	0.063374	0.178478	0.102067	0.058483	0.060048

Figure 9.xxx:

Now, let's display the relationship between the words "values" and "attention" at layer 9 for head 9.

Layer: 9

Head: 9

	the	output	shows	the	attention	values
the	0.037859	0.024309	0.007693	0.009355	0.045668	0.025113
output	0.033815	0.035063	0.031079	0.032262	0.015219	0.003380
shows	0.025111	0.235621	0.059744	0.014839	0.105298	0.003025
the	0.017671	0.054411	0.042255	0.107930	0.056619	0.016663
attention	0.031302	0.026050	0.009859	0.050517	0.103798	0.026981
values	0.024675	0.021757	0.001162	0.003473	0.041717	0.003615

Figure 9.xxx:

The relationship between the source word “values” and the target word “attention” is higher than between “values” and “the.” The model is learning!The representations might show whether the model was sufficiently trained and the architecture is efficient. In any case, the interpretation is precisely what will help you evaluate and interpret a model.

> Note: We only examined part of the model’s architecture before applying other sublayers.

Before leaving this section, we will add ExBERT, another resource, to our BERT model attention head explorations.

ExBERT

In this section, we analyzed the inner workings of the attention layer of a `'bert-base-uncased'` model at a relatively low level. You can also investigate the inner workings of the `'bert-base-uncased` model with ExBERT. It is derived from Hoover et al. (2021), *EXBERT: A Visual Analysis Tool to Explore Learned Representations in Transformers Models.*Hugging Face implemented a user-friendly interface of ExBERT. You can visualize the attention heads of `'bert-base-uncased` at a high level. This high-level interface is well-designed and intuitive, as shown in Figure xxx.xxx.

Figure 9.xxx:

You can select the layers, the heads, and other intuitive options. You can also explore other models with Hugging Face ExBERT: https://huggingface.co/spaces/simon-clmtd/exbert . The server may not always be available, but the system is worth exploring.We will now explore the exciting SHAP explainer for Hugging Face Transformers.

Interpreting Hugging Face transformers with SHAP

In this section, we will interpret the Hugging Face transformers with SHAP. The Hugging Face platform provides an interface for an impressive list of transformer models.The section is divided into two parts:

- Introducing SHAP
- Explaining Hugging Face outputs with SHAP

Introducing SHAP

In Game Theory, a Shapley value expresses the distribution of the total values among "players" through their marginal contribution. In a sentence, the words are the "players." Each word will have a score. The total score is the value of the game. The value of each word is calculated over all the permutations of the sentence.The goal is to see how each word changes the meaning of a sentence.For example, there are seven words in the following sentence: "I love playing chess with my friends"The total

number of permutations = !7= 7x6x5x4x3x2x1= 5040.The immediate conclusion is that SHAP will be challenging for a long text. However,For relatively short texts, it is efficient.The formula to calculate the Shapley value for a player (in this case, a word) *i* in a game (sentence) with N players is as follows:The marginal contribution will thus depend on the permutation computed, which leads to examining all the permutations. Let's assign values to the words in the sentence and some coalitions.The value of the words:

1. 1. I (0.2)
2. 2. love (0.6)
3. 3. playing (0.5)
4. 4. chess (0.4)
5. 5. with (0.1)
6. 6. my (0.3)
7. 7. friends (0.4)

Some values of coalitions:

Copy Explain

```
- "I love" (+0.3)
- "love playing" (+0.25)
- "playing chess" (+0.45)
- "with my friends" (+0.35)
```

The values of these words and coalitions could be the output of a machine learning algorithm, for example.Open Hugging_Face_SHAP.ipynb to dive into the Python example of SHAP.

Copy Explain

```
Let’s write a program with the Python itertools library to run the loops and
permutations for us.
```

First, we enter the value of the words and coalitions:

Copy Explain

```
#SHAP with an Iterator
import itertools
# Define the base scores of the words
words = {'I': 0.2, 'love': 0.6, 'playing': 0.5, 'chess': 0.4, 'with': 0.1, 'my': 0.3,
'friends': 0.4}
# Define the bonus scores for certain combinations of words
bonus = {('I', 'love'): 0.3, ('love', 'playing'): 0.25, ('playing', 'chess'): 0.45,
('with', 'my', 'friends'): 0.35}
```

We define a function to compute the total value of a coalition:

Copy Explain

```
# Function to calculate the total score of a coalition
def total_score(coalition):
    score = sum(words[word] for word in coalition)
    for b in bonus.keys():
        if all(word in coalition for word in b):
            score += bonus[b]
    return score
```

Now, we ask the iterator to calculate the value of a word for each coalition:

Copy Explain

```
# Function to calculate the Shapley value of a word
def shapley_value(word):
    N = len(words)
    permutations = list(itertools.permutations(words))
    marginal_contributions = []
    counter = 0  # Counter initialization
    for permutation in permutations:
        index = permutation.index(word)
        coalition_without_word = permutation[:index]
        coalition_with_word = permutation[:index+1]
        marginal_contribution = total_score(coalition_with_word) -
total_score(coalition_without_word)
        marginal_contributions.append(marginal_contribution)
        counter += 1  # Increment counter
    print(f"Processed {counter} permutations")  # Print counter
    return sum(marginal_contributions)
```

Finally, we ask the function to display the contributions of each word:

```
# Calculate the Shapley value of each word
for word in words:
    print(f"The Shapley value of '{word}' is {shapley_value(word)}")
```

The output displays the marginal contribution of each word in the sentence:

```
Processed 5040 permutations
The Shapley value of 'I' is 1764.0000000000798
Processed 5040 permutations
The Shapley value of 'love' is 4409.99999999998
Processed 5040 permutations
The Shapley value of 'playing' is 4283.999999999768
Processed 5040 permutations
The Shapley value of 'chess' is 3150.000000000016
Processed 5040 permutations
The Shapley value of 'with' is 1092.000000000074
Processed 5040 permutations
The Shapley value of 'my' is 2099.999999999982
Processed 5040 permutations
The Shapley value of 'friends' is 2604.0000000001683
```

Now, let's apply SHAP to Hugging Face Transformers.

Explaining Hugging Face outputs with SHAP

We will apply SHAP to Hugging Face transformers to analyze a sentiment analysis output.Open Hugging_Face_SHAP.ipynb in this chapter of the GitHub repository.First, we will install the packages we need:

```
#Hugging Face Transformers
!pip install transformers
#Transformer building blocks
!pip install xformers
#SHAP
!pip install shap
```

We installed the Hugging Face transformers library, `xformers`, which contains transformer building blocks, and SHAP.You can now enter the sentence you wish to explore SHAP for Hugging Face transformers:

```
#@title Enter your sentence here:
sentence = 'SHAP is a useful explainer' #@param {type:"string"}
```

A standard sentiment analysis is performed with the `distilbert-base-uncased-finetuned-sst-2-english`:

```
import transformers
# load a transformers pipeline model
model = transformers.pipeline('sentiment-analysis', model='distilbert-base-uncased-finetuned-sst-2-english')
# analyze the sentiment of the input sentence
result = model(sentence)[0]
print(result)
```

The output provides the label and the score:

```
{'label': 'POSITIVE,' 'score': 0.9869391918182373}
```

The scores will vary with the sentence and the model's training level.We can now implement the SHAP explainer, compute the values, and plot the result:

```
import shap
# explain the model on the input sentence
explainer = shap.Explainer(model)
shap_values = explainer([sentence])
# visualize the first prediction's explanation for the predicted class
predicted_class = result['label']
shap.plots.text(shap_values[0, :, predicted_class])
```

The output displays the label of the sentiment analysis class and the words that contributed to the classification:

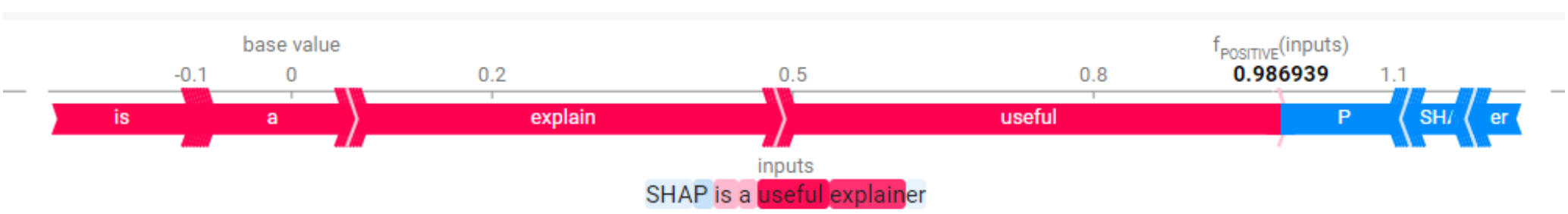

Figure 9.XXX:

The bar on the left (red) displays the words contributing to the classification. The bar on that right(blue) contains the words that pushed the classification. Regardless of the classification (positive or negative), the words on the right push the score toward the classification, and the words on the right push the score back. We can display the score of each word:

```
# get the sentiment score
sentiment_score = model(sentence)[0]['score']
# print the SHAP values for each word
words = sentence.split(' ')
for word, shap_value in zip(words, shap_values.values[0, :, 0]):
    print(f"Word: {word}, SHAP value: {shap_value}")
```

The individual score of each word is not a label (positive or negative). It is the marginal mathematical contribution of the word to the final output:

```
Word: SHAP, SHAP value: 0.0
Word: is, SHAP value: 0.12446051090955734
Word: a, SHAP value: 0.17050934582948685
Word: useful, SHAP value: -0.11854982748627663
Word: explainer, SHAP value: -0.09484541043639183
```

Take some time to enter sentences of your choice to see how the marginal contribution of the words influences the classification obtained.Let's continue our journey and visualize transformer layers through dictionary learning.

Transformer visualization via dictionary learning

Transformer visualization via dictionary learning is based on transformer factors. The goal is to analyze words in their context.

Transformer factors

A transformer factor is an embedding vector that contains contextualized words. A word without context can have many meanings, creating a polysemy issue. For example, the word `separate` can be a verb or an adjective.
Furthermore, `separate` can mean disconnect, discriminate, scatter, and many other definitions.*Yun* et al. (2021) thus created an embedding vector with contextualized words. A word embedding vector can be constructed with sparse linear representations of word factors. For example, depending on the context of the sentences in a dataset, `separate` can be represented as:

```
separate=0.3" keep apart"+"0.3" distinct"+ 0.1 "discriminate"+0.1 "sever" + 0.1
"disperse"+0.1 "scatter"
```

To ensure that a linear representation remains sparse, we don't add 0 factors that would create huge matrices with 0 values. Thus, we do not include useless information such as:

```
separate= 0.0"putting together"+".0" "identical"
```

The whole point is to keep the representation sparse by forcing the coefficients of the factors to be greater than 0.The hidden state for each word is retrieved for each layer. Since each layer progresses in understanding the representation of the word in the dataset of sentences, the latent dependencies build up. This sparse linear superposition of transformer factors becomes a dictionary matrix with a sparse vector of coefficients to be inferred that we can sum up as:

$$\varphi R^{dxm} \alpha$$

In which:

- φ (phi) is the dictionary matrix
- α is the sparse vector of coefficients to be inferred

Yun et al.(2021), added, ε, Gaussian noise samples to force the algorithm to search for deeper representations.Also, to ensure the representation remains sparse, the equation must be written s.t. (such that) $\alpha > 0$.The authors refer to *X* as the set of hidden states of the layers and *x* as a sparse linear superposition of transformer factors that belong to *X*.They beautifully sum up their sparse dictionary learning model as:

$$X = \varphi\alpha + \varepsilon \; s \; t \; \alpha > 0$$

In the dictionary matrix, φ:,c refers to a column of the dictionary matrix and contains a transformer factor.φ:,c is divided into three levels:**Low-level** transformer factors to solve polysemy problems through word-level disambiguation**Mid-level** transformer factors take us further into sentence-level patterns that will bring vital context to the low level.**High-level** transformer patterns that will help understand long-range dependenciesThe method is innovative, exciting, and seems efficient. However, there is no visualization functionality at this point. Therefore, *Yun* et al.(2021) created the necessary information for LIME, a standard interpretable AI method to visualize their findings.The interactive transformer visualization page is thus based on LIME for its outputs. The following section is a brief introduction to LIME.

Introducing LIME

LIME stands for **Local Interpretable Model-Agnostic Explanations.** The name of this explainable AI method speaks for itself. It is *model-agnostic*. Thus, we can draw immediate consequences about the method of transformer visualization via dictionary learning:

> This method does not dig into transformer layers' matrices, weights, and matrix multiplications.
>
> The method does not explain how a transformer model works, as we did in *Chapter 2*, *Getting Started with the Architecture of the Transformer Model*.
>
> In this chapter, the method peeks into the mathematical outputs provided by the sparse linear superpositions of transformer factors.

LIME does not try to parse all of the information in a dataset. Instead, LIME finds out whether a model is *locally reliable* by examining the features around a prediction.LIME does not apply to the model globally. Instead, it focuses on the local

environment of a prediction.This is particularly efficient when dealing with NLP because LIME explores the context of a word, providing invaluable information on the model's output.In visualization via dictionary learning, an instance *x* can be represented as:

$$x \in \mathrm{R}^d$$

The interpretable representation of this instance is a binary vector:

$$x' \in \{0,1\}^{d'}$$

The goal is to determine the local presence or absence of a feature or several features. In NLP, the features are tokens that can be reconstructed into words.For LIME, *g* represents a transformer model or any other machine learning model. *G* represents a set of transformer models containing *g*, among other models:

$g \in G$LIME's algorithm can thus be applied to any transformer model.At this point, we know that:

- LIME targets a word and searches the local context for other words
- LIME thus provides the local context of a word to explain why that word was predicted and not another one

For more on LIME, see the *References* section.Let's now see how LIME fits in the method of transformer visualization via dictionary learning.Let's now explore the visualization interface.

The visualization interface

Visit the following site to access the interactive transformer visualization page: https://transformervis.github.io/transformervis/.The visualization interface provides intuitive instructions to start analyzing a transformer factor of a specific layer in one click, as shown in *Figure 9.12*:

Visualization

In the following box, input a number c indicating the transformer factor $\Phi_{:,c}$ you want to visualize. Then click the button "Visualize!" to visualize this transformer factor at a particular layer. For a transformer factor $\Phi_{:,c}$ and for a layer-l, the visualization is done by listing the 200 word and context with the largest sparse coefficients $\alpha_c^{(l)}$'s

421 ← **Enter an integer from 0 to 531, indicating the transformer factor you want to visualize.**

Figure 9.12: Selecting a transformer factor

Once you have chosen a factor, you can click on the layer you wish to visualize for this factor:

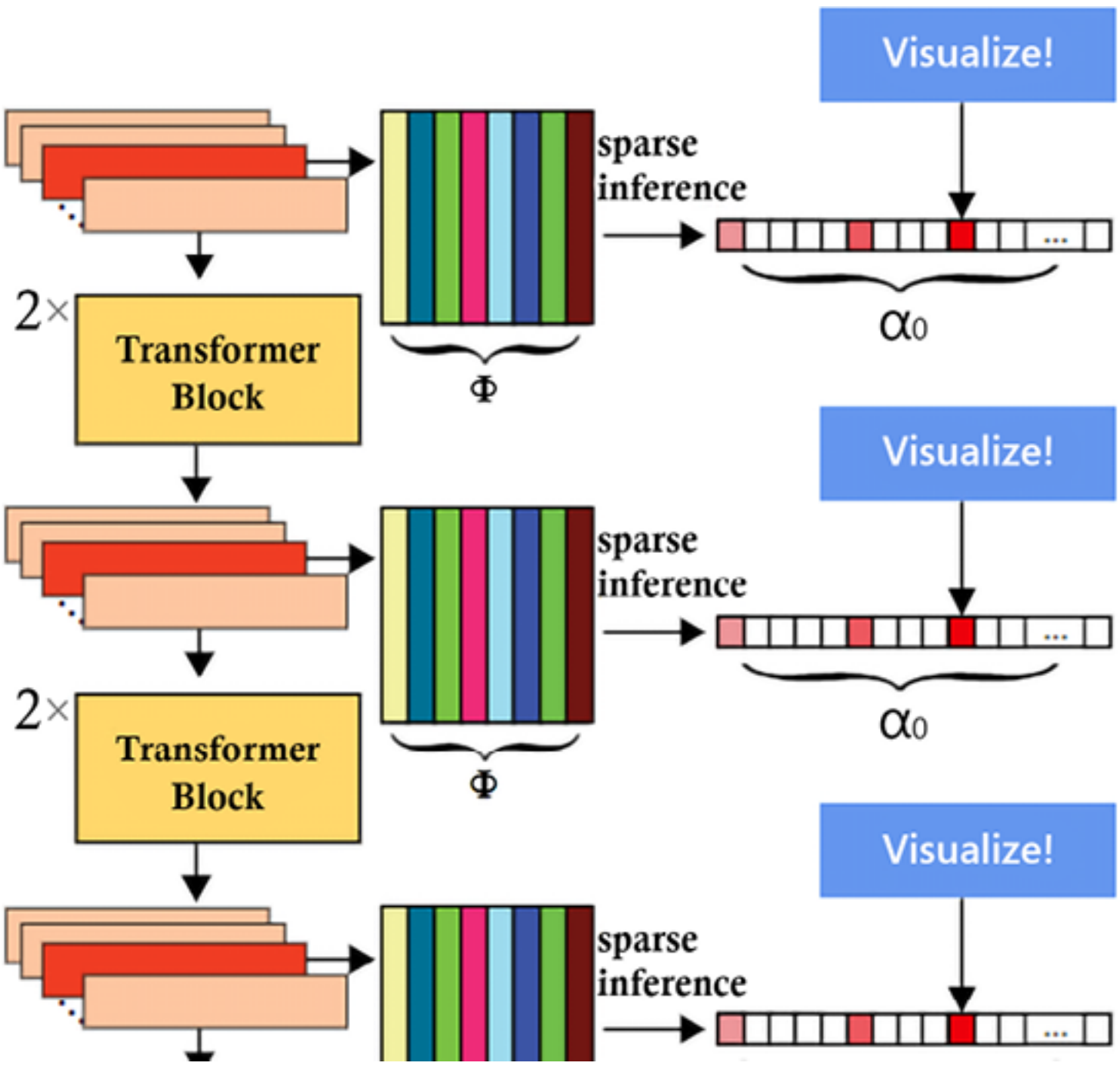

Figure 9.13: Visualize function per layer

The first visualization shows the activation of the factor layer by layer:

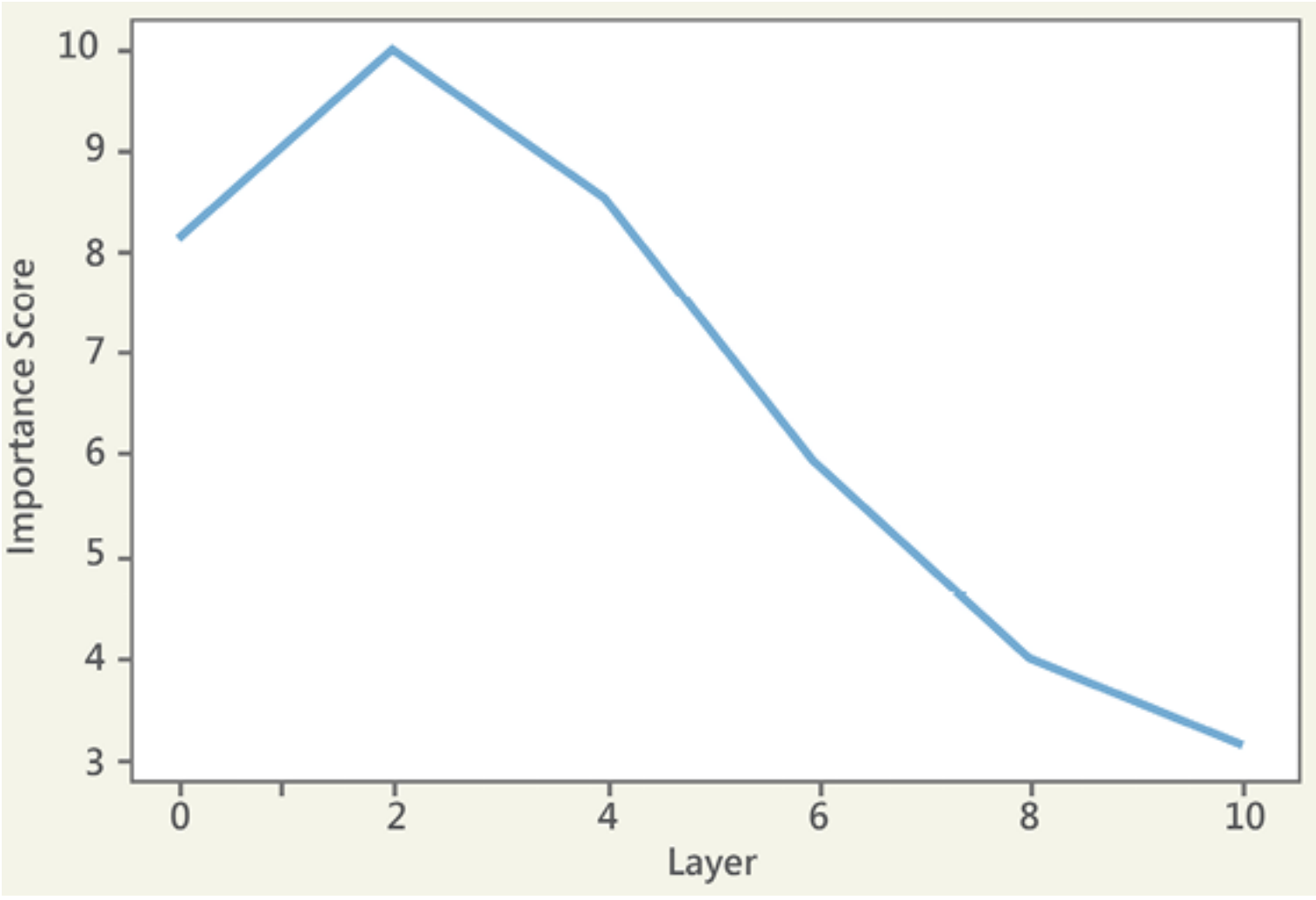

Figure 9.14: Importance of a factor for each layer

Factor `421` focuses on the lexical field of `separate`, as shown in the lower layers:

• music, and while the band initially kept these releases separate, alice in chains' self@-@
• and o. couesi were again regarded as separate as a result of further work in texas,
• in july 2014, and changed to read" a separate moh is presented to an individual for each
• without giving it proper structure or establishing it as a separate doctrine.
• those species, and is now considered to form a separate, monotypic genus – homarinus.
•rp, each npc is typically played by a separate crew member.
•," abzug" is presented as a separate track.

Figure 9.15: The representation of "separate" in the lower layers

As we visualize higher layers, longer-range representations emerge.

Factor `421` began with the representation of `separate`. But at higher levels, the transformer began to form a deeper understanding of the factor and associated `separate` with `distinct`, as shown in *Figure 9.16*:

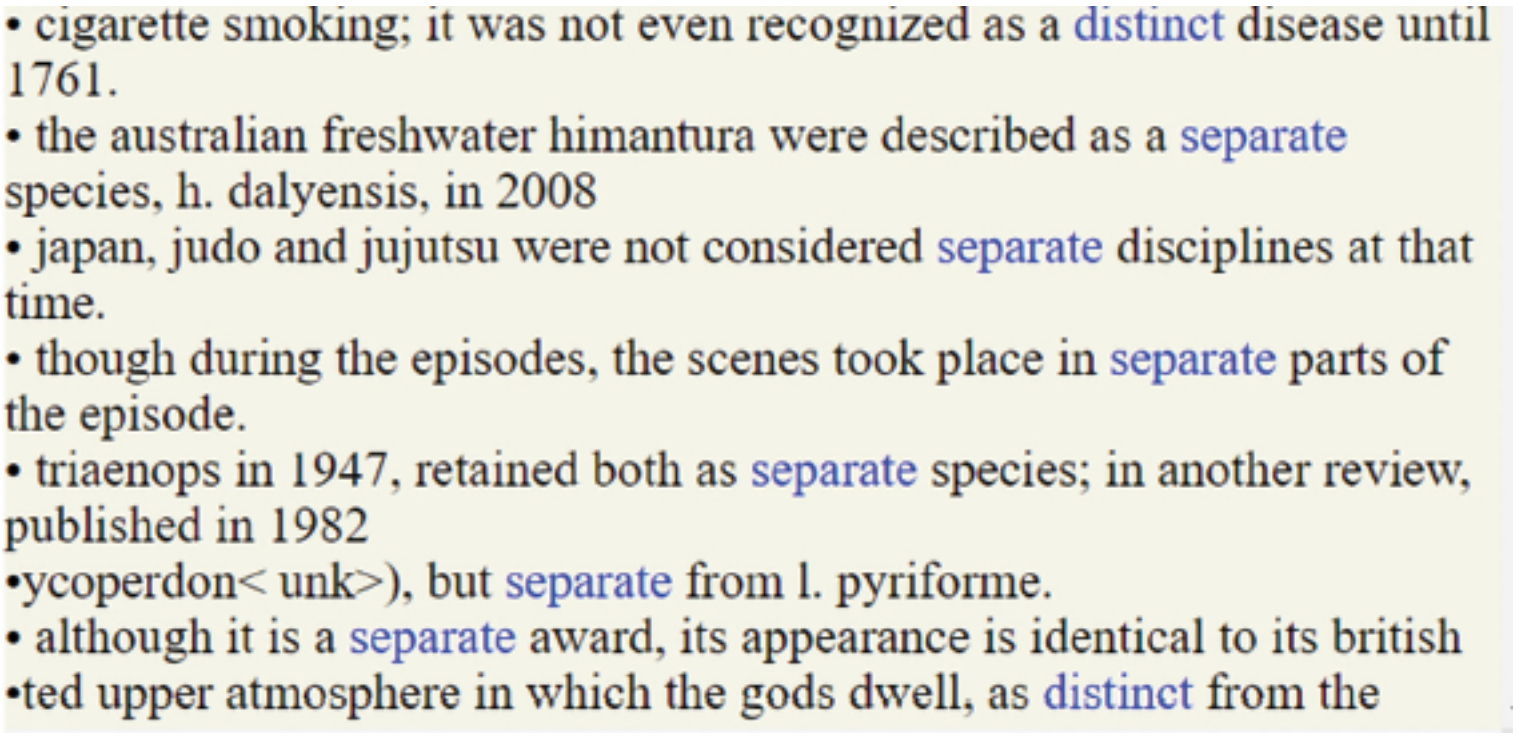

Figure 9.16: The higher-layer representations of a transformer factor

Let's conclude our exploration of the available interpretability tools by briefly going through LIT and OpenAI's nascent GPT-4 explainer.

Other Interpretable AI Tools

There are many other methods and tools to interpret transformer models. We will briefly examine two efficient tools: LIT and OpenAI's GPT-4 explainer. Let's now begin with the intuitive LIT tool.

LIT

LIT's visual interface will help you find examples that the model processes incorrectly, analyze similar examples, see how the model behaves when you change a context, and more language issues related to transformer models.LIT does not display the activities of the attention heads as `BertViz` does. However, it's worth analyzing why things went wrong and trying to find solutions.You can choose a **Uniform Manifold Approximation and Projection** (**UMAP**) visualization or a PCA projector representation. PCA will make more linear projections in specific directions and magnitude. UMAP will break its projections down into mini-clusters. Both approaches make sense depending on how far you want to go when analyzing the output of a model. You can run both and obtain different perspectives of the same model and examples.This section will use PCA to run LIT. Let's begin with a brief reminder of how PCA works.

PCA

PCA takes data and represents it at a higher level.Imagine you are in your kitchen. Your kitchen is a 3D Cartesian coordinate system. The objects in your kitchen are all at specific *x*, *y*, *z* coordinates too.You want to cook a recipe and gather the ingredients on your kitchen table. Your kitchen table is a higher-level representation of the recipe in your kitchen.The kitchen table uses a cartesian coordinate system too. But when you extract the *main features* of your kitchen to represent the recipe on your kitchen table, you are performing PCA. This is because you have displayed the principal components that fit together to make a specific recipe.The same representation can be applied to NLP. For example, a dictionary is a list of words. But the words that mean something together constitute a representation of the principal components of a sequence.The PCA representation of sequences in LIT will help visualize the outputs of a transformer.The main steps to obtain an NLP PCA representation are:**Variance:** The numerical variance of a word in a dataset; the frequency and frequency of its meaning, for example.**Covariance:** The variance of more than one word is related to that of another word in the dataset.**Eigenvalues and eigenvectors:** To obtain a representation in the cartesian system, we need the vectors and magnitudes representation of the covariances. The eigenvectors will provide the direction of the vectors. The eigenvalues will provide their magnitudes.**Deriving the data:** The last step is to apply the feature vectors to the original dataset by multiplying the row feature vector by the row data:**Data to display** = row of feature vector * row of dataPCA projections provide a clear linear visualization of the data points to analyze.Let's now first get started with LIT.

Running LIT

You can run LIT online or open it in a Google Colaboratory notebook. Click on the following link to access both options:https://pair-code.github.io/lit/The tutorial page contains several types of NLP tasks to analyze:https://pair-code.github.io/lit/tutorials/In this section, we will run LIT online and explore a sentiment analysis classifier:https://pair-code.github.io/lit/tutorials/sentiment/Click on **Explore this demo yourself**, and you will enter the intuitive LIT interface. The transformer model is small:

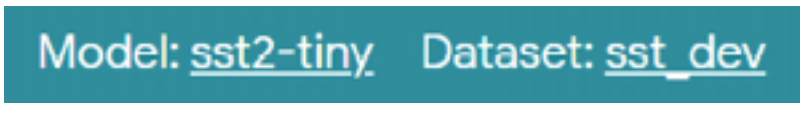

Figure 9.7: Selecting a model

You can change the model by clicking on the model. You can test this type of model and similar ones directly on Hugging Face on its hosted API page:https://huggingface.co/sshleifer/tiny-distilbert-base-uncased-finetuned-sst-2-englishThe NLP models might change on LIT's online version based on subsequent

updates. The concepts remain the same, just the models change.Let's begin by selecting the PCA projector. The projector displays a binary `0` or `1` classification label of the sentiment analysis of each example:

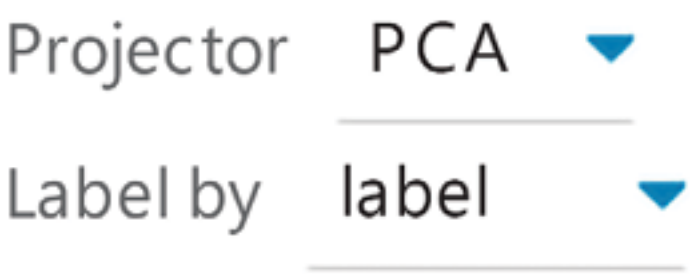

Figure 9.8: Selecting the projector and type of label

We then go to the data table and click on a **sentence** and its classification **label:**

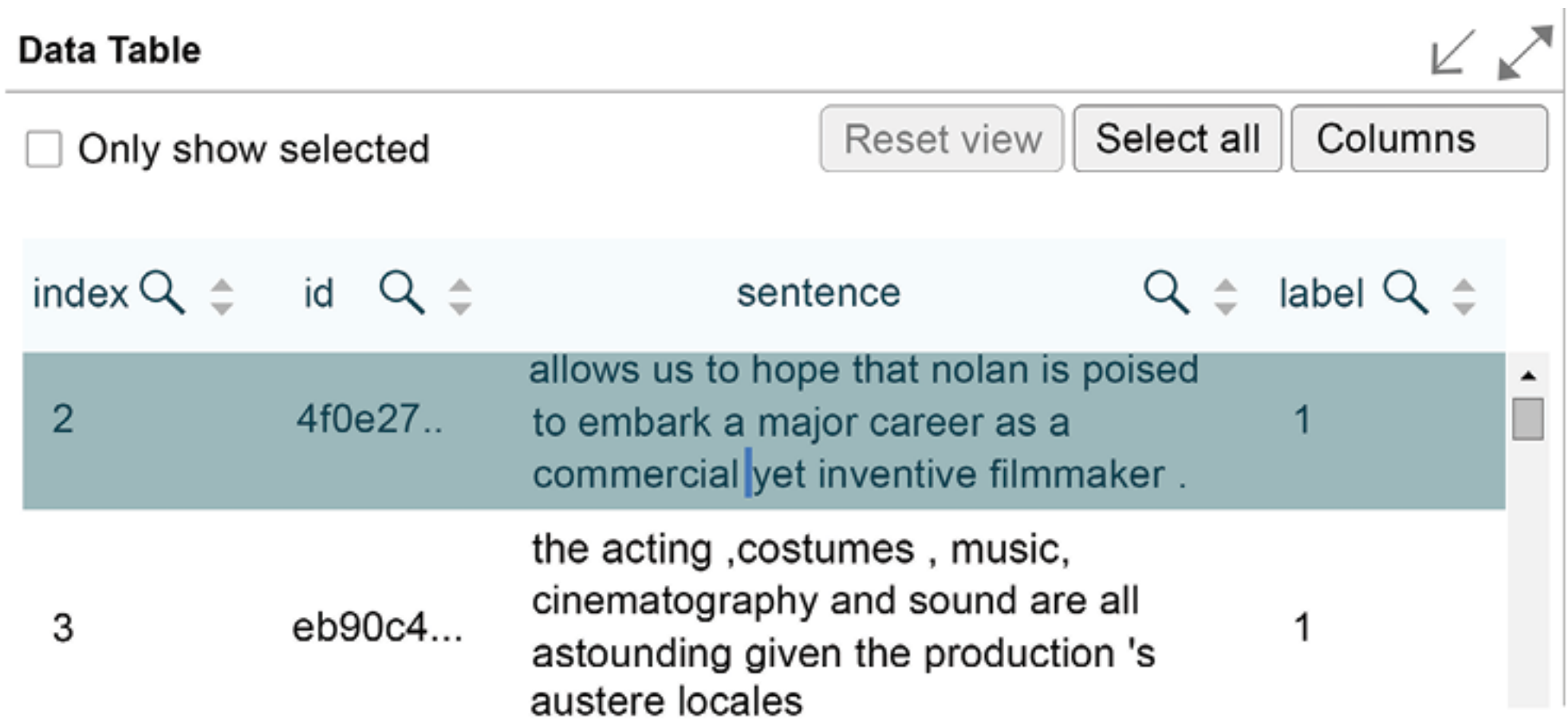

Figure 9.9: Selecting a sentence

The algorithm is stochastic so the output can vary from one run to another.The sentence will also appear in the datapoint editor:

Figure 9.10: Datapoint editor

The datapoint editor allows you to change the context of the sentence. For example, you might want to find out what went wrong with a counterfactual classification that should have been in one class but ended up in another. You can change the context of the sentence until it appears in the correct class to understand how the model works and why it made a mistake.The sentence will appear in the PCA projector with its classification:

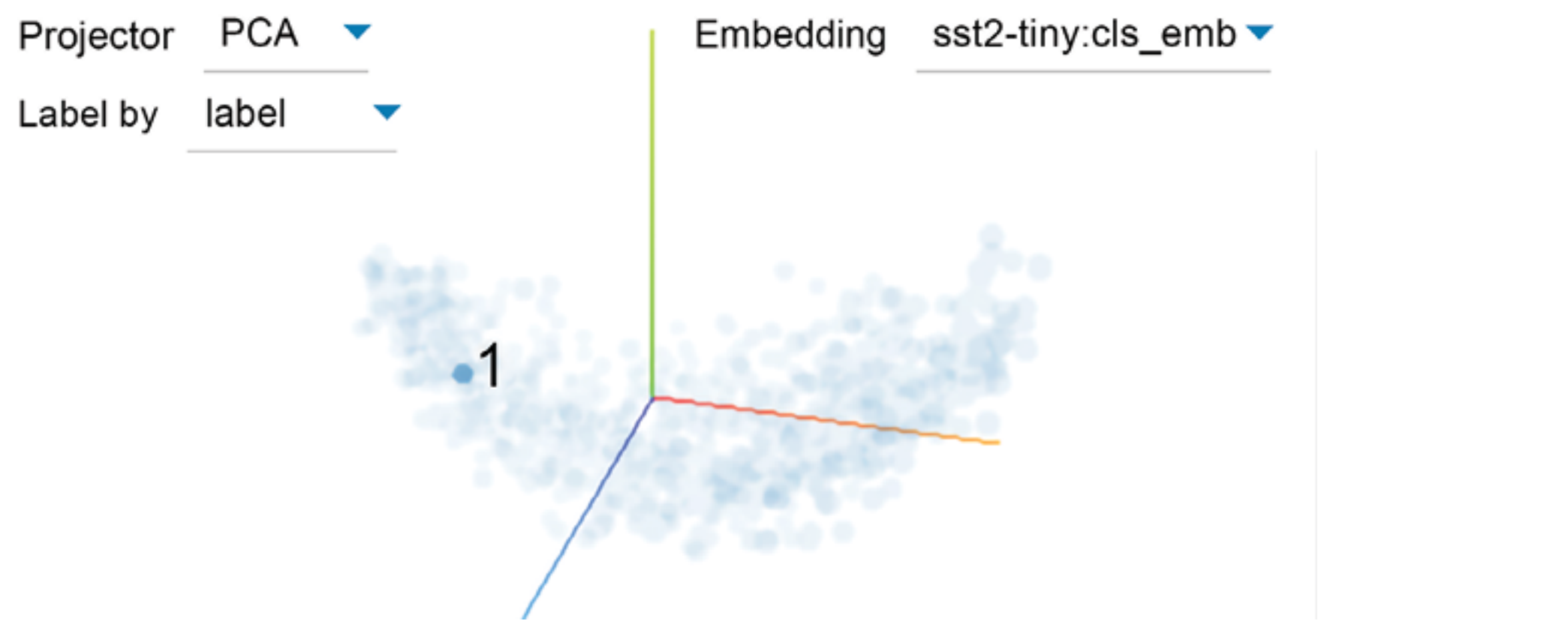

Figure 9.11: PCA projector in a positive cluster

You can click the data points in the PCA projector, and the sentences will appear in the data point editor under the sentence you selected. That way, you can compare results.LIT contains a wide range of interactive functions you can explore and use.The results obtained in LIT are not always convincing. However, LIT provides valuable insights in many cases. Also, getting involved in these emerging tools and techniques is essential.We will now discover how GPT-4 can explain the activity of neurons in an LLM model.

OpenAI LLMs Explain Neurons in Transformers

Large Language Models such as GPT-4 can explain neurons in language models. OpenAI created an intuitive interface and made it public in May 2023.The history of AI has gone to another level in a very short time. This first generation of LLMs explaining LLMs will lead to subsequent possibly exponential progress that might extend to other domains. We are in the prehistory of a new era!Click on the following link to run the explainer:https://openaipublic.blob.core.windows.net/neuron-explainer/neuron-viewer/index.htmlYou will be asked to pick a neuron for the example that is displayed:

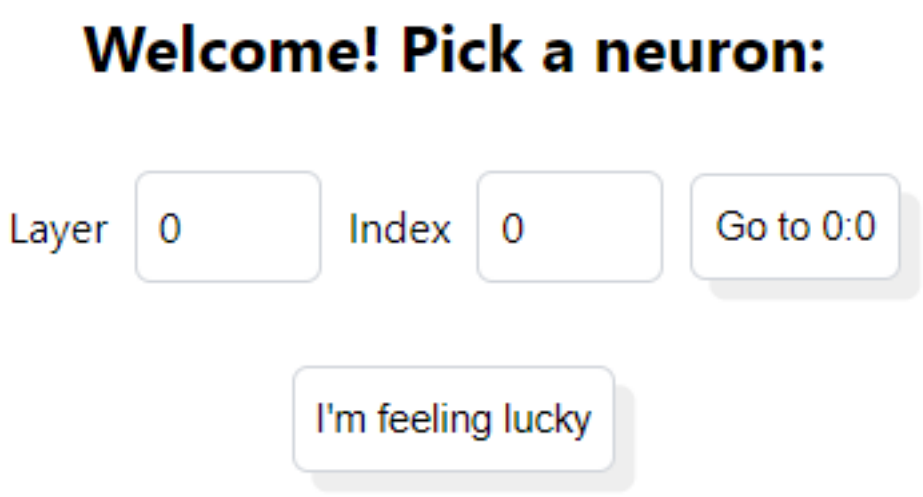

Figure 9.xxx:

Pick a neuron and click on “Go to” + the neuron. The first neuron is on layer 0, index 0. In this case, the token “ME” appears with an explanation:

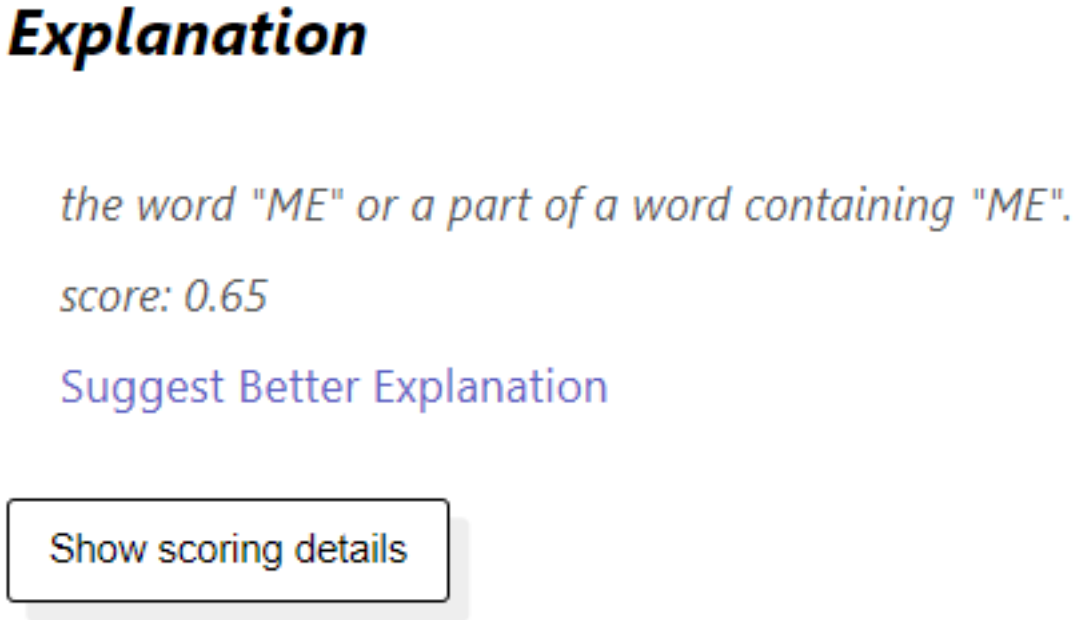

Figure 9.xxx:

We can suggest a better explanation and also ask to see the scoring details, which will explain the neuron’s scoring details by highlighting them using GPT-4, as shown in the following excerpt of the real activations:

Top

Real activations:

.txt FORMAT: the file is tab delimited with ID, MEAN-SENTIMENT-RATING, STANDARD DEVIATION, and RAW-SENTIMENT-RATINGS DESCRIPTION: Sentiment ratings from a minimum of 20 independent human raters (all pre

Simulated activations:

.txt FORMAT: the file is tab delimited with ID, MEAN-SENTIMENT-RATING, STANDARD DEVIATION, and RAW-SENTIMENT-RATINGS DESCRIPTION: Sentiment ratings from a minimum of 20 independent human raters (all pre

Real activations:

AFTER WE WERE MARRIED FOR 6 MONTHS - CHEATS ONE ME SIX MONTHS LATER, WANTS DIVORCE, AND IS GOING TO TRY FOR CHILD SUPPORT. JUST TO HURT ME AND TRY TO RUIN MY LIFE. add your own

We can gain insights by GPT-4 by clicking on `Activations:`

Activations

Top show more

Figure 9.xxx:

Quantile range [0.99, 0.999] sample show more

" (7 2/3 cm) tall. My blanket's finished size was 22" x 34" (56cm x 86cm).

The blanket can be modified to be as big or as small as you want. I did the border and center portion in seed stitch, but garter or stockinette

Figure 9.xxx:

When GPT-4 generations are applied, the system chooses the top quantile range of the top activations. At the bottom of the page, you can find the related tokens:

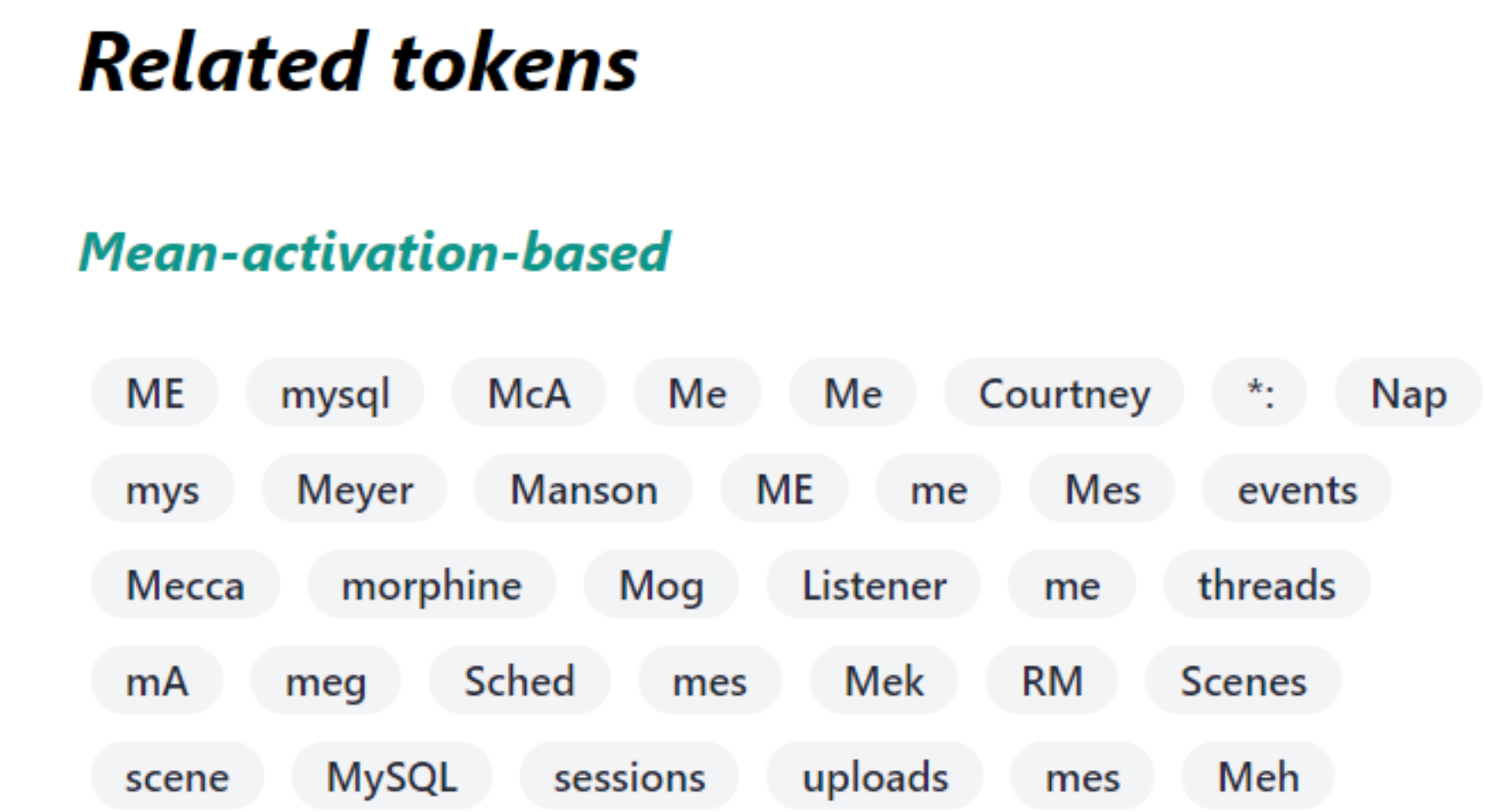

Figure 9.xxx:

Finally, we can see the related neurons:

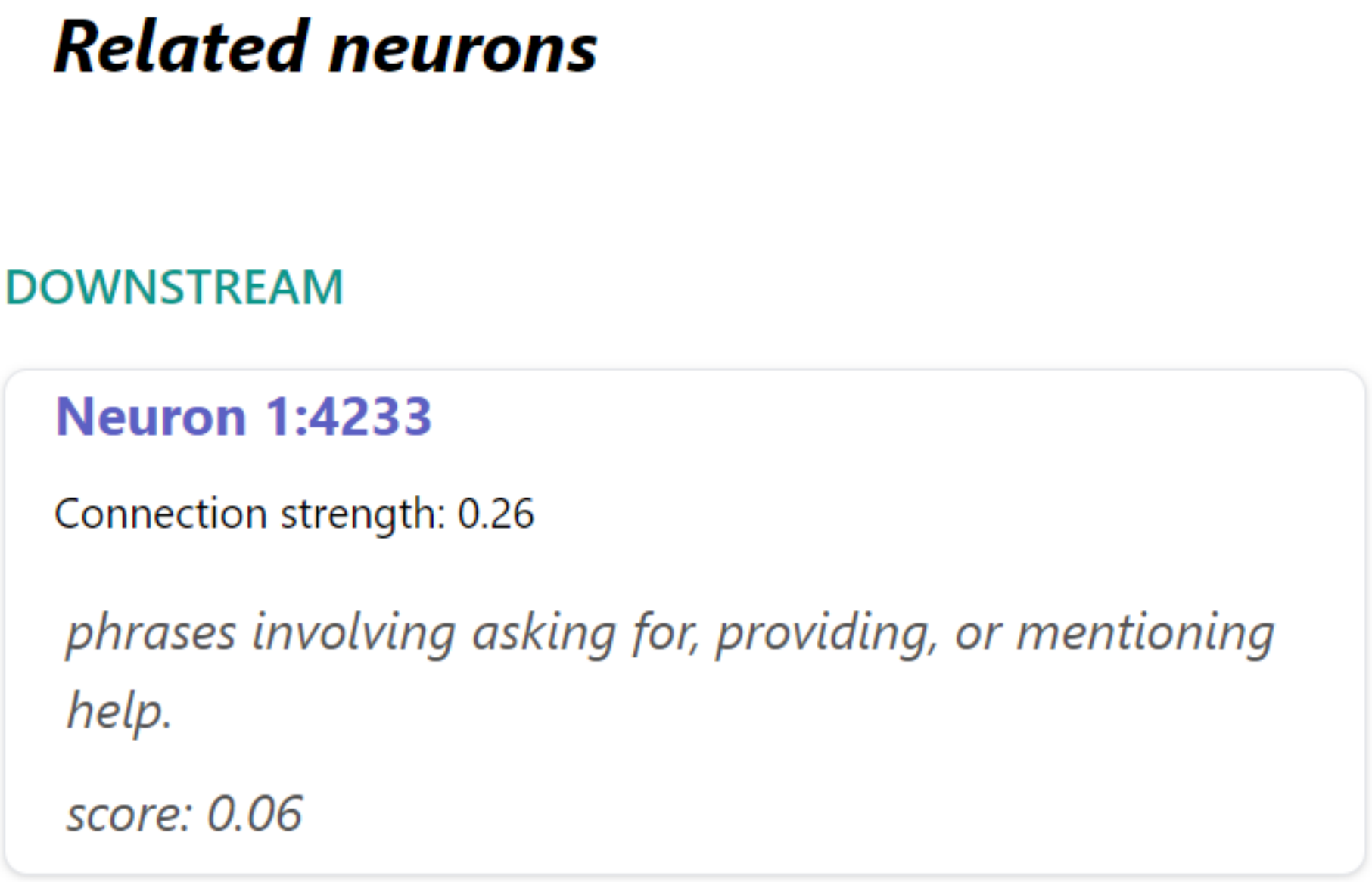

Figure 9.xxx:

The interface is in its early stages. Nevertheless, explaining an LLM with GPT-4 is quite a step forward.Exploring the neurons in this interface will help visualize a transformer's activity at the neuron level.**Takeaway:** *The explanations are obtained by comparing the actual and simulated activations with GPT-4. This research has once again shattered the transformer models' black box.* A long journey is ahead to

improve the interface's clarity and the explanations' quality. But the first step has been made!The evolution of this approach might solve a fair share of the issue of transformer interpretation.We will show why human control remains necessary.

Limitations and Human Control

Interpreting transformer models has progressed, but much remains to be done. Some of the stumbling blocks remain quite challenging:

- The embedding sublayer is based on stochastic calculations added to the complex positional encoding
- The stochastic nature of the mechanisms of the multiple attention heads makes it difficult to pinpoint why and how the score of a token prevailed after having gone through several layers.
- Softmax functions applied to raw outputs blur the tracks.
- Dropout sublayers erase some of the tracks.
- The deep learning regularizations such as ReLU and GELU hinder reverse engineering an output.

The numbers are numbing. A vintage model such as GPT-3 has 175 billion parameters and 9,216 attention heads (96 layers x 96 heads)And these are only some of the difficulties!Human evaluation remains a key resource to evaluate and find ways to help improve transparency. The increasing pressure to explain AI is pushing interpretation tools forward.You will find many good examples and poor results at the project level. Focus on the good examples to understand how a transformer makes its way through language learning. Use the poor results to understand why a transformer made a mistake. In any case, get involved and stay in the loop of this ever-evolving field!The visual interfaces explored in this chapter are fascinating. However, there is still much work to do.Human evaluation, control, intervention, and development will remain necessary for quite a long time!

Summary

Transformer models are trained to resolve word-level polysemy disambiguation and low-level, mid-level, and high-level dependencies. The process is achieved by training million - to trillion-parameter models. The task of interpreting these giant models

seems daunting. However, several tools are emerging.We first installed `BertViz`. We learned how to interpret the computations of the attention heads with an interactive interface. We saw how words interacted with other words for each layer. We introduced ExBERT, another approach to visualizing BERT, among other models.The chapter continued by defining SHAP and revealing the contribution of each word processed by Hugging Face Transformers.We then ran transformer visualization via dictionary learning with LIME. A user can choose a transformer factor to analyze and visualize the evolution of its representation from the lower layers to the higher layers of the transformer. The factor will progressively go from polysemy disambiguation to sentence context analysis and finally to long-term dependencies.Other tools, such as LIT, plug a PCA projector and UMAP representations into the outputs of a BERT transformer model. We can then analyze clusters of outputs to see how they fit together. OpenAI's innovative approach showed how Large Language Models can explain the neurons in transformers.The tools of this chapter will evolve along with other techniques. However, the key takeaway of this chapter is that transformer model activity can be visualized and interpreted in a user-friendly manner. In the next chapter, *Investigating the Role of Tokenizers in Shaping Transformer Models*, we will deepen our understanding of the inner workings of transformers. We will see why tokenizers are the cornerstone of transformer models.

Questions

1. BertViz only shows the output of the last layer of the BERT model. (True/False)
2. BertViz shows the attention heads of each layer of a BERT model. (True/False)
3. BertViz shows how the tokens relate to each other. (True/False)
4. LIT shows the inner workings of attention heads like BertViz. (True/False)
5. Probing is a way for an algorithm to predict language representations. (True/False)
6. NER is a probing task. (True/False)
7. PCA and UMAP are non-probing tasks. (True/False)
8. LIME is model agnostic. (True/False)
9. Transformers deepen the relationships of the tokens layer by layer. (True/False)
10. OpenAI Large Language Models (LLMs) can explain LLMs. (True/False)

References

BertViz: https://github.com/jessevig/BertViz*Transformer visualization via dictionary learning*: *Zeyu Yun*, *Yubei Chen*, *Bruno A Olshausen*, *Yann LeCun*, 2021, Transformer visualization via dictionary learning: contextualized embedding as a linear superposition of transformer factors, https://arxiv.org/abs/2103.15949Hugging Face with Slunberg SHAP, https://github.com/slundberg/SHAPTransformer Visualization via dictionary learning: https://transformervis.github.io/transformervis/OpenAI, Large Language Models can explain neurons in language models: https://openai.com/research/language-models-can-explain-neurons-in-language-modelsOpenAI neuro explainer paper: https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.htmlLIT: https://pair-code.github.io/lit/

Further Reading

Hoover et al. (2021), *EXBERT: A Visual Analysis Tool to Explore Learned Representations in Transformers Models*, https://arxiv.org/abs/1910.05276BertViz: *Jesse Vig,2019, A Multiscale Visualization of Attention in the Transformer Model,2019,* https://aclanthology.org/P19-3007.pdf

Join our book's Discord space

Jin the book's Discord workspace:https://www.packt.link/Transformers

Previous Chapter

Next Chapter