Chapter 11: Attention Visualization and Experiment Tracking

In this chapter, we will cover two different technical concepts, **attention visualization** and **experiment tracking**, and we will practice them through sophisticated tools such as **exBERT** and **BertViz.** These tools provide important functions for interpretability and explainability. First, we will discuss how to visualize the inner parts of attention by utilizing the tools. It is important to interpret the learned representations and to understand the information encoded by self-attention heads in the Transformer. We will see that certain heads correspond to a certain aspect of syntax or semantics. Secondly, we will learn how to track experiments by logging and then monitoring by using **TensorBoard** and **Weights & Biases** (**W&B**). These tools enable us to efficiently host and track experimental results such as loss or other metrics, which helps us to optimize model training. You will learn how to use exBERT and BertViz to see the inner parts of their own models and will be able to utilize both TensorBoard and W&B to monitor and optimize their models by the end of the chapter.

We will cover the following topics in this chapter:

- Interpreting attention heads
- Tracking model metrics

Technical requirements

The code for this chapter is found at https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH11, which is the GitHub repository for this book. We will be using Jupyter Notebook to run our coding exercises that require Python 3.6.0 or above, and the following packages will need to be installed:

```
tensorflow
pytorch
Transformers >=4.00
tensorboard
wandb
bertviz
ipywidgets
```

Check out the following link to see Code in Action Video:

https://bit.ly/3iM4Y1F

Interpreting attention heads

As with most **Deep Learning** (**DL**) architectures, both the success of the Transformer models and how they learn have been not fully understood, but we know that the Transformers—remarkably—learn many linguistic features of the language. A significant amount of learned linguistic knowledge is distributed both in the hidden state and in the self-attention heads of the pre-trained model. There have been substantial recent studies published and many tools developed to understand and to better explain the phenomena.

Thanks to some **Natural Language Processing** (**NLP**) community tools, we are able to interpret the information learned by the self-attention heads in a Transformer model. The heads can be interpreted naturally, thanks to the weights between tokens. We will soon see that in further experiments in this section, certain heads correspond to a certain aspect of syntax or semantics. We can also observe surface-level patterns and many other linguistic features.

In this section, we will conduct some experiments using community tools to observe these patterns and features in the attention heads. Recent studies have already revealed many of the features of self-attention. Let's highlight some of them before we get into the experiments. For example, most of the heads attend to delimiter tokens such as **Separator** (**SEP**) and **Classification** (**CLS**), since these tokens are never masked out and bear segment-level information in particular. Another observation is

that most heads pay little attention to the current token, but some heads specialize in only attending the next or previous tokens, especially in earlier layers. Here is a list of other patterns found in recent studies that we can easily observe in our experiments:

Attention heads in the same layer show similar behavior.

Particular heads correspond to specific aspects of syntax or semantic relations.

Some heads encode so that the direct objects tend to attend to their verbs, such as *<lesson, take>* or *<car, drive>*.

In some heads, the noun modifiers attend to their noun (for example, *the hot water*; *the next layer*), or the possessive pronoun attends to the head (for example, *her car*).

Some heads encode so that passive auxiliary verbs attend to a related verb, such as *Been damaged, was taken*.

In some heads, coreferent mentions attend to themselves, such as *talks-negotiation, she-her, President-Biden*.

The lower layers usually have information about word positions.

Syntactic features are observed earlier in the transformer, while high-level semantic information appears in the upper layers.

The final layers are the most task-specific and are therefore very effective for downstream tasks.

To observe these patterns, we can use two important tools, **exBERT** and **BertViz**, here. These tools have almost the same functionality. We will start with exBERT.

Visualizing attention heads with exBERT

exBERT is a visualization tool to see the inner parts of Transformers. We will use it to visualize the attention heads of the *BERT-base-cased* model, which is the default model in the exBERT interface. Unless otherwise stated, the model we will use in the following examples is *BERT-base-cased*. This contains 12 layers and 12 self-attention heads in each layer, which makes for 144 self-attention heads.

We will learn how to utilize exBERT step by step, as follows:

1. Let's click on the exBERT link hosted by *Hugging Face*: https://huggingface.co/exbert.
2. Enter the sentence **The cat is very sad.** and see the output, as follows:

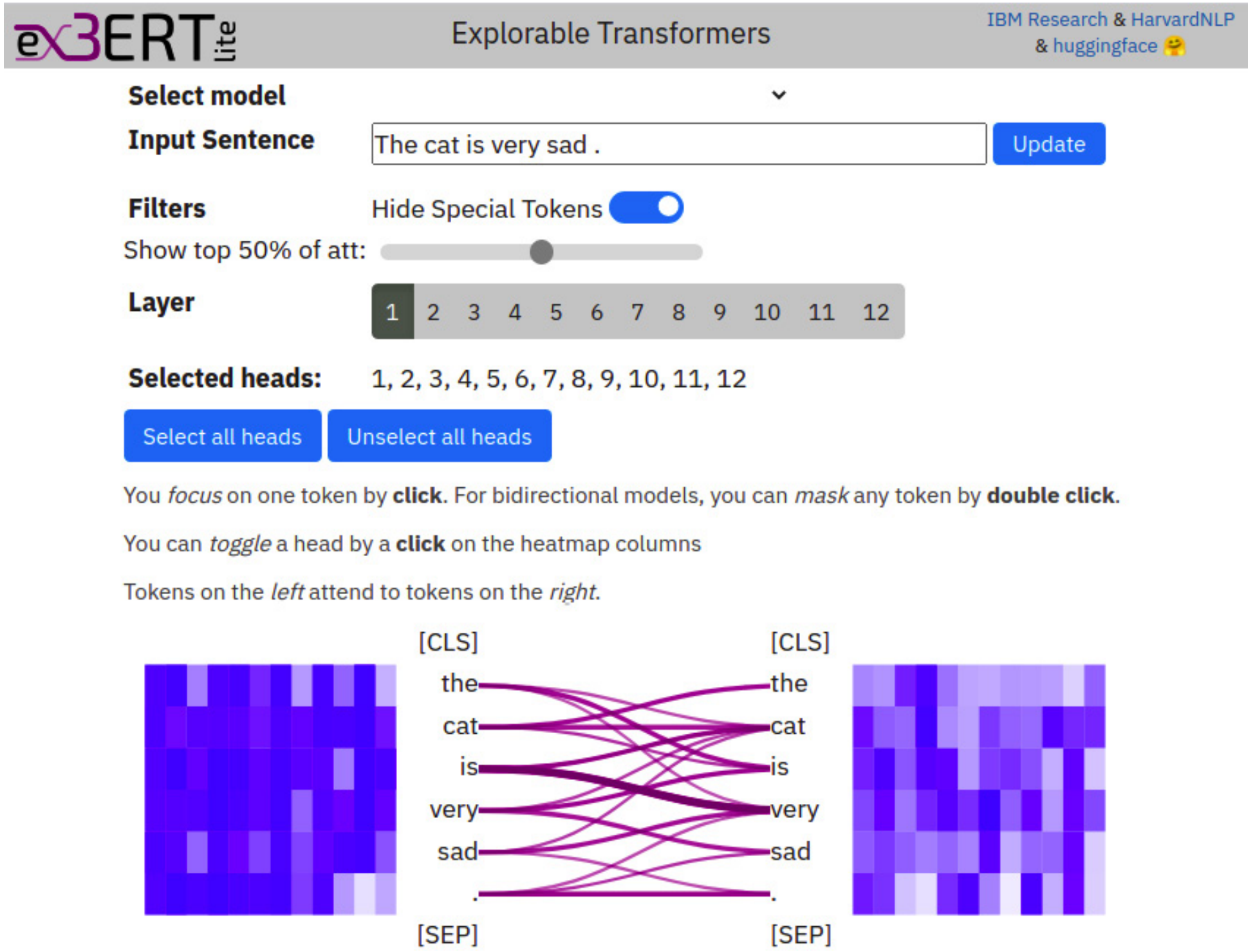


Figure 11.1 – exBERT interface

In the preceding screenshot, the left tokens attend to the right tokens. The thickness of the lines represents the value of the weights. Because CLS and SEP tokens have very frequent and dense connections, we cut off the links associated with them for simplicity. Please see the **Hide Special Tokens** toggle switch. What we see now is the attention mapping at layer 1, where the lines correspond to the sum of the weights on all heads. This is called a **multi-head attention mechanism**, in which 12 heads work in parallel to each other. This mechanism allows us to capture a wider range of relationships than is possible with single-head attention. This is why we see a broadly attending pattern in *Figure 11.1*. We can also observe any specific head by clicking the **Head** column.

If you hover over a token at the left, you will see the specific weights of that token connecting to the right ones. For more detailed information on using the interface, read the paper *exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformer Models*, *Benjamin Hoover*, *Hendrik Strobelt*, *Sebastian Gehrmann*, *2019* or watch the video at the following link: https://exbert.net/.

3. Now, we will try to support the findings of other researchers addressed in the introductory part of this section. Let's take the *some heads specialize in only*

attending the next or previous tokens, especially in earlier layers pattern, and see if there's a head that supports this.

4. We will use **<Layer-No, Head-No>** notation to denote a certain self-attention head for the rest of the chapter, where the indices start at 1 for exBERT and start at 0 for BertViz—for example, *<3,7>* denotes the seventh head at the third layer for exBERT. When you select the **<2,5> (or <4,12> or <6,2 >)** head, you will get the following output, where each token attends to the previous token only:

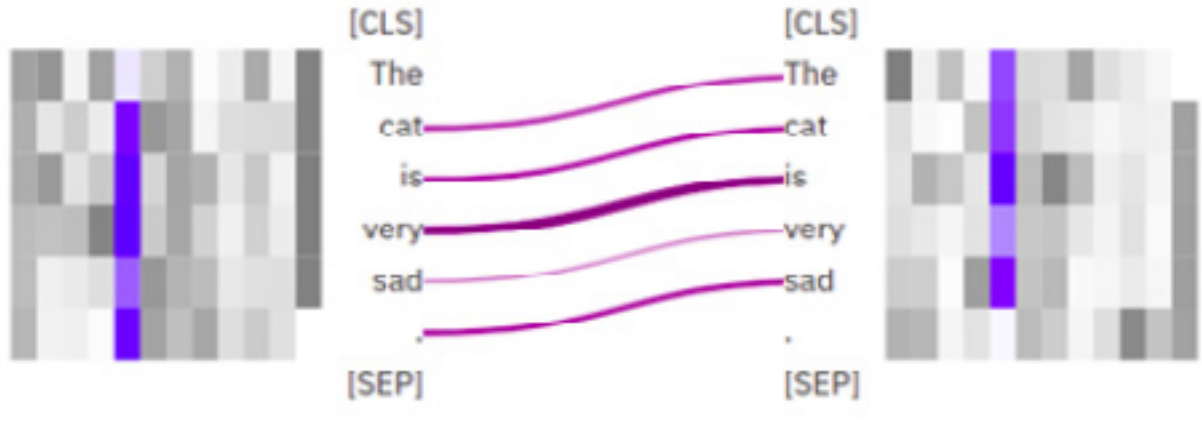


Figure 11.2 – Previous-token attention pattern

5. For the **<2, 12>** and **<3, 4>** heads, you will get a pattern whereby each token attends to the next token, as follows:

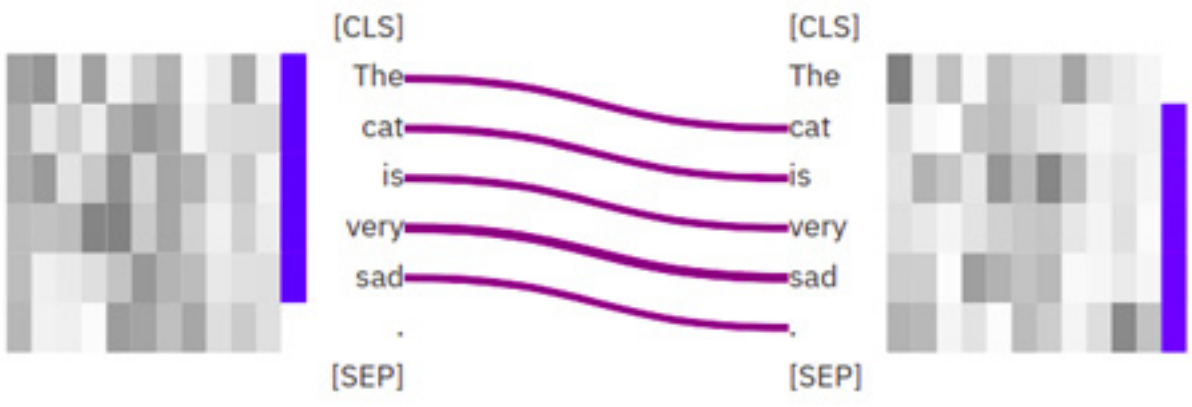


Figure 11.3 – Next-token attention pattern

These heads serve the same functionality for other input sentences—that is, they work independently of the input. You can try different sentences yourself.

We can use an attention head for advanced semantic tasks such as pronoun resolution using a **probing classifier.** First, we will qualitatively check if the internal representation has such a capacity for pronoun resolution (or coreference resolution) or not. Pronoun resolution is considered a challenging semantic relation task since the distance between the pronoun and its antecedent is usually very long.

6. Now, we take the sentence *The cat is very sad. Because it could not find food to eat.* When you check each head, you will notice that the *<9,9>* and *<9,12>* heads encode the pronoun relation. When hovering over **it** at the *<9,9>* head, we get the following output:

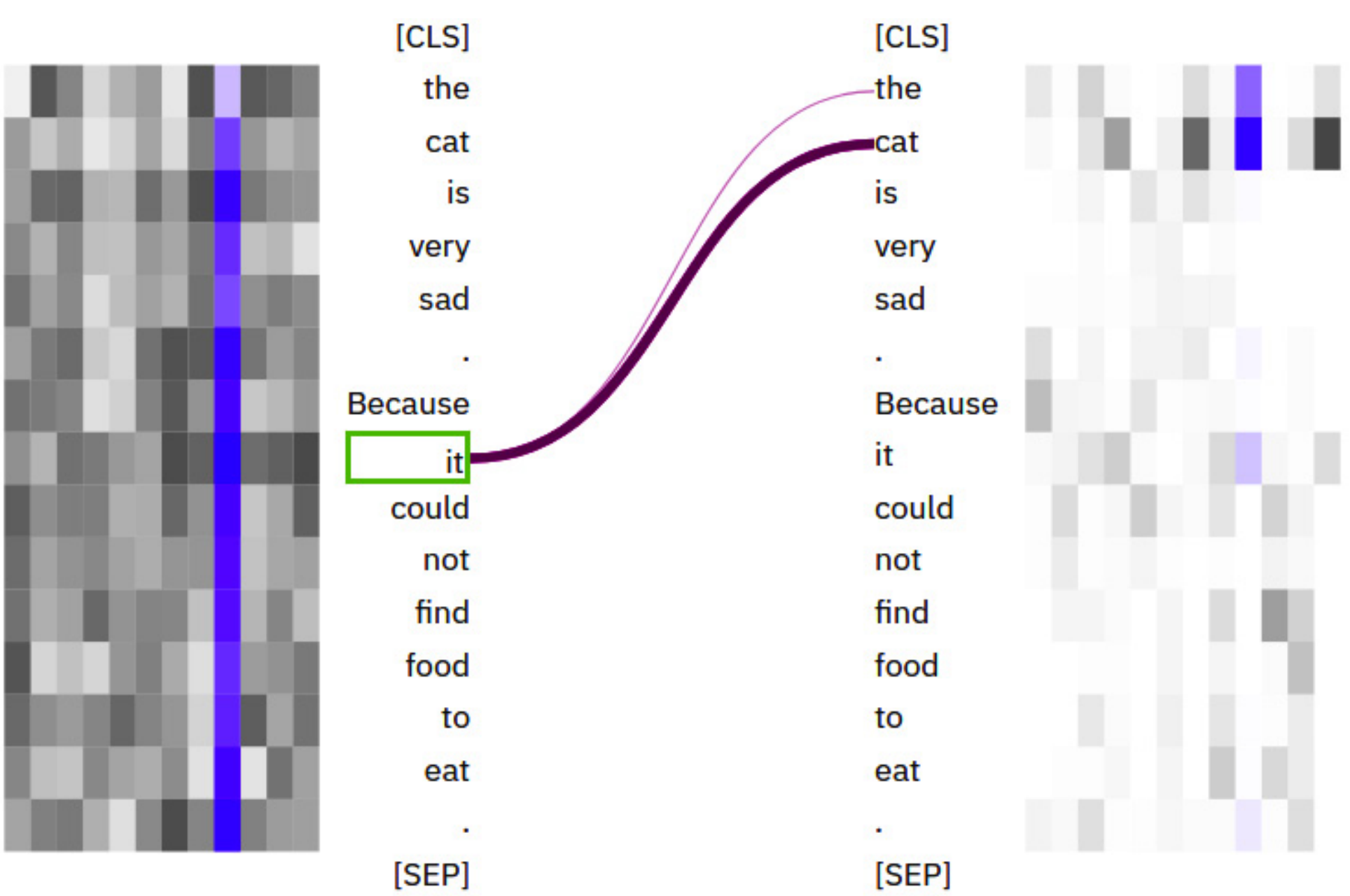


Figure 11.4 – The coreference pattern at the <9,9> head

The *<9,12>* head also works for pronoun relation. Again, on hovering over **it**, we get the following output:

Figure 11.5 – The coreference pattern at the <9,12> head

From the preceding screenshot, we see that the **it** pronoun strongly attends to its antecedent, **cat.** We change the sentence a bit so that the **it** pronoun now refers to the **food** token instead of **cat**, as in **the cat did not eat the food because it was not fresh.** As seen in the following screenshot, which relates to the *<9,9>* head, **it** properly attends to its antecedent **food**, as expected:

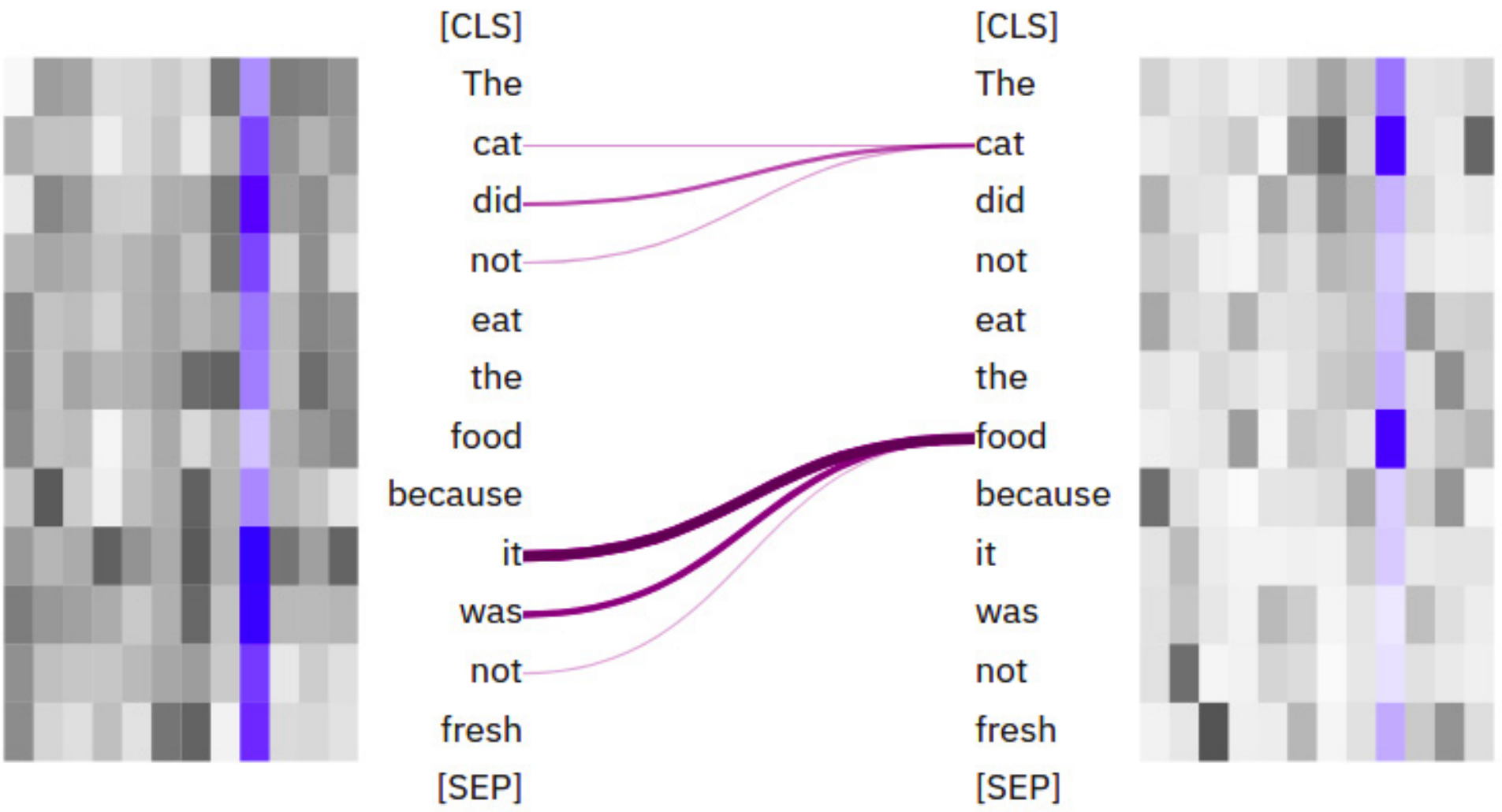


Figure 11.6 – The pattern at the <9,9> head for the second example

7. Let's take another run, where the pronoun refers to **cat**, as in **The cat did not eat the food because it was very angry.** In the *<9,9>* head, the **it** token mostly attends to the **cat** token, as shown in the following screenshot:

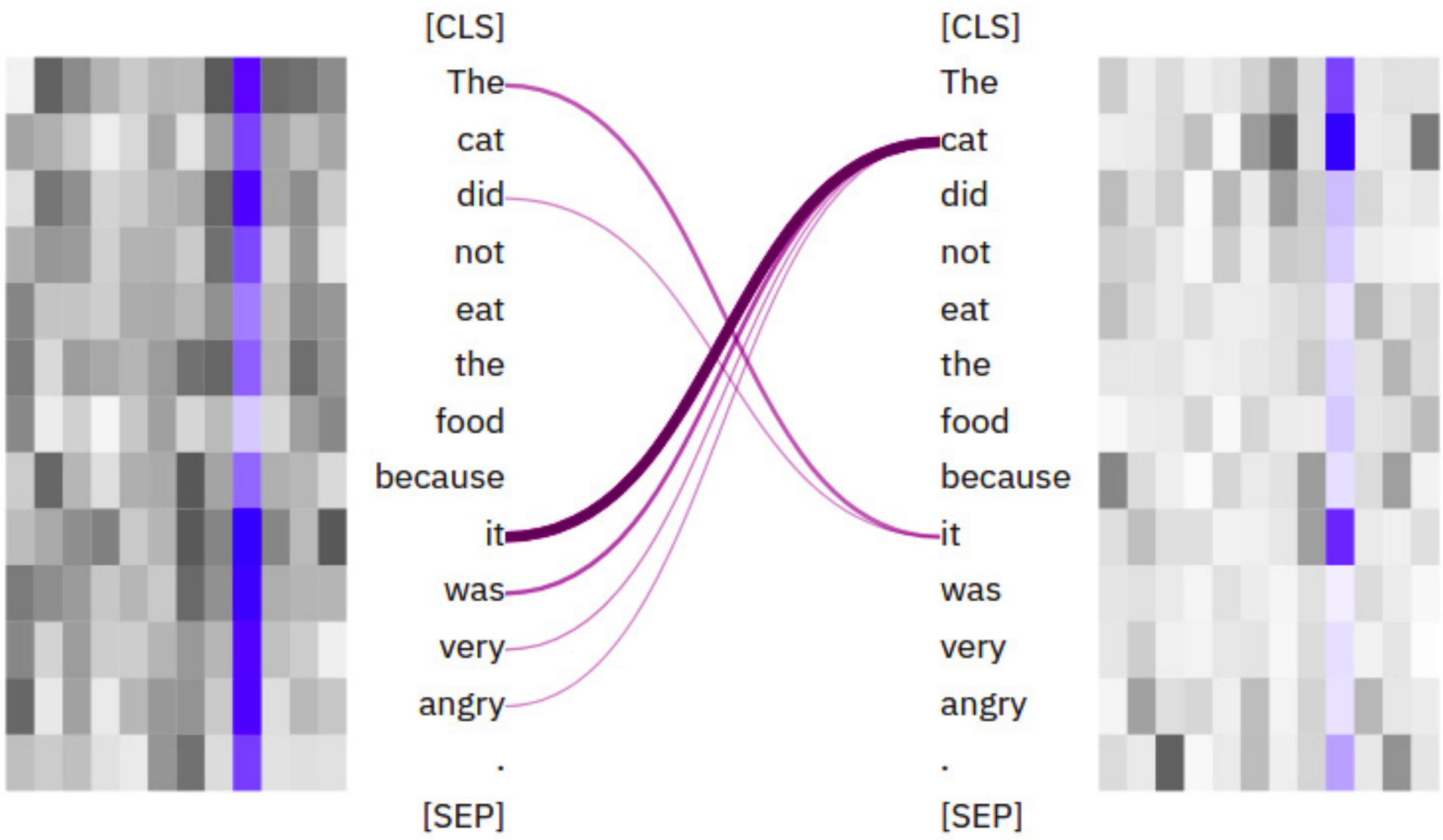


Figure 11.7 – The pattern at the <9,9> head for the second input

8. I think those are enough examples. Now, we will use the exBERT model differently to evaluate the model capacity. Let's restart the exBERT interface, select the last layer (*layer 12*), and keep all heads. Then, enter the sentence **the cat did not eat the food.** and mask out the **food** token. Double-clicking masks the **food** token out, as follows:

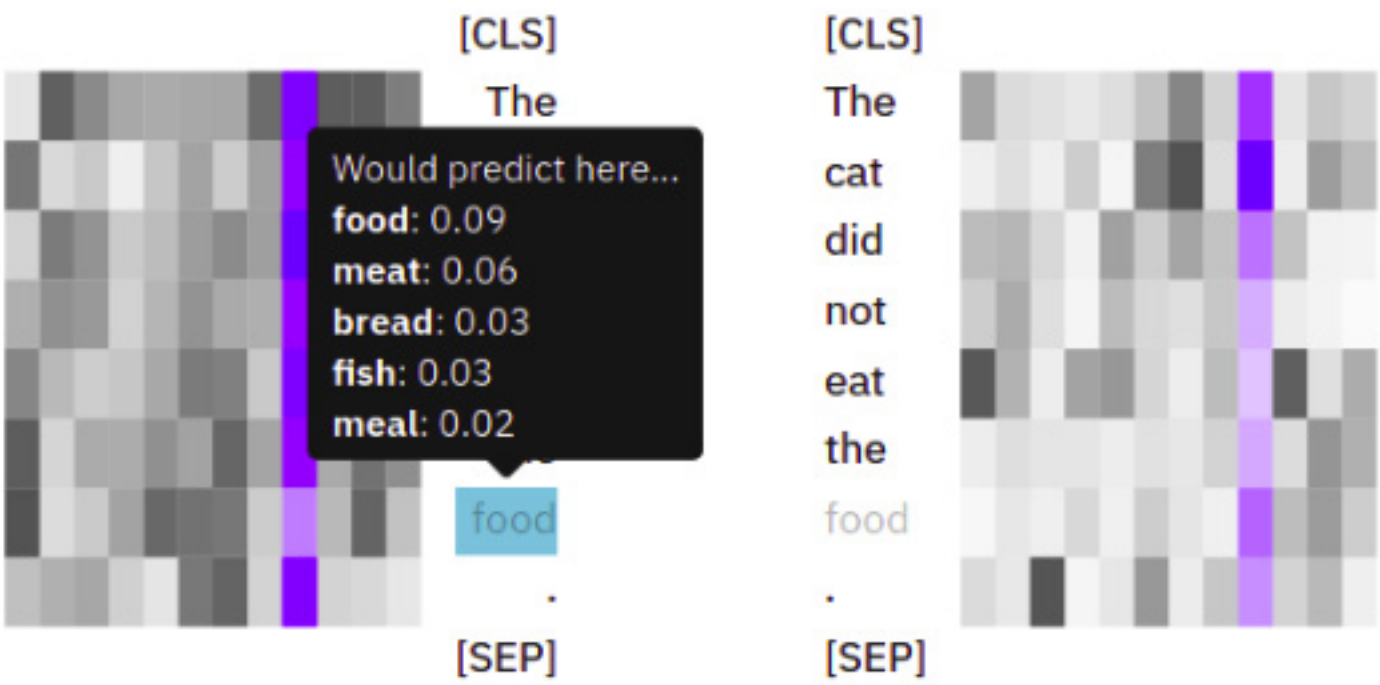


Figure 11.8 – Evaluating the model by masking

When you hover on that masked token, you can see the prediction distribution of the *Bert-base-cased* model, as shown in the preceding screenshot. The first prediction is **food**, which is expected. For more detailed information about the tool, you can use exBERT's web page, at https://exbert.net/.

Well done! In the next section, we will work with BertViz and write some Python code to access the attention heads.

Multiscale visualization of attention heads with BertViz

Now, we will write some code to visualize heads with BertViz, which is a tool to visualize attention in the Transformer model, as is exBERT. It was developed by Jesse Vig in 2019 (*A Multiscale Visualization of Attention in the Transformer Model*, *Jesse Vig*, *2019*). It is the extension of the work of the Tensor2Tensor visualization tool (Jones, 2017). We can monitor the inner parts of a model with multiscale qualitative analysis. The advantage of BertViz is that we can work with most Hugging Face-hosted models (such as **Bidirectional Encoder Representations from Transformers (BERT)**, **Generated Pre-trained Transformer (GPT)**, and **Cross-lingual Language Model (XLM)**) through the Python **Application Programming Interface (API)**. Therefore, we will be able to work with non-English models as well, or any pre-trained model. We will examine such examples together shortly. You can access BertViz resources and other information from the following GitHub link: https://github.com/jessevig/bertviz.

As with exBERT, BertViz visualizes attention heads in a single interface. Additionally, it supports a **bird's eye view** and a low-level **neuron view**, where we observe how individual neurons interact to build attention weights. A useful demonstration video can be found at the following link: https://vimeo.com/340841955.

Before starting, we need to install the necessary libraries, as follows:

```
!pip install bertviz ipywidgets
```

We then import the following modules:

```
from bertviz import head_view
from Transformers import BertTokenizer, BertModel
```

BertViz supports three views: a **head view**, a **model view**, and a **neuron view**. Let's examine these views one by one. First of all, though, it is important to point out that we started from 1 to index layers and heads in exBERT. But in BertViz, we start from 0 for indexing, as in Python programming. If I say a *<9,9>* head in exBERT, its BertViz counterpart is *<8,8>*.

Let's start with the head view.

Attention head view

The head view is the BertViz equivalent of what we have experienced so far with exBERT in the previous section. The **attention head view** visualizes the attention patterns based on one or more attention heads in a selected layer:

1. First, we define a `get_bert_attentions()` function to retrieve attentions and tokens for a given model and a given pair of sentences. The function definition is shown in the following code block:

```
def get_bert_attentions(model_path, sentence_a, sentence_b):
    model = BertModel.from_pretrained(model_path,
        output_attentions=True)
    tokenizer = BertTokenizer.from_pretrained(model_path)
    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]
    input_id_list = input_ids[0].tolist()
    tokens = tokenizer.convert_ids_to_tokens(input_id_list)
    return attention, tokens
```

2. In the following code snippet, we load the `bert-base-cased` model and retrieve the tokens and corresponding attentions of the given two sentences. We then call the `head_view()` function at the end to visualize the attentions. Here is the code execution:

```
model_path = 'bert-base-cased'
sentence_a = "The cat is very sad."
sentence_b = "Because it could not find food to eat."
attention, tokens=get_bert_attentions(model_path,
    sentence_a, sentence_b)
head_view(attention, tokens)
```

The code output is an interface, as displayed here:

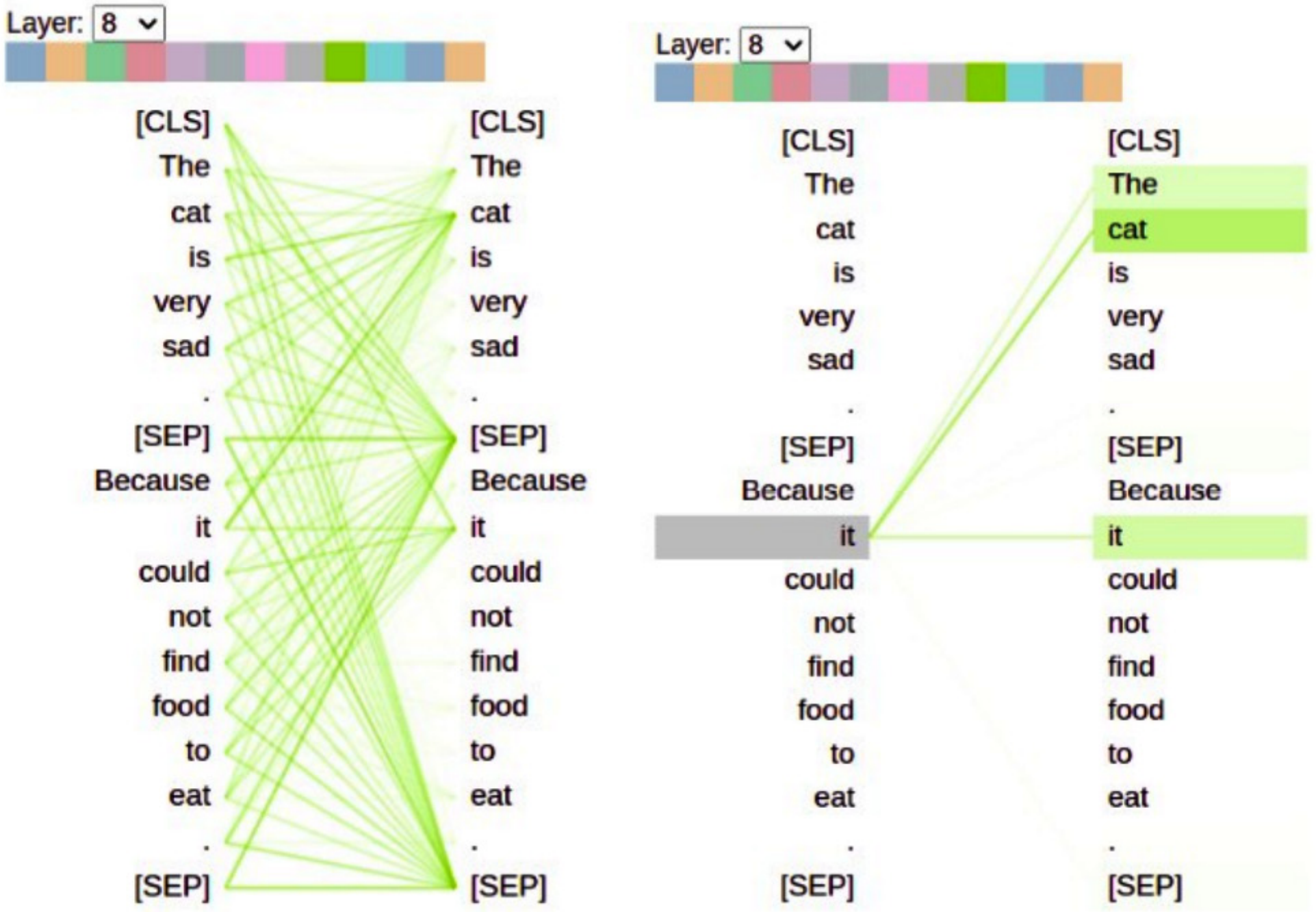


Figure 11.9 – Head-view output of BertViz

The interface on the left of *Figure 11.9* comes first. Hovering over any token on the left will show the attention going from that token. The colored tiles at the top correspond to the attention head. Double-clicking on any of them will select it and discard the rest. The thicker attention lines denote higher attention weights.

Please remember that in the preceding exBERT examples, we observed that the *<9,9>* head (the equivalent in BertViz is *<8, 8>*, due to indexing) bears a pronoun-antecedent relationship. We observe the same pattern in *Figure 11.9*, selecting layer 8 and head 8. Then, we see the interface on the right of *Figure 11.9* when we hover on *it*, where *it* strongly attends to the *cat* and *it* tokens. So, can we observe these semantic patterns in other pre-trained language models? Although the heads are not exactly the same in other models, some heads can encode these semantic properties. We also know from recent work that semantic features are mostly encoded in the higher layers.

3. Let's look for a coreference pattern in a Turkish language model. The following code loads a Turkish `bert-base` model and takes a sentence pair. We observe here that the *<8,8>* head has the same semantic feature in Turkish as in the English model, as follows:

```
model_path = 'dbmdz/bert-base-turkish-cased'
sentence_a = "Kedi çok üzgün."
sentence_b = "Çünkü o her zamanki gibi çok fazla yemek yedi."
attention, tokens=\
get_bert_attentions(model_path, sentence_a, sentence_b)
head_view(attention, tokens)
```

From the preceding code, `sentence_a` and `sentence_b` mean *The cat is sad* and *Because it ate too much food as usual*, respectively. When hovering over **o (it)**, **it** attends to **Kedi (cat)**, as follows:

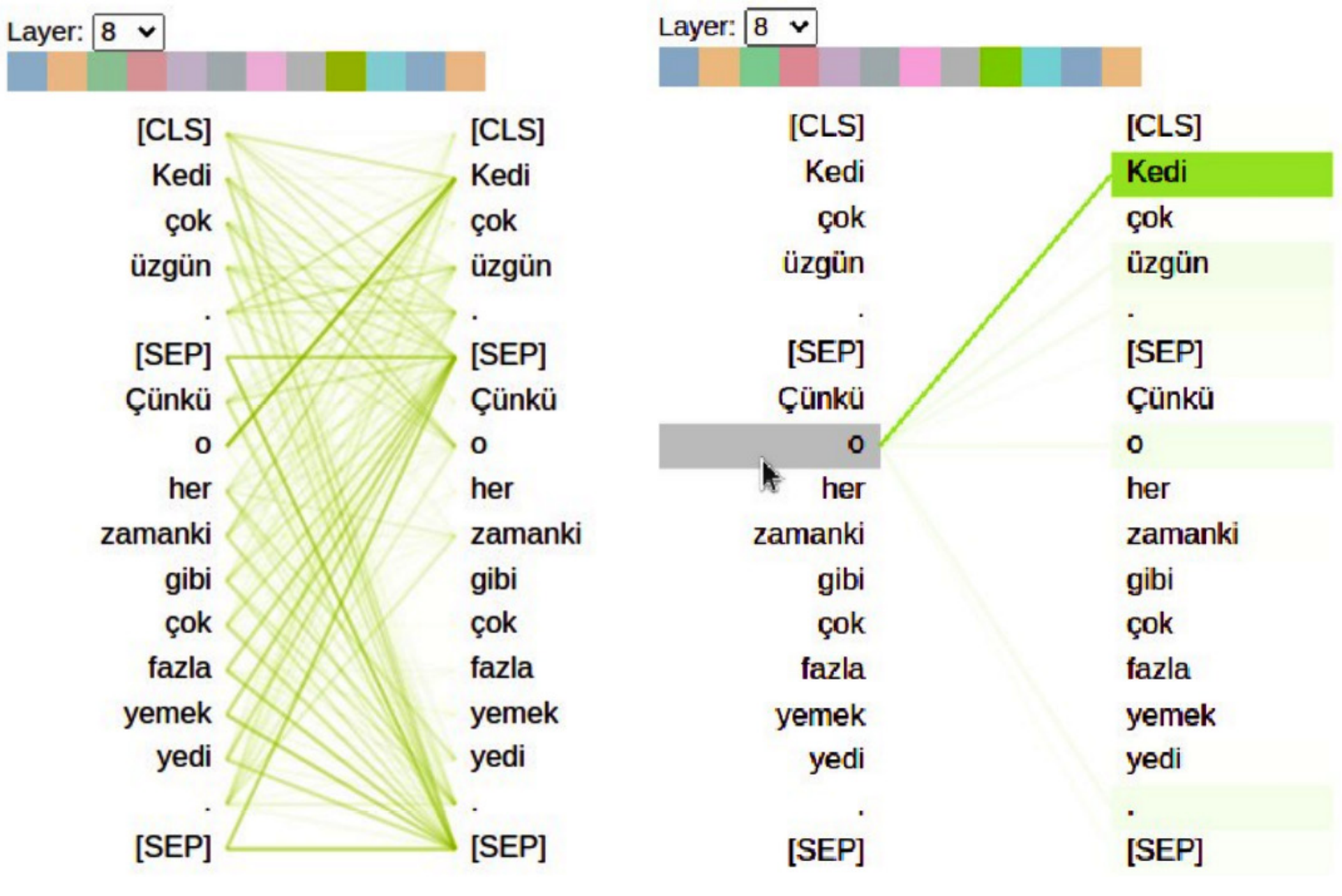


Figure 11.10 – Coreference pattern in the Turkish language model

All other tokens except **o** mostly attend to the SEP delimiter token, which is a dominant behavior pattern in all heads in the BERT architecture.

4. As a final example for the head view, we will interpret another language model and move on to the model view feature. This time, we choose the `bert-base-german-cased` German language model and visualize it for the input—that is, the German equivalent of the same-sentence pair we used for Turkish.
5. The following code loads a German model, consumes a pair of sentences, and visualizes them:

```
model_path = 'bert-base-german-cased'
sentence_a = "Die Katze ist sehr traurig."
sentence_b = "Weil sie zu viel gegessen hat"
attention, tokens=\
get_bert_attentions(model_path, sentence_a, sentence_b)
head_view(attention, tokens)
```

6. When we examine the heads, we can see the coreference pattern in the 8th layer again, but this time in the 11th head. To select the *<8,11>* head, pick layer 8 from the drop-down menu and double-click on the last head, as follows:

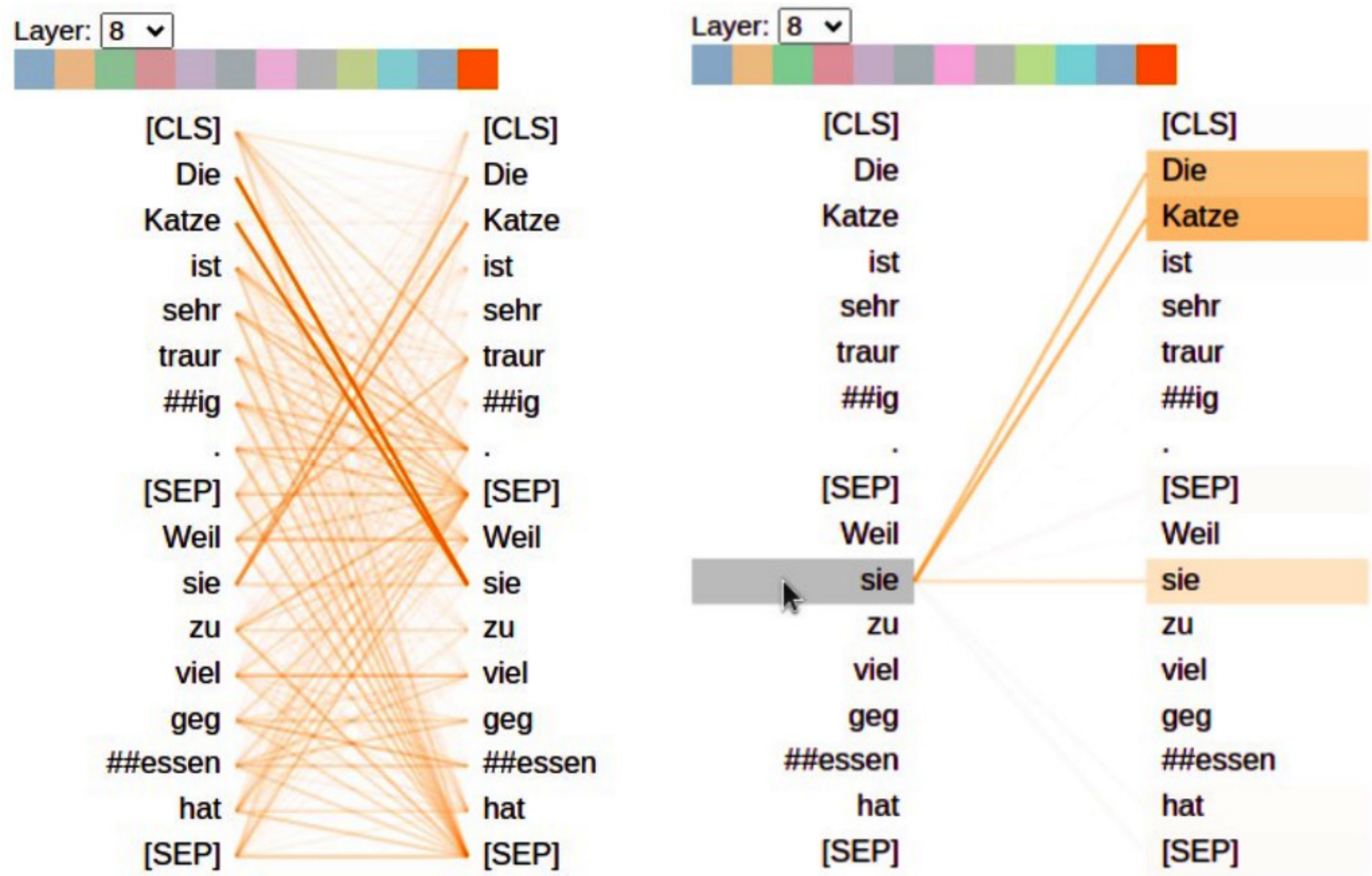


Figure 11.11 – Coreference relation pattern in the German language model

As you see, when hovering over **sie**, you will see strong attentions to the **Die Katze.** While this *<8,11>* head is the strongest one for coreference relations (known as anaphoric relations in computational linguistics literature), this relationship may have spread to many other heads. To observe it, we will have to check all the heads one by one.

On the other hand, BertViz's model view feature gives us a basic bird's-eye view to see all heads at once. Let's take a look at it in the next section.

Model view

Model view allows us to have a bird's-eye view of attentions across all heads and layers. Self-attention heads are shown in tabular form, with rows and columns corresponding to layers and heads, respectively. Each head is visualized in the form of a clickable thumbnail that includes the broad shape of the attention model.

The view can tell us how BERT works and makes it easier to interpret. Many recent studies, such as *A Primer in BERTology: What We Know About How BERT Works*, *Anna Rogers*, *Olga Kovaleva*, *Anna Rumshisky*, *2021*, found some clues about the behavior of the layers and came to a consensus. We already listed some of them in the *Interpreting attention heads* section. You can test these facts yourself using BertViz's model view.

Let's view the German language model that we just used, as follows:

1. First, import the following modules:

```
from bertviz import model_view
from Transformers import BertTokenizer, BertModel
```

2. Now, we will use a `show_model_view()` wrapper function developed by Jesse Vig. You can find the original code at the following link: https://github.com/jessevig/bertviz/blob/master/notebooks/model_view_bert.ipynb.
3. You can also find the function definition in our book's GitHub link, at https://github.com/PacktPublishing/Mastering-Transformers/tree/main/CH11. We are just dropping the function header here:

```
def show_model_view(model, tokenizer, sentence_a,
     sentence_b=None, hide_delimiter_attn=False,
     display_mode="dark"):
. . .
```

4. Let's load the German model again. If you have already loaded it, you can skip the first five lines. Here is the code you'll need:

```
model_path='bert-base-german-cased'
sentence_a = "Die Katze ist sehr traurig."
sentence_b = "Weil sie zu viel gegessen hat"
model = BertModel.from_pretrained(model_path, output_attentions=True)
tokenizer = BertTokenizer.from_pretrained(model_path)
show_model_view(model, tokenizer, sentence_a, sentence_b,
    hide_delimiter_attn=False,
    display_mode="light")
```

Copy Explain

This is the output:

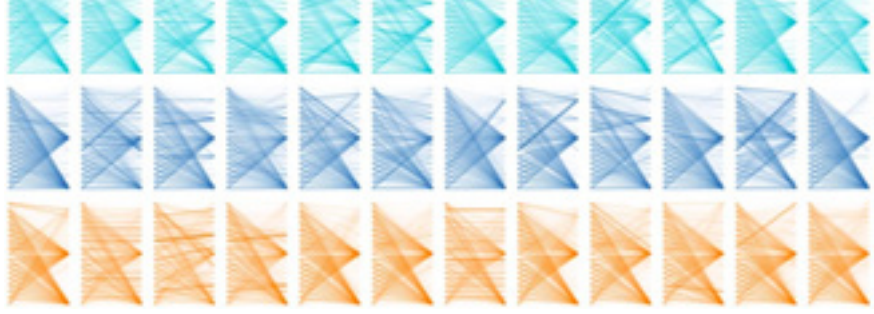


Figure 11.12 – The model view of the German language model

This view helps us easily observe many patterns such as next-token (or previous-token) attention patterns. As we mentioned earlier in the *Interpreting attention heads* section, tokens often tend to attend to delimiters—specifically, CLS delimiters at lower layers and SEP delimiters at upper layers. Because these tokens are not masked out, they can ease the flow of information. In the last layers, we only observe SEP-delimiter-focused attention patterns. It could be speculated that SEP is used to collect segment-level information, which can be used then for inter-sentence tasks such as **Next Sentence Prediction** (**NSP**) or for encoding sentence-level meaning.

On the other hand, we observe that coreference relation patterns are mostly encoded in the *<8,1>, <8,11>, <10,1>, and <10,7>* heads. Again, it can be clearly said that the *<8, 11>* head is the strongest head that encodes the coreference relation in the German model, which we already discussed.

5. When you click on that thumbnail, you will see the same output, as follows:

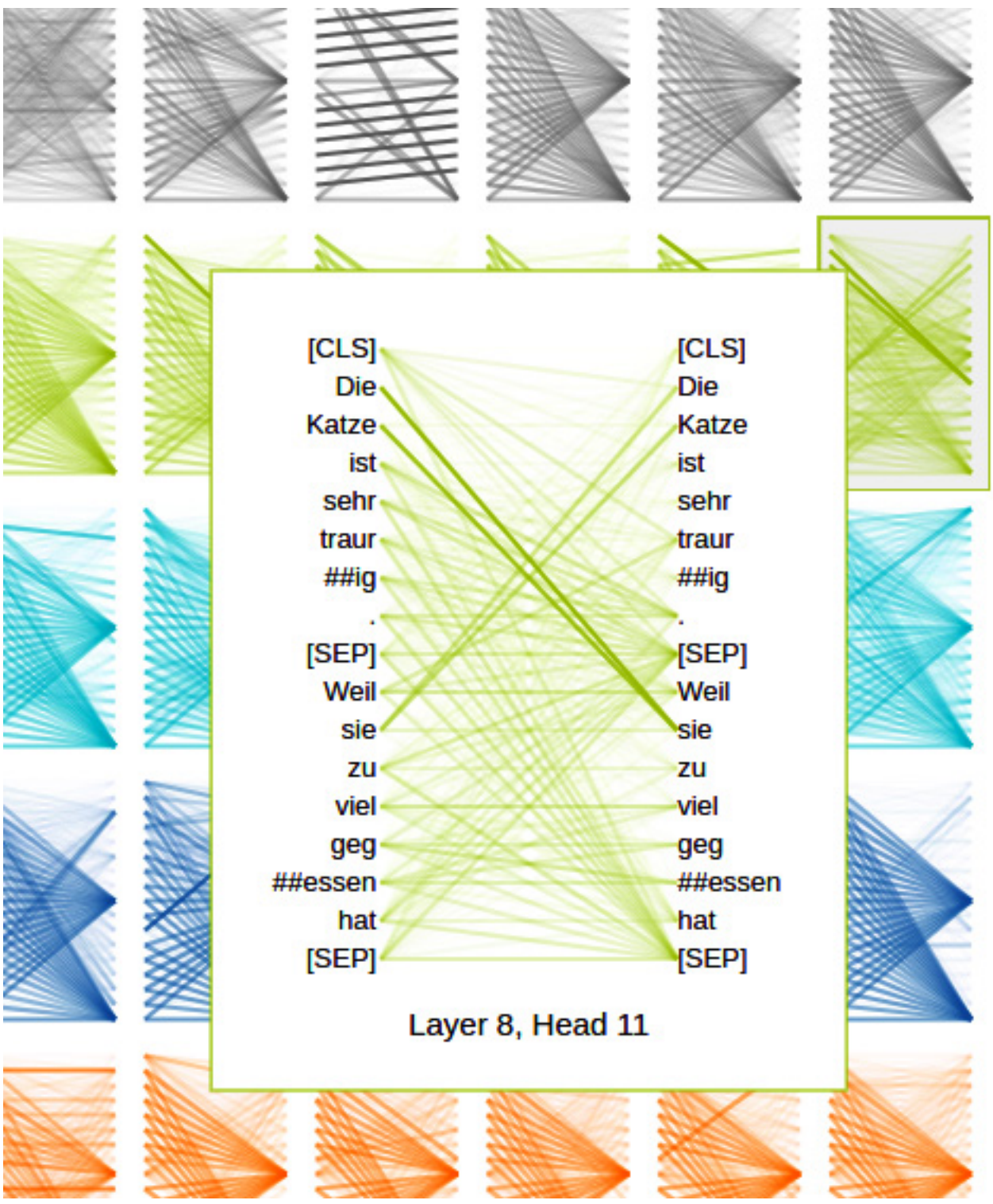


Figure 11.13 – Close-up of the <8,11> head in model view

You can again hover over the tokens and see the mappings.

I think that's enough work for the head view and the model view. Now, let's deconstruct the model with the help of the neuron view and try to understand how these heads calculate weights.

Neuron view

So far, we have visualized computed weights for a given input. The **neuron view** visualizes the neurons and the key vectors in a query and how the weights between tokens are computed based on interactions. We can trace the computation phase between any two tokens.

Again, we will load the German model and visualize the same-sentence pair we just worked with, to be coherent. We execute the following code:

```
from bertviz.Transformers_neuron_view import BertModel, BertTokenizer
from bertviz.neuron_view import show
model_path='bert-base-german-cased'
sentence_a = "Die Katze ist sehr traurig."
sentence_b = "Weil sie zu viel gegessen hat"
model = BertModel.from_pretrained(model_path, output_attentions=True)
tokenizer = BertTokenizer.from_pretrained(model_path)
model_type = 'bert'
show(model, model_type, tokenizer, sentence_a, sentence_b, layer=8, head=11)
```

This is the output:

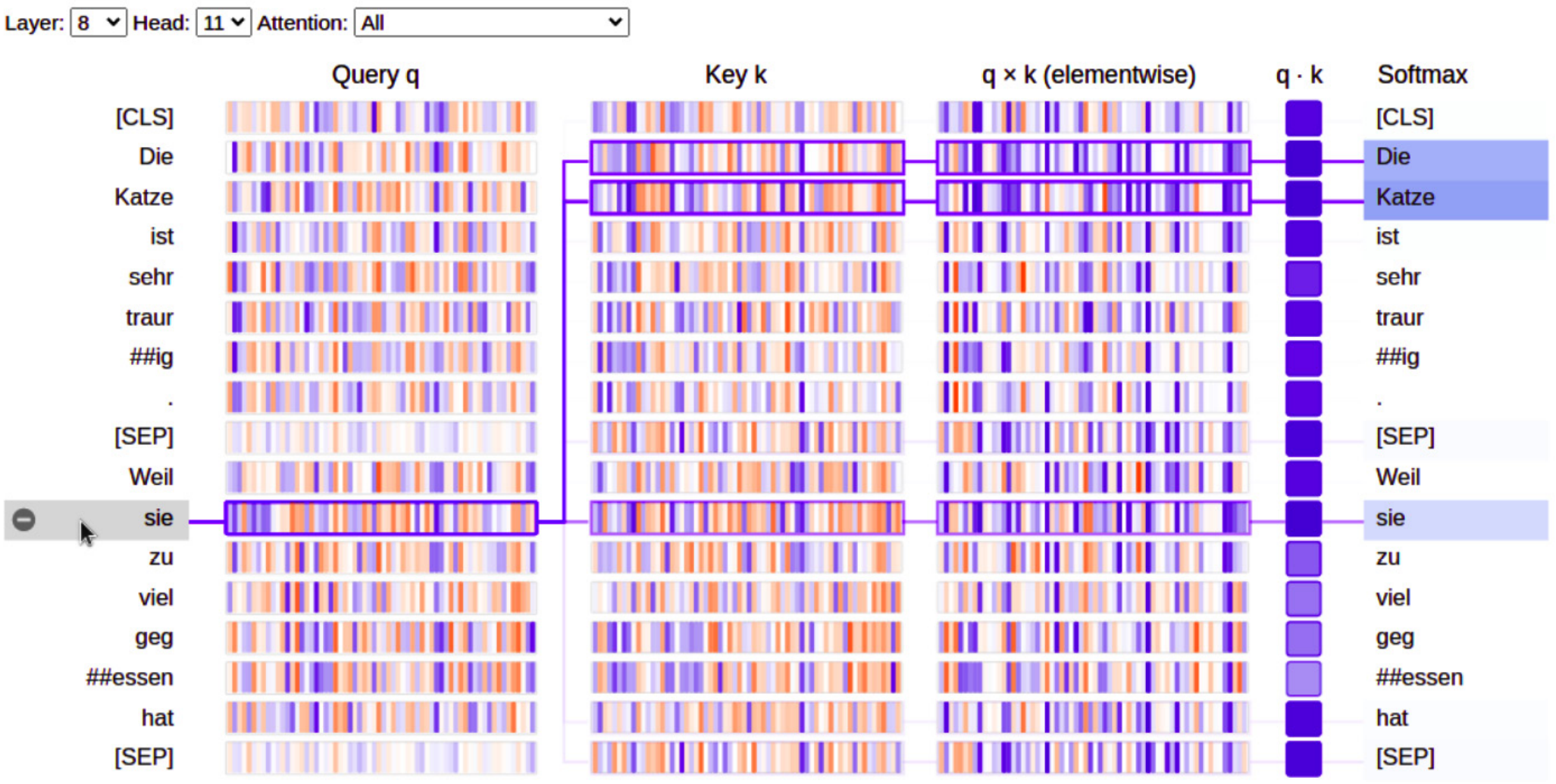


Figure 11.14 – Neuron view of the coreference relation pattern (head <8,11>)

The view helps us to trace the computation of attention from the **sie** token that we selected on the left to the other tokens on the right. Positive values are blue and negative values are orange. Color intensity represents the magnitude of the numerical value. The query of **sie** is very similar to the keys of **Die** and **Katze**. If you look at the patterns carefully, you will notice how similar these vectors are. Therefore, their dot product goes higher than the other comparison, which establishes strong attention between those tokens. We also trace the dot product and the Softmax function output as we go to the right. When clicking on the other tokens on the left, you can trace other computations as well.

Now, let's select a head-bearing next-token attention pattern for the same input, and trace it. To do so, we select the *<2,6>* head. In this pattern, virtually all the attention is focused on the next word. We click the **sie** token once again, as follows:

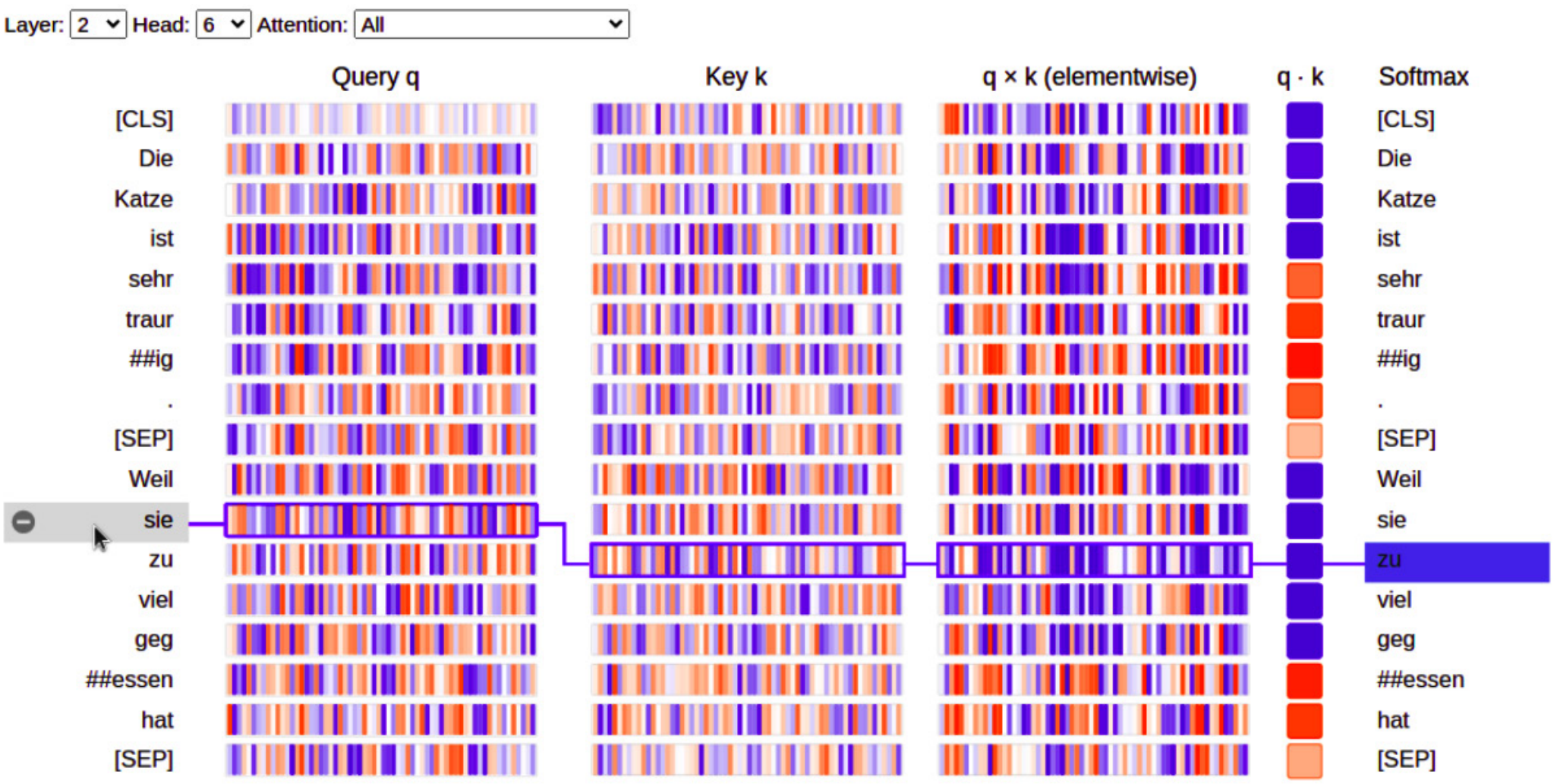


Figure 11.15 – Neuron view of next-token attention patterns (the <2,6> head)

Now, the **sie** token is focused on the next token instead of its own antecedent (**Die Katze**). When we carefully look at the query and the candidate keys, the most similar key to the query of **sie** is the next token, **zu**. Likewise, we observe how the dot product and Softmax function are applied in order.

In the next section, we will briefly talk about probing classifiers for interpreting Transformers.

Understanding the inner parts of BERT with probing classifiers

The opacity of what DL learns has led to a number of studies on the interpretation of such models. We attempt to answer the question of which parts of a Transformer model are responsible for certain language features, or which parts of the input lead the model to make a particular decision. To do so, other than visualizing internal representations, we can train a classifier on the representations to predict some external morphological, syntactic, or semantic properties. Hence, we can determine if we associate internal *representations* with external *properties*. The successful training of the model would be quantitative evidence of such an association—that is, the language model has learned information relevant for an external property. This approach is called a **probing-classifier** approach, which is a prominent analysis

technique in NLP and other DL studies. An attention-based probing classifier takes an attention map as input and predicts external properties such as coreference relations or head-modifier relations.

As seen in the preceding experiments, we get the self-attention weights for a given input with the `get_bert_attention()` function. Instead of visualizing these weights, we can directly transfer them to a classification pipeline. So, with supervision, we can determine which head is suitable for which semantic feature—for example, we can figure out which heads are suitable for coreference with labeled data.

Now, let's move on to the model-tracking part, which is crucial for building efficient models.

Tracking model metrics

So far, we have trained language models and simply analyzed the final results. We have not observed the training process or made a comparison of training using different options. In this section, we will briefly discuss how to monitor model training. For this, we will handle how to track the training of the models we developed before in *Chapter 5*, *Fine-Tuning Language Models for Text Classification*.

There are two important tools developed in this area—one is TensorBoard, and the other is W&B. With the former, we save the training results to a local drive and visualize them at the end of the experiment. With the latter, we are able to monitor the model-training progress live in a cloud platform.

This section will be a short introduction to these tools without going into much detail about them, as this is beyond the scope of this chapter.

Let's start with TensorBoard.

Tracking model training with TensorBoard

TensorBoard is a visualization tool specifically for DL experiments. It has many features such as tracking, training, projecting embeddings to a lower space, and visualizing model graphs. We mostly use it for tracking and visualizing metrics such as loss. Tracking a metric with TensorBoard is so easy for Transformers that adding a couple of lines to model-training code will be enough. Everything is kept almost the same.

Now, we will repeat the **Internet Movie Database (IMDb)** sentiment fine-tuning experiment we did in *Chapter 5*, *Fine-Tuning Language Models for Text Classification*, and will track the metrics. In that chapter, we already trained a sentiment model with an IMDb dataset consisting of a **4 kilo (4K)** training dataset, a 1K validation set, and a 1K test set. Now, we will adapt it to TensorBoard. For more details about TensorBoard, please visit https://www.tensorflow.org/tensorboard.

Let's begin:

1. First, we install TensorBoard if it is not already installed, like this:

```
!pip install tensorboard
```

2. Keeping the other code lines of IMDb sentiment analysis as-is from *Chapter 5*, *Fine-Tuning Language Models for Text Classification*, we set the training argument as follows:

```
from Transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir='./MyIMDBModel',
    do_train=True,
    do_eval=True,
    num_train_epochs=3,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=32,
    logging_strategy='steps',
    logging_dir='./logs',
    logging_steps=50,
    evaluation_strategy="steps",
    save_strategy="epoch",
    fp16=True,
    load_best_model_at_end=True
)
```

3. In the preceding code snippet, the value of `logging_dir` will soon be passed to TensorBoard as a parameter. As the training dataset size is 4K and the training batch size is 16, we have 250 steps (4K/16) for each epoch, which means 750 steps for three epochs.
4. We set `logging_steps` to 50, which is a sampling interval. As the interval is decreased, more details about where model performance rises or falls are recorded. We'll do another experiment later on, reducing this sampling interval at step 27.
5. Now, at every 50 steps, the model performance is measured in terms of the metrics that we define in `compute_metrics()`. The metrics to be measured

are Accuracy, F1, Precision, and Recall. As a result, we will have 15 (750/50) performance measurements to be recorded. When we run `trainer.train()`, this starts the training process and records the logs under the `logging_dir='./logs'` directory.

6. We set `load_best_model_at_end` to `True` so that the pipeline loads whichever checkpoint has the best performance in terms of loss. Once the training is completed, you will notice that the best model is loaded from `checkpoint-250` with a loss score of `0.263`.
7. Now, the only thing we need to do is to call the following code to launch TensorBoard:

```
%reload_ext tensorboard
%tensorboard --logdir logs
```

This is the output:

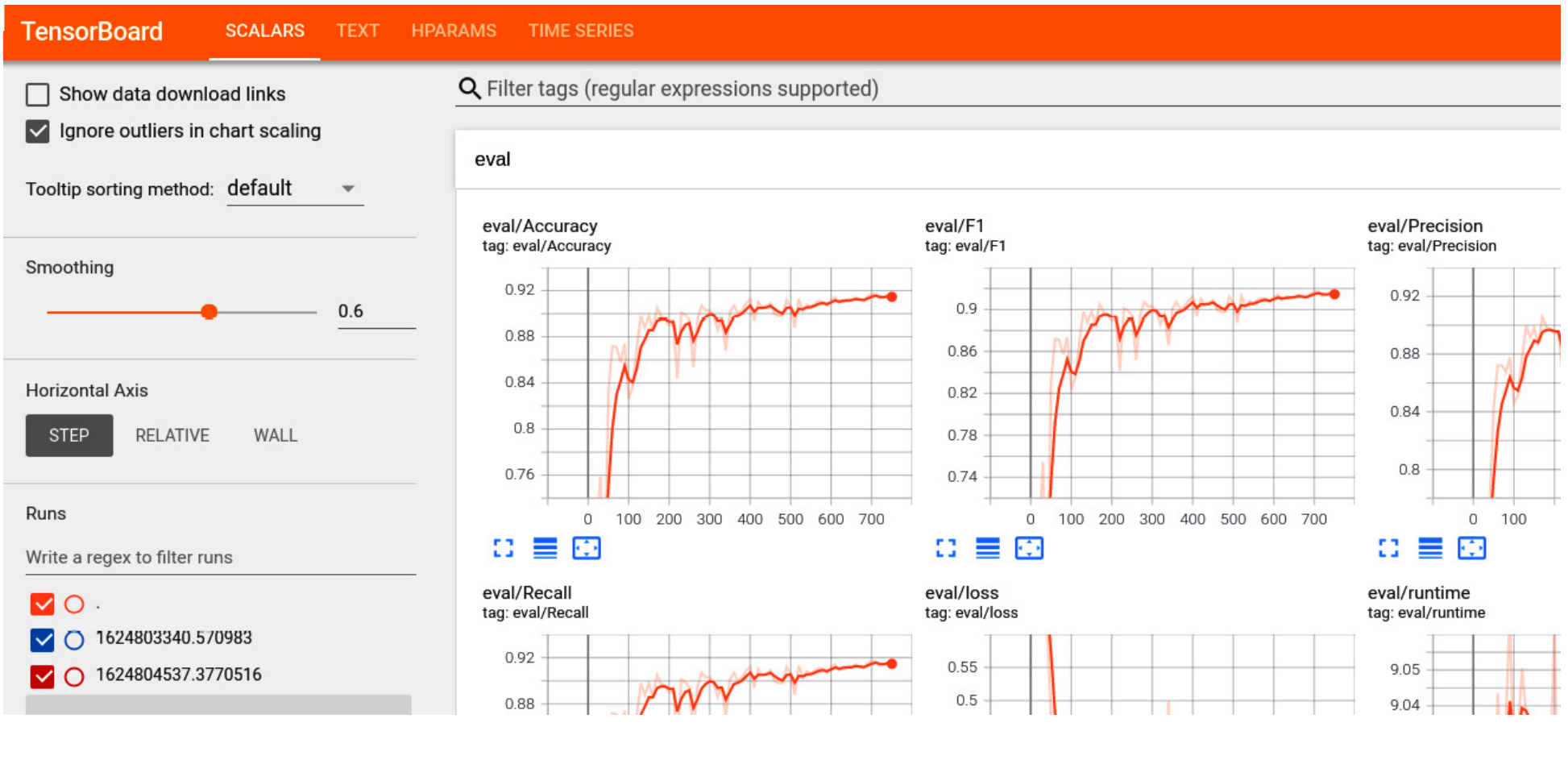


Figure 11.16 – TensorBoard visualization for training history

As you may have noticed, we can trace the metrics that we defined before. The horizontal axis goes from 0 to 750 steps, which is what we calculated before. We will not discuss TensorBoard in detail here. Let's just look at the **eval/loss** chart only. When you click on the maximization icon at the left-hand bottom corner, you will see the following chart:

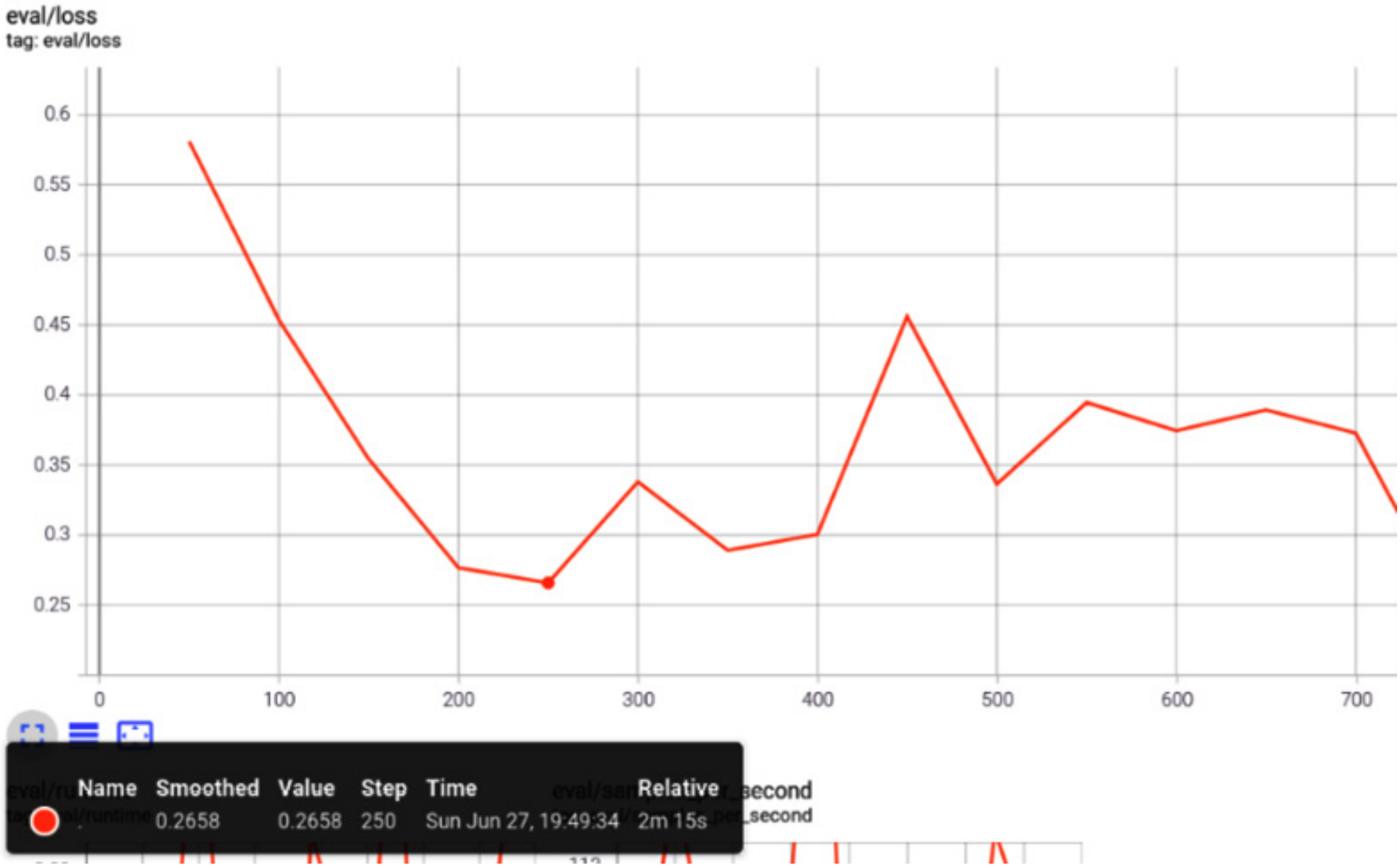


Figure 11.17 – TensorBoard eval/loss chart for logging steps of 50

In the preceding screenshot, we set the smoothing to 0 with the slider control on the left of the TensorBoard dashboard to see scores more precisely and focus on the global minimum. If your experiment has very high volatility, the smoothing feature can work well to see overall trends. It functions as a **Moving Average (MA)**. This chart supports our previous observation, in which the best loss measurement is **0.2658** at step **250**.

8. As `logging_steps` is set to **10**, we get a high resolution, as in the following screenshot. As a result, we will have 75 (750 steps/10 steps) performance measurements to be recorded. When we rerun the entire flow with this resolution, we get the best model at step 220, with a loss score of 0.238, which is better than the previous experiment. The result can be seen in the following screenshot. We naturally observe more fluctuations due to higher resolution:

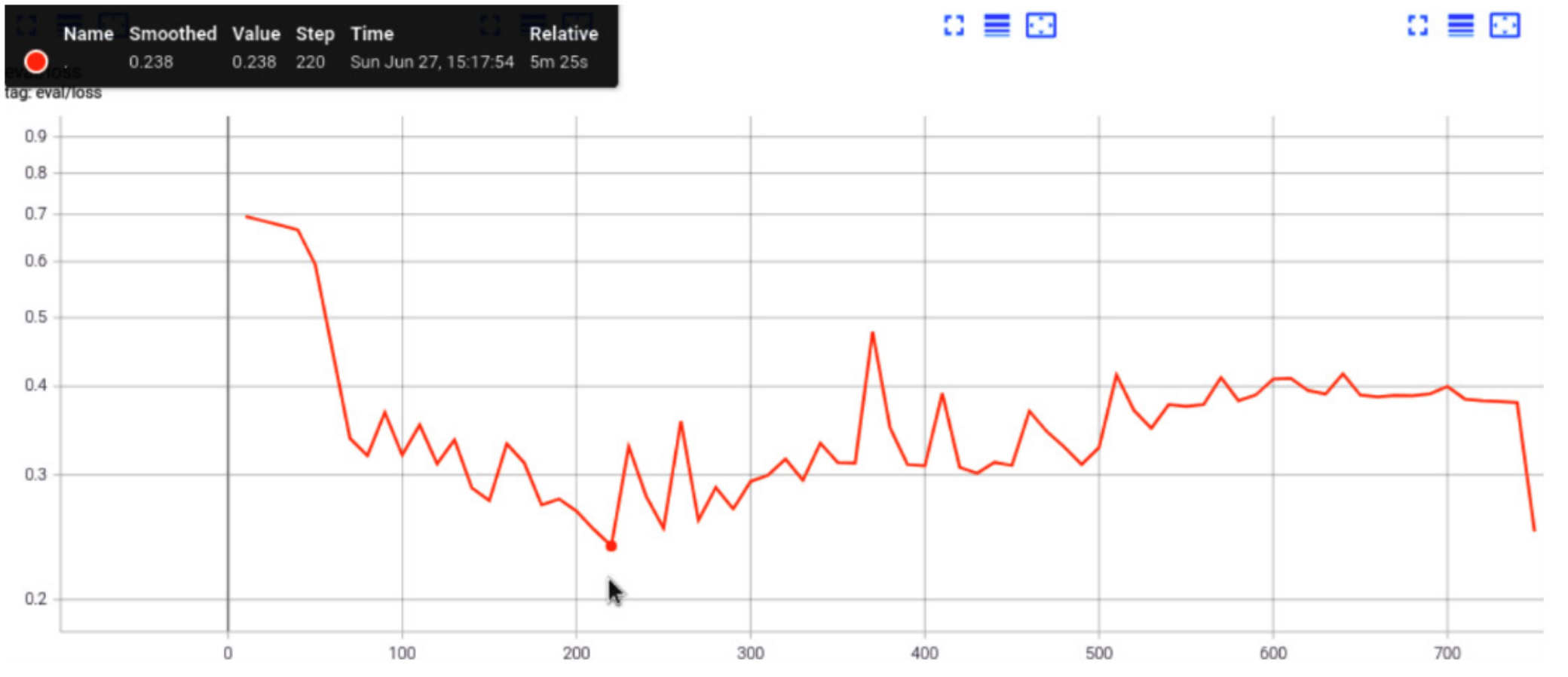


Figure 11.18 – Higher-resolution eval/loss chart for logging steps of 10

We are done with TensorBoard for now. Let's work with W&B!

Tracking model training live with W&B

W&B, unlike TensorBoard, provides a dashboard in a cloud platform, and we can trace and back up all experiments in a single hub. It also allows us to work with a team for development and sharing. The training code is run on our local machine, while the logs are kept in the W&B cloud. Most importantly, we can follow the training process live and share the result immediately with the community or team.

We can enable W&B for our experiments by making very small changes to our existing code:

1. First of all, we need to create an account in `wandb.ai`, then install the Python library, as follows:

```
!pip install wandb
```

2. Again, we will take the IMDb sentiment-analysis code and make minor changes to it. First, let's import the library and log in to wandB, as follows:

```
import wandb
!wandb login
```

wandb requests an API key that you can easily find at the following link: https://wandb.ai/authorize.

3. Alternatively, you can set the `WANDB_API_KEY` environment variable to your API key, as follows:

```
!export WANDB_API_KEY=e7d*********
```

4. Again, keeping the entire code as-is, we only add two parameters, `report_to="wandb"` and `run_name="..."`, to `TrainingArguments`, which enables logging in to W&B, as shown in the following code block:

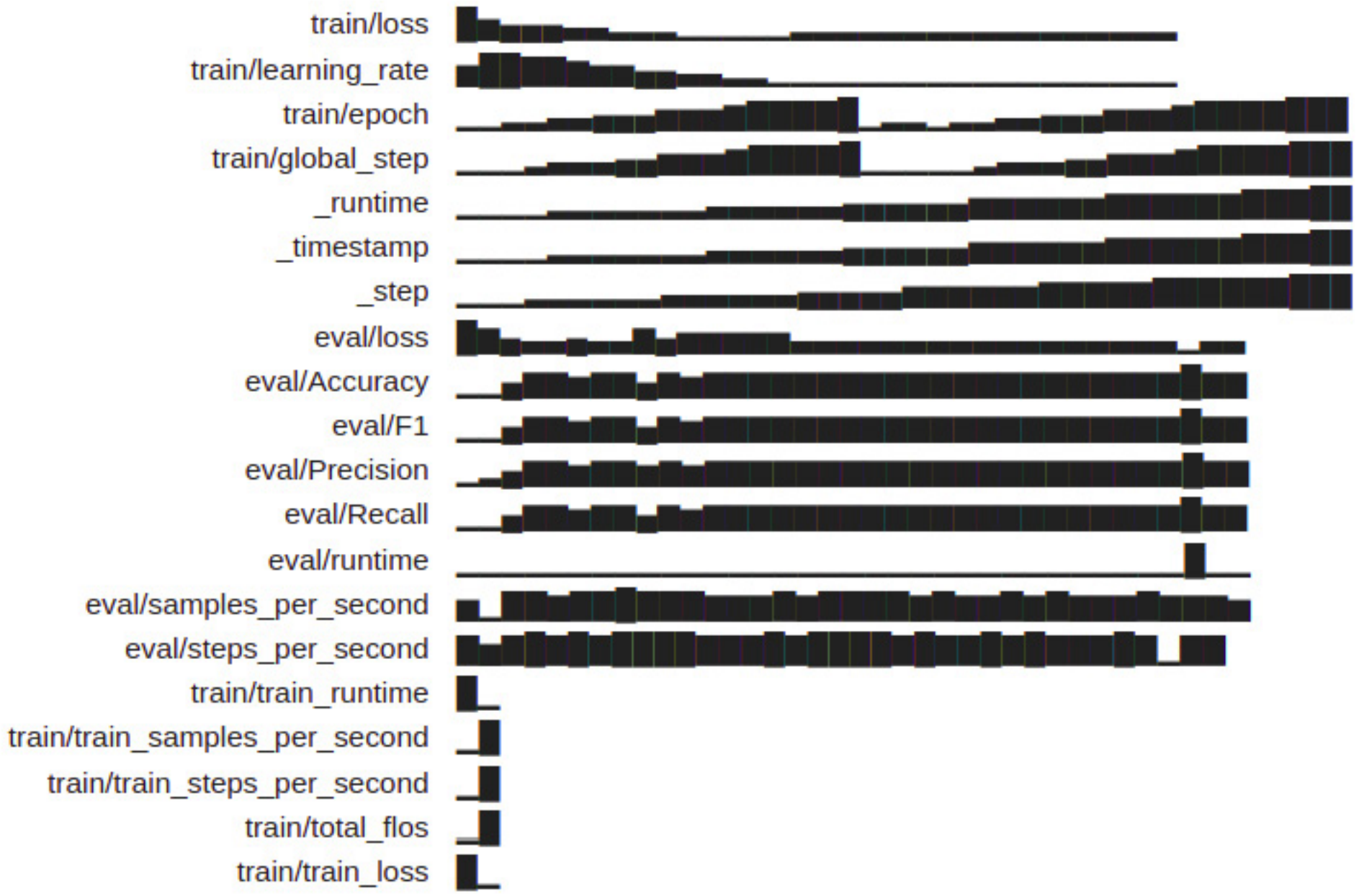
```
training_args = TrainingArguments(
    ...  the rest is same ...
    run_name="IMDB-batch-32-lr-5e-5",
    report_to="wandb"
)
```

5. Then, as soon as you call `trainer.train()`, logging starts on the cloud. After the call, please check the cloud dashboard and see how it changes. Once the `trainer.train()` call has completed successfully, we execute the following line to tell `wandB` we are done:

```
wandb.finish()
```

The execution also outputs run history locally, as follows:

Run history:



Synced 5 W&B file(s), 1 media file(s), 0 artifact file(s) and 0 other file(s)

Synced ./MyIMDBModel: https://wandb.ai/savasy/huggingface/runs/204bdria

Figure 11.19 – The local output of W&B

When you connect to the link provided by W&B, you will get to an interface that looks something like this:

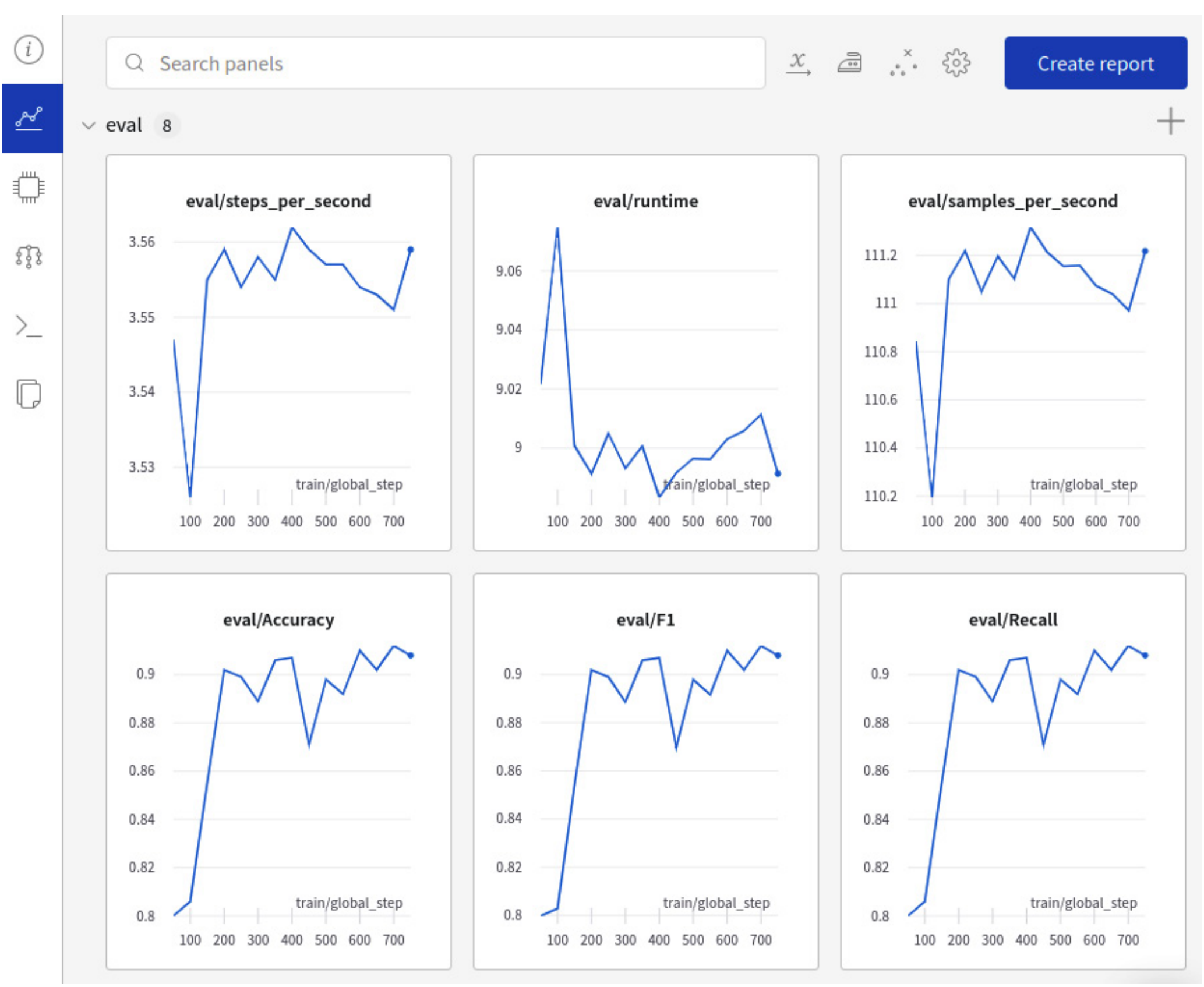


Figure 11.20 – The online visualization of a single run on the W&B dashboard

This visualization gives us a summarized performance result for a single run. As you see, we can trace the metrics that we defined in the `compute_metric()` function.

Now, let's take a look at the evaluation loss. The following screenshot shows exactly the same plot that TensorBoard provided, where the minimum loss is around 0.2658, occurring at step 250:

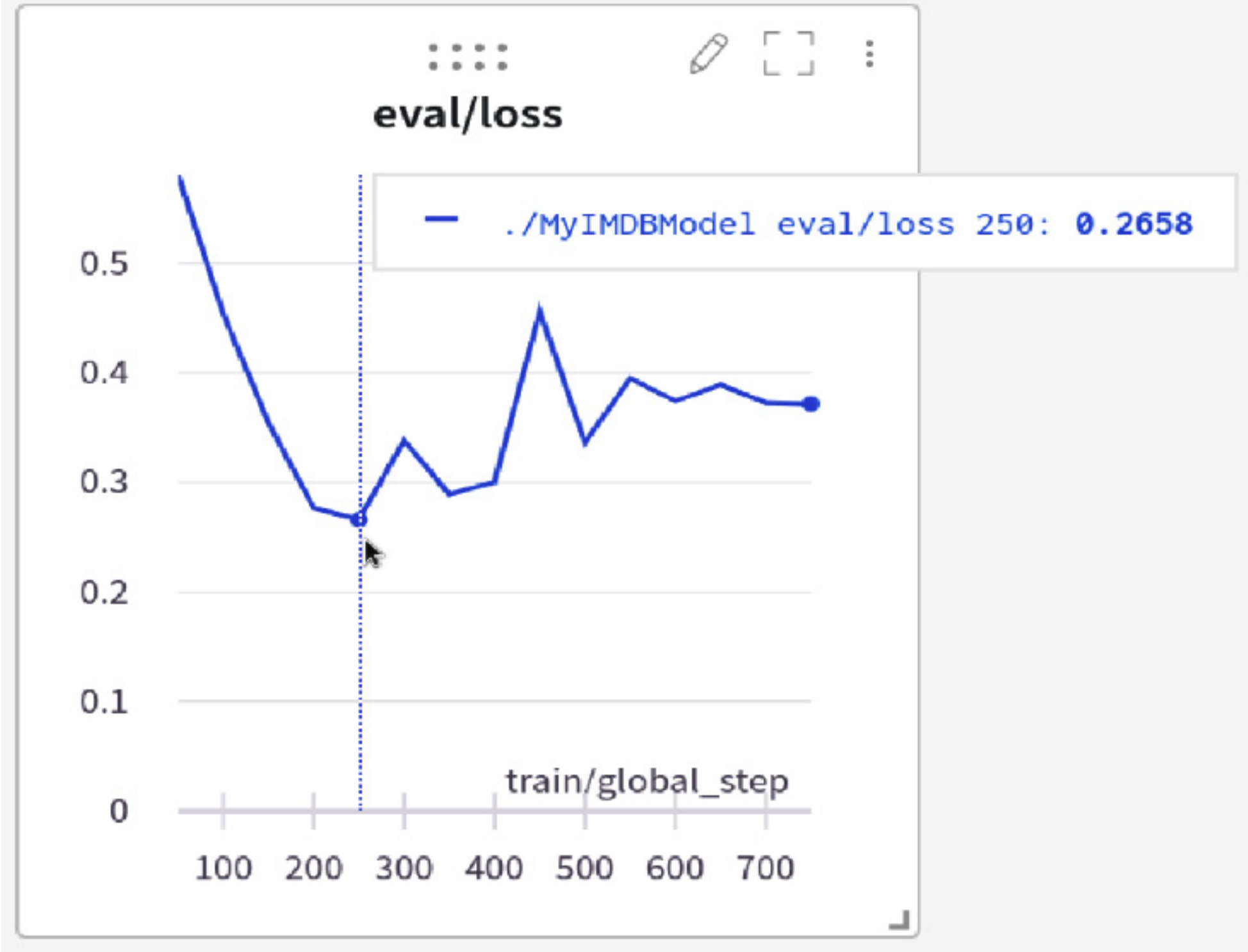


Figure 11.21 – The eval/loss plot of IMDb experiment on the W&B dashboard

We have only visualized a single run so far. W&B allows us to explore the results dynamically across lots of runs at once—for example, we can visualize the results of models using different hyperparameters such as learning rate or batch size. To do so, we instantiate a `TrainingArguments` object properly with another different hyperparameter setting and change `run_name="..."` accordingly for each run.

The following screenshot shows our several IMDb sentiment-analysis runs using different hyperparameters. We can also see the batch size and learning rate that we changed:

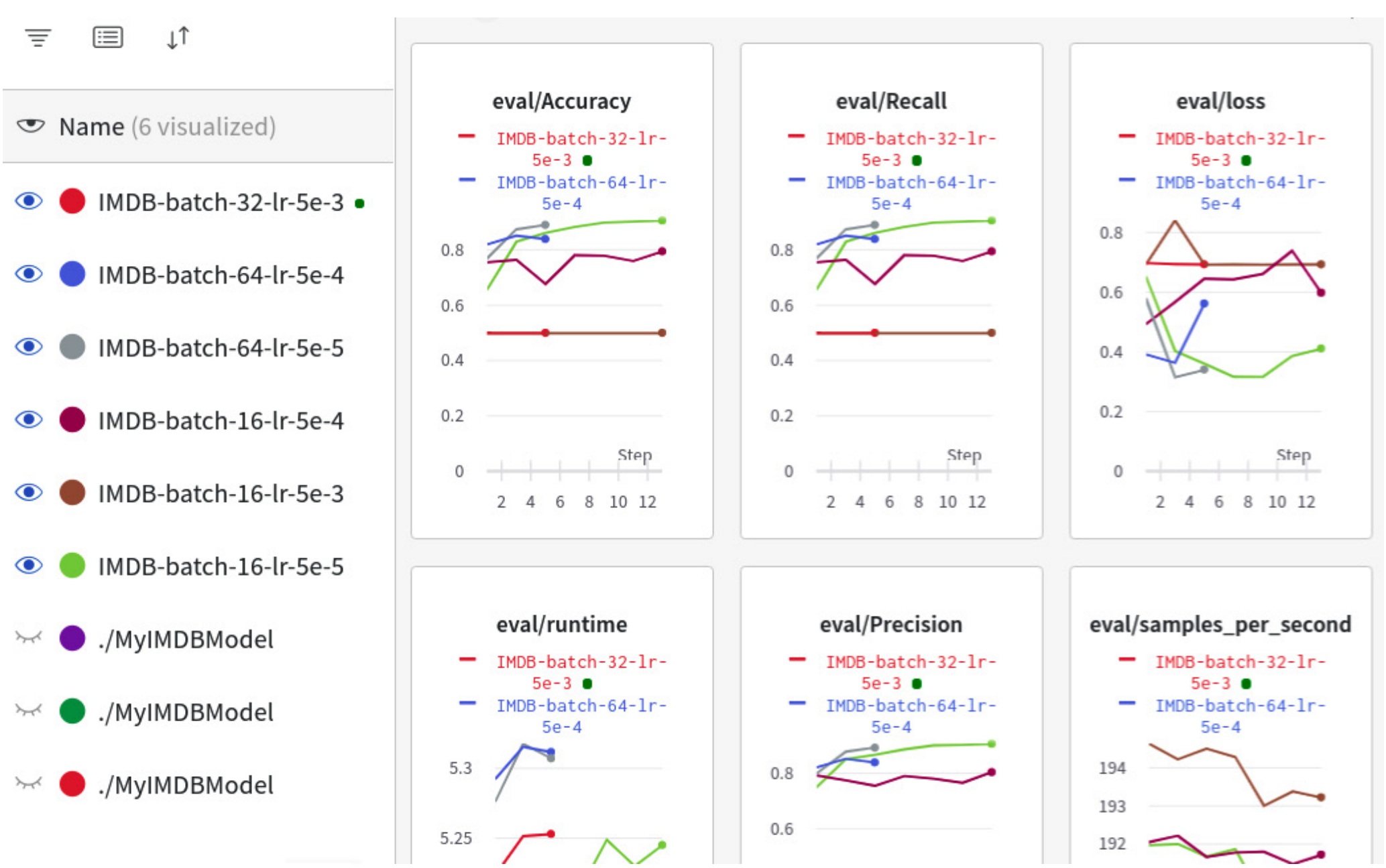


Figure 11.22 – Exploring the results across several runs on the W&B dashboard

W&B provides useful functionality—for instance, it automates hyperparameter optimization and searching the space of possible models, called W&B Sweeps. Other than that, it also provides system logs relating to **Graphics Processing Unit** (**GPU**) consumption, **Central Processing Unit** (**CPU**) utilization, and so on. For more detailed information, please check the following website: https://wandb.ai/home.

Well done! In the last section, *References*, we will focus more on technical tools, since it's crucial to use such utility tools to develop better models.

Summary

In this chapter, we introduced two different technical concepts: attention visualization and experiment tracking. We visualized attention heads with the exBERT online interface first. Then, we studied BertViz, where we wrote Python code to see three BertViz visualizations: head view, model view, and neuron view. The BertViz interface gave us more control so that we could work with different language models. Moreover, we were also able to observe how attention weights between tokens are computed. These tools provide us with important functions for interpretability and

exploitability. We also learned how to track our experiments to obtain higher-quality models and do error analysis. We utilized two tools to monitor training: TensorBoard and W&B. These tools were used to effectively track experiments and to optimize model training.

Congratulations! You've finished reading this book by demonstrating great perseverance and persistence throughout this journey. You can now feel confident as you are well equipped with the tools you need, and you are prepared for developing and implementing advanced NLP applications.

References

exBERT: A Visual Analysis Tool to Explore Learned Representations in Transformer Models, *Benjamin Hoover*, *Hendrik Strobelt*, *Sebastian Gehrmann*, *2019*.

Vig, J., 2019. A multiscale visualization of attention in the Transformer model. arXiv preprint arXiv:1906.05714.

Clark, K., Khandelwal, U., Levy, O. and *Manning, C.D., 2019. What does bert look at? An analysis of bert's attention. arXiv preprint arXiv:1906.04341.7*

Biewald, L., Experiment tracking with weights and biases, 2020. Software available from `wandb.com`, *2(5)*.

Rogers, A., Kovaleva, O. and *Rumshisky, A.,2020. A primer in BERTology: What we know about how BERT works. Transactions of the Association for Computational Linguistics, 8, pp.842-866*.

W&B: https://wandb.ai

TensorBoard: https://www.tensorflow.org/tensorboard

exBert—Hugging Face: https://huggingface.co/exbert

exBERT: https://exbert.net/

Why subscribe?

Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals

Improve your learning with Skill Plans built especially for you

Get a free eBook or video every month

Fully searchable for easy access to vital information

Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Previous Chapter

Next Chapter