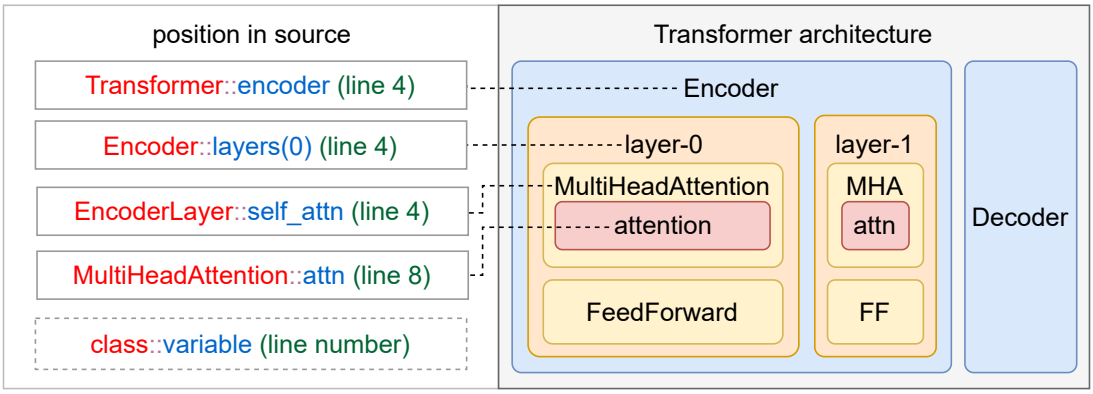
A step-by-step example for building an interface with NNVisBuilder

Contents

Chapter 1

Target Model and Layer

In this tutorial, we will design a simple interface to visualize the self-attention of the encoder in the Transformer model with NNVisBuilder. Self-attention is a crucial component of Transformer architecture, and it plays a critical role in modeling the relationships between the elements of a sequence. By analyzing the attention scores assigned to different input tokens, we can gain insights into how the model is able to capture and exploit the dependencies between the elements of a sequence. Before creating the visualization, we use a diagram combined with code to indicate the location of the network layer corresponding to the attention we analyze. The diagram is shown below:



The darker block on the right of the above diagram indicates the location of the attention we focus on in the network structure, while the lighter block shows the attribute names corresponding to different structures and the line numbers in the code for the corresponding class. We select the self-attention in the first layer of the encoder to analyze, and the corresponding layer name is `encoder.layers.0.self_attn.attn`. The source code for the `Transformer`, `Encoder`, `EncoderLayer`, and `MultiHeadAttention` classes are shown below:

```
class Transformer(nn.Module):
    def __init__(self, encoder, decoder, src_embed, tgt_embed, generator):
        super(Transformer, self).__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.src_embed = src_embed
        self.tgt_embed = tgt_embed
        self.generator = generator

    def encode(self, src, src_mask):
        return self.encoder(self.src_embed(src), src_mask)

    def decode(self, memory, src_mask, tgt, tgt_mask):
        return self.decoder(self.tgt_embed(tgt), memory, src_mask, tgt_mask)

    def forward(self, src, tgt, src_mask, tgt_mask):
        return self.decode(self.encode(src, src_mask), src_mask, tgt, tgt_mask)
```

```python
class Encoder(nn.Module):
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)
```

```python
class EncoderLayer(nn.Module):
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        # d_model
        self.size = size

    def forward(self, x, mask):
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

```python
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = nn.Softmax(dim=-1)

    def forward(self, query, key, value, mask=None):
        if mask is not None:
            mask = mask.unsqueeze(1)
        n_batches = query.size(0)
        query, key, value = [l(x).view(n_batches, -1, self.h, self.d_k).transpose(1, 2)
                             for l, x in zip(self.linears, (query, key, value))]
        d_k = query.size(-1)
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(d_k)
        if mask is not None:
            scores = scores.masked_fill(mask == 0, -1e9)
        attn = self.attn(scores)
        x = torch.matmul(attn, value)
        x = x.transpose(1, 2).contiguous().view(n_batches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Chapter 2

Building Visualization

In this chapter, we will show how to build an interface to analyze the attention of a pre-trained Transformer model step-by-step. In each section, we will present the code, the corresponding interface, and the corresponding data abstraction model (referred to as the flowchart below to distinguish it from the network model). If you are only interested in how to code with NNVisBuilder to build interfaces, you can ignore the content related to the flowchart.

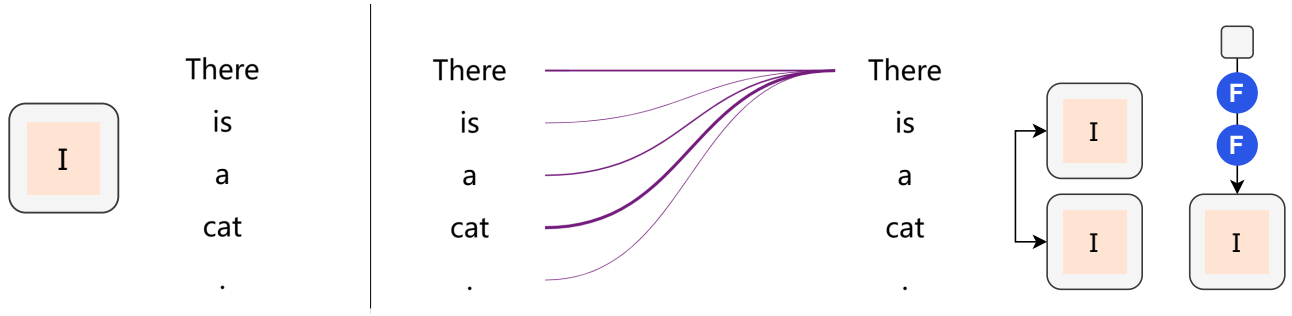
We intend to use a `LinkView` to visualize attention, where the width of the link reflects the attention score. This is an intuitive way of representing attention. Besides, we want the links to connect to the corresponding words, so we first add a `TextView` to display the words.

2.1 Add the first view: A TextView

Here's the code for creating the first view:

```
from NNVisBuilder import Builder
from NNVisBuilder.Views.Views import *
from NNVisBuilder.Views.Widgets import *
from NNVisBuilder.Data import *

src = ['There', 'is', 'a', 'cat', '.']
text_data = Data(src)
textview1 = TextView(text_data)      # or textview1 = TextView(src)
Builder().run()
```

In the above code, we create a `Data` object using the word list `src`, define a `TextView` based on this object, and then launch the interface by calling the `run` method of a `Builder` object. In this way, when we update the value of this `Data` object later, the display content of the `TextView` will also change accordingly. If we only need to create a static interface, we can directly use `src` to define the `TextView`, without the need to create a `Data` object. The interface and the flowchart are shown on the left-hand side of the following figure.



2.2 Add another TextView and a LinkView

Next, we add another `TextView` and the `LinkView` between them to obtain the interface shown on the right-hand side of the above figure. These two `TextViews` are bound to the same data, which can be considered as an identity transformation

relationship between the data of these two views. Therefore, in the flowchart, the two corresponding data rectangles are connected by a bidirectional data transformation arrow without any label. The following code adds another `TextView` to the right of the first `TextView`:

```
textview2 = TextView(text_data, textview1.align('right(300)'))
```

To add the `LinkView`, we need to obtain the attention data first. The following code shows how to obtain the attention data via the `Builder`.

```
builder = Builder(model)
layer = 'encoder.layers.0.self_attn.attn'
builder.add_hiddens(layer)
process(model, src, info)
# builder.embedding[layer].shape: (1, 8, 5, 5)
# Where 1 is the batch size, 8 is the number of heads, 5 is the number of words.
```

The attention data belongs to the hidden layer data of the model. In the above code, we define a `Builder` object with the model as a parameter and specified the hidden layer that needs to record data by `add_hidden`. After that, whenever input is fed into the model, the attention data will be recorded in the `embedding` attribute of `builder`. Later, we can launch the interface using the `run` method of `builder` without the need to create a new `Builder` object.

We use the `process` function to input `src` into the model. The info contains the mapping between tokens and token IDs. The `process` function converts the words in `src` into token IDs and feeds them into the model. The code inside this function is part of the original machine learning code. In other words, it is not new code that users need to add when using NNVisBuilderto build interfaces. The mapping between token IDs and tokens does not need to be converted into `Data` (tensor data) here. In fact, the data that needs to be converted into tensor data mainly includes two types: (1) data that may change and need to dynamically trigger view changes when it changes; (2) data that may change and need to dynamically trigger changes in other tensor data when it changes.

Excluding the batch size dimension, the obtained attention data is a three-order tensor. We choose to represent it as a `LinkView` corresponding to one of the heads and one of the words. To do this, we use two filters to filter the data, corresponding to the head and the word respectively. We temporarily choose the first head and the first word. When defining the `LinkView`, we need to specify the width and the start and end positions of the links. We use the filtered data as the widths and obtain link positions based on the two `Textviews`. The code is shown below:

```
attn_data = Data(builder.embeddings[layer][0])  # attn_data.size(): (8, 5, 5)
filter_head = Filter(value=0)
filter_word = Filter(value=0)
link_widths = attn_data.apply_transform(filter_head)
                .apply_transform(filter_word)   # link_widths.size(): (5,)
link_positions = Data([textview1.text_positions(), textview2.text_positions(0)])
# link_positions.size(): (2, 5), represent the start and end positions of the links
linkview = LinkView(link_widths, textview1.align('right(70)'), size=[200, -1],
            node_positions=link_positions)
```

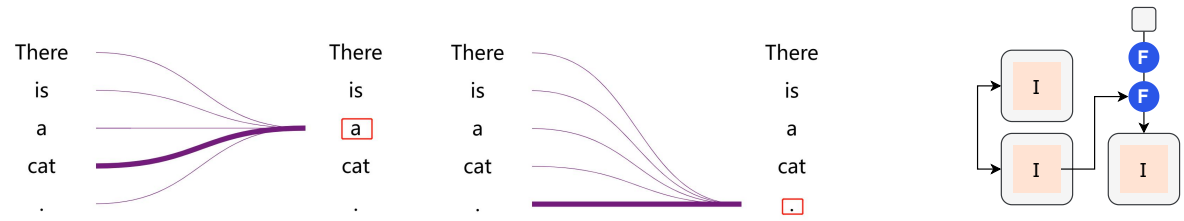
Only the main attribute (the first parameter of the constructor of a view) of each view will be displayed in the flowchart. The main attribute of `LinkView` is the link width. `node_positions` has the same shape as the main attribute, and when it changes, the main attribute must have changed as well. So we do not represent it in the flowchart.

2.3 Add interaction to choose the word

Next, we want to be able to select the attention visualization for which word by clicking on `textview2`. To do this, we need to add a click handler to `textview2` and modify the filter related to the word in it. Besides, we also need to modify the link positions so that the links can be connected to the correct word. The code is shown below:

```
def f(value):
    filter_word.update(value)
    link_positions.update([textview1.text_positions(), textview2.text_positions(value)])

textview2.onclick(f)
```

After this, clicking on the word in `textview2` will specify the attention visualization for that word, as shown in the figure below. We use red rectangles to highlight the primary interaction that causes the two figures to differ (applies to all the following figures as well).



2.4 Add a HeatMap and interaction

Next, we add a `HeatMap` to display the multi-head attention to one word. We modify the code that generates data for the `Linkview` and apply the two filters separately. The code is shown below:

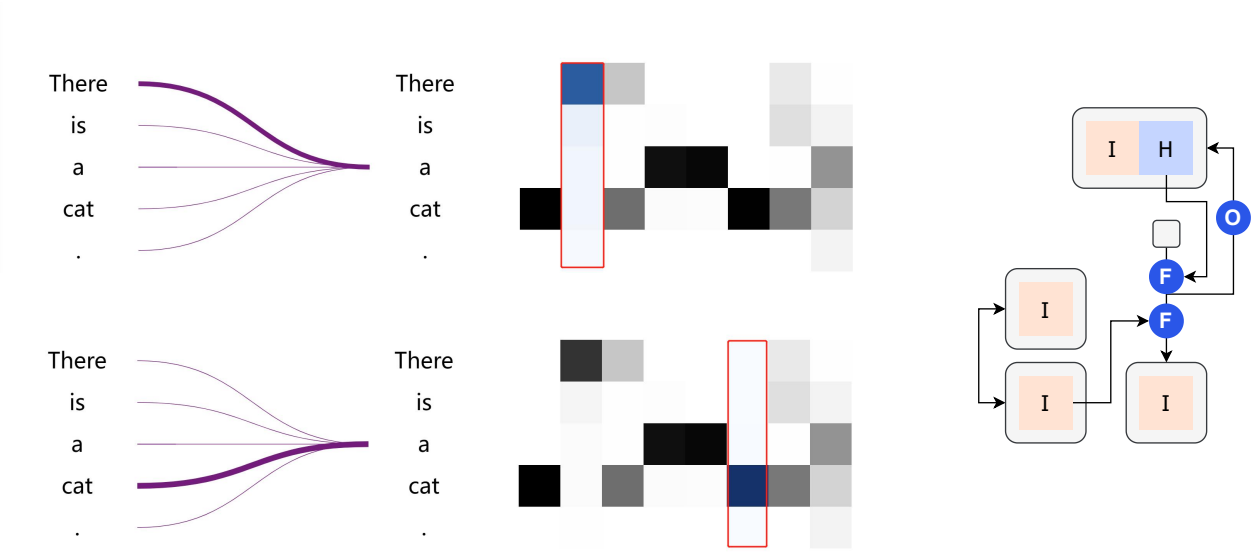
```
heatmap_data = attn_data.apply_transform(filter_word, dim=1)     # heatmap_data.size(): (8, 5)
link_widths = heatmap_data.apply_transform(filter_head)          # links_widths.size(): (5,)

heatmap = HeatMap(heatmap_data.apply_transform(Transpose()), textview2.align('right(100)'),
            cell_size=textview2.cell_size, cm='Greys', cm_highlight='Blues',
            highlighter=HighLighter('heatmap_cm', type=Type.Scalar), selector='col')
```

In the above code, we transpose the filtered data and use the `cell_size` of the `TextView` as the `cell_size` so that the `HeatMap` can be aligned with the `TextView`. At the same time, we specify the origin colormap and the colormap when highlighted, define the corresponding highlighter, and specify the selector as `col`, indicating that the column index will be returned when clicked.

Since we defined the highlighter, NNVisBuilderwill generate a default event handler for the `HeatMap`. This function modifies the subset information of the highlighter based on the value returned when clicked. We leverage this feature and define the mapping on the highlighter. Thus every time the `HeatMap` is clicked, this mapping will be executed. This way, we can omit the code that modifies the subset information of the highlighter.

```
def f(value):
    filter_head.update(value)
heatmap.highlighter.add_mapping(f)
```

The interface and the flowchart are shown below. Other transformation (marked with the blue O label) in the flowchart represents a transposition transformation.



2.5 Add an input widget

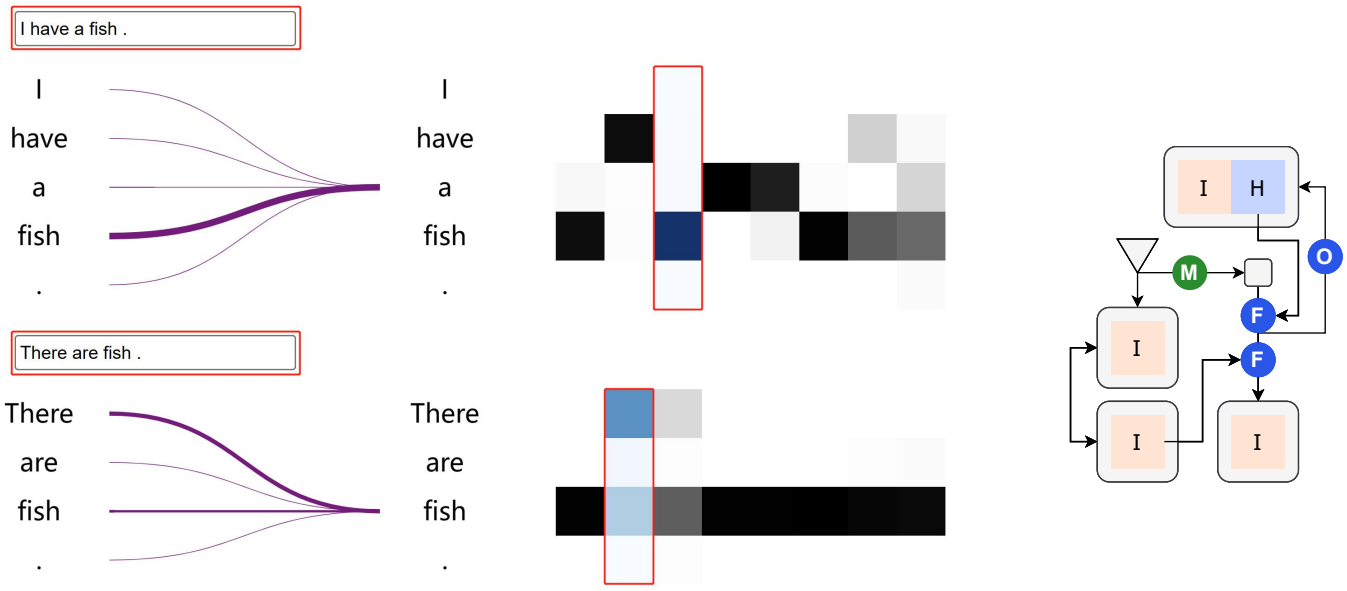
Next, we add an input widget to specify the input sentence. We want the attention values to be recalculated every time a new sentence is entered. To do this, we need to modify the calling of `add_hiddens` and set the `stage` to `all`, which means that the attention data will be recorded in both stages. NNVisBuilderdivides the recording of hidden states into two stages, and we used the data from the first stage when building the interface before. The first stage represents that the data is generated before the calling of `builder.run`, which can be uniformly specified before views are defined. In this section, we use the data from the second stage, which represents the data that will be generated after the calling of `builder.run` and in the event handler. The reason why NNVisBuilderdistinguishes between these two stages of data is that sometimes developers may want to record the data of specific layers in the second stage, but do not need to record these data in the first stage. For example, when analyzing a Transformer, we may need to record the encoder outputs of a large number of training samples but do not need to record the attention of these samples. Then we can specify to only record the attention in the second stage, which can save a lot of memory. The following code is the modified calling of `add_hiddens`:

```
builder.add_hiddens(layer, stage='all')
# The embedding of stage 1 will be saved in builder.embeddings_1
```

The code for defining the input widget and its event handler is shown below:

```
input_widget = Input(textview1.align('under(-40)'))

def f(value):
    src = value.split(' ')
    builder.reset_embedding()
    process(model, src, info)
    text_data.update(src)
    attn_data.update(builder.embeddings_1[layer][0])

input_widget.onclick(f)
```

In the event handler of the widget, we first reset the embedding, then feed the input sentence into the model, and modify the data corresponding to the sentence and attention. The interface and the flowchart are shown below. The green M label in the flowchart represents the mapping between the sentence and the attention data, which refers to the process of feeding a sentence into the model.



2.6 Add a BarChart

Finally, we add a `BarChart` to display the average attention of all heads for a specific word. Here is the code:

```
barchart_data = heatmap_data.apply_transform(Aggregation(op='avg'))
# heatmap_data.size(): (8, 5),   barchart_data.size(): (5,)
barchart = BarChart(barchart_data, textview1.align('under(200), right(20)'),
            size=[200, 180], titles=text_data, max_value=1)
```

In the above code, we obtain the data for the `BarChart` by averaging the data of the `HeatMap`. We also specified `titles` and `max_value`, which represent the texts displayed on the x-axis and the value corresponding to the maximum height of the `BarChart`, respectively. The interface and the flowchart are shown as the figures: