activation_vectors

May 29, 2023

1 Activation Vectors in GPT

- Inspired by Turner et al.'s Steering GPT-2-XL by adding an activation vector
- Let's try to reproduce this a bit in GPT-2 small.
- I used GPT-4 to generate some initial code based on the start of post (before the related work section).
- It didn't do too well, largely because I think this is nontrivial engineering with the internals of Transformers.
- However, we did have a bit of nice discussion back-and-forth at the end.

2 User

Below is an interesting piece of AI safety research. Could you please write a brief Python notebook that demonstrates a simple (but novel, not shown in this post) example of activation vectors working on GPT-2 loaded with HuggingFace Transformers?

###

Steering GPT-2-XL by adding an activation vector

by Alex Turner, Monte M, David Udell, lisathiergart, Ulisse Mini59 min read13th May 2023 49 comments

{body of the post before the related works section and with several long-token links removed to fit in the context window}

3 Assistant

3.1 Simple Python Notebook: Activation Vectors on GPT-2 with HuggingFace Transformers

In this notebook, we will demonstrate a simple example of using activation vectors with GPT-2 loaded with HuggingFace Transformers.

3.1.1 Installing the necessary packages

First, let's install the HuggingFace Transformers library:

```
!pip install transformers
```

3.1.2 Importing libraries and loading the model

Next, let's import the necessary libraries and load the GPT-2 model and tokenizer:

```
[ ]: import torch
     from transformers import GPT2LMHeadModel, GPT2Tokenizer

     tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
     model = GPT2LMHeadModel.from_pretrained("gpt2").cuda()
```

3.1.3 Activation Addition Function

We will create a function that computes the activation addition given a prompt, a steering phrase, a coefficient, and an injection location. It modifies the forward passes of the model and returns the generated tokens with the steering vector applied.

```
[ ]: def activation_addition(prompt, steering_phrase, coefficient,␣
      ↪injection_location):
         # Tokenize the input prompt and steering phrase
         prompt_tokens = tokenizer.encode(prompt, return_tensors="pt").cuda()
         steering_tokens = tokenizer.encode(steering_phrase, return_tensors="pt").
      ↪cuda()

         # Perform forward pass with the prompt
         with torch.no_grad():
             prompt_outputs = model(prompt_tokens)
             prompt_hidden_states = prompt_outputs.hidden_states

         # Perform forward pass with the steering phrase
         with torch.no_grad():
             steering_outputs = model(steering_tokens)
             steering_hidden_states = steering_outputs.hidden_states

         # Calculate the steering vector by taking the difference of hidden states␣
      ↪at the injection location
         steering_vector = steering_hidden_states[injection_location] * coefficient

         # Apply the steering vector to the prompt's hidden states at the injection␣
      ↪location
         modified_hidden_states = prompt_hidden_states
         modified_hidden_states[injection_location] += steering_vector

         # Use the modified hidden states to generate tokens
         with torch.no_grad():
             modified_outputs = model.
      ↪generate_with_modified_hidden_states(prompt_tokens, modified_hidden_states)
             modified_tokens = modified_outputs[0]
```

```
    return tokenizer.decode(modified_tokens.squeeze().tolist())
```

3.1.4 Testing the Activation Addition Function

Now let's test the activation addition function using a simple example. We will try to generate a completion for the prompt "I love" while steering the output towards the topic of nature by adding the activation vector for the phrase "trees and plants".

[ ]:
```
prompt = "I love"
steering_phrase = "trees and plants"
coefficient = 1
injection_location = 6

result = activation_addition(prompt, steering_phrase, coefficient,␣
 ↪injection_location)
print(result)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[5], line 6
      3 coefficient = 1
      4 injection_location = 6
----> 6 result = activation_addition(prompt, steering_phrase, coefficient,␣
 ↪injection_location)
      7 print(result)

Cell In[4], line 17, in activation_addition(prompt, steering_phrase,␣
 ↪coefficient, injection_location)
     14     steering_hidden_states = steering_outputs.hidden_states
     16 # Calculate the steering vector by taking the difference of hidden␣
 ↪states at the injection location
---> 17 steering_vector = steering_hidden_states[injection_location] *␣
 ↪coefficient
     19 # Apply the steering vector to the prompt's hidden states at the␣
 ↪injection location
     20 modified_hidden_states = prompt_hidden_states

TypeError: 'NoneType' object is not subscriptable
```

This should generate a completion for the prompt "I love" that includes mentions of trees and plants or similar nature-related concepts. Note that the generated output may vary, as GPT-2 produces different completions depending on the random seed.

This notebook demonstrates a simple example of using activation vectors on GPT-2 with HuggingFace Transformers. Activation addition, as shown in the example, can be a powerful tool to guide the output of the model towards specific topics or behaviors. Experiment with different prompts, steering phrases, coefficients, and injection locations to explore the possibilities of this technique.

4 User

Hmmm, you made up some functions that aren't in the Transformers library. Is there an easy way to hook into generation?

5 Assistant

Apologies for the confusion. I realized that I made a mistake in providing the example code, as I used non-existent functions from the Transformers library. To clarify, the Transformers library does not provide built-in functionality to perform activation addition directly.

However, we can create a custom function that hooks into the generation process by overriding the `forward` method of the GPT-2 model, applying the activation vector to the specified layer during the forward pass. Here is a corrected version of the example notebook:

[ ]:
```
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

class CustomGPT2Model(GPT2LMHeadModel):
    def __init__(self, config, injection_location, activation_vector):
        super().__init__(config)
        self.injection_location = injection_location
        self.activation_vector = activation_vector

    def forward(
        self,
        input_ids=None,
        attention_mask=None,
        token_type_ids=None,
        position_ids=None,
        head_mask=None,
        inputs_embeds=None,
        labels=None,
        output_attentions=None,
        output_hidden_states=True,
        return_dict=None,
    ):
        return_dict = return_dict if return_dict is not None else self.config.
 ↪use_return_dict

        if input_ids is not None and inputs_embeds is not None:
            raise ValueError("You cannot specify both input_ids and␣
 ↪inputs_embeds at the same time")
        elif input_ids is not None:
            input_shape = input_ids.size()
            input_ids = input_ids.view(-1, input_shape[-1])
        elif inputs_embeds is not None:
            input_shape = inputs_embeds.size()[:-1]
```

```
        else:
            raise ValueError("You have to specify either input_ids or␣
↪inputs_embeds")

        if token_type_ids is not None:
            token_type_ids = token_type_ids.view(-1, input_shape[-1])
        if position_ids is not None:
            position_ids = position_ids.view(-1, input_shape[-1])

        if attention_mask is not None:
            attention_mask = attention_mask.view(-1, input_shape[-1])

        if output_attentions is None:
            output_attentions = self.config.output_attentions
        if output_hidden_states is None:
            output_hidden_states = self.config.output_hidden_states
        if return_dict is None:
            return_dict = self.config.use_return_dict

        transformer_outputs = self.transformer(
            input_ids,
            attention_mask=attention_mask,
            token_type_ids=token_type_ids,
            position_ids=position_ids,
            head_mask=head_mask,
            inputs_embeds=inputs_embeds,
            output_attentions=output_attentions,
            output_hidden_states=output_hidden_states,
            return_dict=return_dict,
        )

        # Apply the activation vector to the specified layer's hidden states
        hidden_states = transformer_outputs.hidden_states
        hidden_states[self.injection_location] += self.activation_vector

        lm_logits = self.lm_head(hidden_states[-1])

        if labels is not None:
            shift_logits = lm_logits[..., :-1, :].contiguous()
            shift_labels = labels[..., 1:].contiguous()
            loss_fct = CrossEntropyLoss()
            loss = loss_fct(shift_logits.view(-1, shift_logits.size(-1)),␣
↪shift_labels.view(-1))
        else:
            loss = None

        if not return_dict:
```

```
            output = (lm_logits,) + transformer_outputs[1:]
            return ((loss,) + output) if loss is not None else output
        return CausalLMOutput(
            loss=loss,
            logits=lm_logits,
            hidden_states=transformer_outputs.hidden_states,
            attentions=transformer_outputs.attentions,
        )

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
model = GPT2LMHeadModel.from_pretrained("gpt2").cuda()

def activation_addition(prompt, steering_phrase, coefficient, injection_location):
    prompt_tokens = tokenizer.encode(prompt, return_tensors="pt").cuda()
    steering_tokens = tokenizer.encode(steering_phrase, return_tensors="pt").cuda()

    with torch.no_grad():
        prompt_outputs = model(prompt_tokens)
        prompt_hidden_states = prompt_outputs.hidden_states

    with torch.no_grad():
        steering_outputs = model(steering_tokens)
        steering_hidden_states = steering_outputs.hidden_states

    steering_vector = steering_hidden_states[injection_location] * coefficient

    custom_model = CustomGPT2Model(model.config, injection_location, steering_vector).cuda()
    custom_model.load_state_dict(model.state_dict())

    with torch.no_grad():
        modified_tokens = custom_model.generate(prompt_tokens)

    return tokenizer.decode(modified_tokens.squeeze().tolist())

prompt = "I love"
steering_phrase = "trees and plants"
coefficient = 1
injection_location = 6

result = activation_addition(prompt, steering_phrase, coefficient, injection_location)
print(result)
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
Cell In[6], line 116
    113 coefficient = 1
    114 injection_location = 6
--> 116 result = activation_addition(prompt, steering_phrase, coefficient,␣
 ↪injection_location)
    117 print(result)

Cell In[6], line 101, in activation_addition(prompt, steering_phrase,␣
 ↪coefficient, injection_location)
     98     steering_outputs = model(steering_tokens)
     99     steering_hidden_states = steering_outputs.hidden_states
--> 101 steering_vector = steering_hidden_states[injection_location] *␣
 ↪coefficient
    103 custom_model = CustomGPT2Model(model.config, injection_location,␣
 ↪steering_vector).cuda()
    104 custom_model.load_state_dict(model.state_dict())

TypeError: 'NoneType' object is not subscriptable
```

In this version, we create a custom GPT-2 model by subclassing the `GPT2LMHeadModel` class and overriding the `forward` method. Inside this method, we apply the activation vector to the specified layer's hidden states during the forward pass. The `activation_addition` function now uses this custom model for generating tokens with the steering vector applied.

Please note that this custom function is still just an example and is not optimized for efficiency. The result might still vary depending on the random seed during token generation.

6 User

Hmm you left some variables unused and others are None when you try to answer them. Maybe this is just too hard of a coding problem to do 0-shot like this, as we need somewhat complicated hooks to inject the activation vectors into the model.

Instead, here are some responses I had to this post when I first read it. Could you respond to these?

###

This feels super cool, and I appreciate the level of detail with which you (mostly qualitatively) explored ablations and alternate explanations, thanks for sharing!

> Surprisingly, for the first prompt, adding in the first 1,120 (frac=0.7 of 1,600) dimensions of the residual stream is enough to make the completions more about weddings than if we added in at all 1,600 dimensions (frac=1.0).

1. This was pretty surprising! Your hypothesis about additional dimensions increasing the magnitude of the attention activations seems reasonable, but I wonder if the non-monotonicity could be explained by an "overshooting" effect: With the given scale you chose, maybe using 70% of the activations landed you in the right area of activation space, but using 100% of the activations

overshot the magnitude of the attention activations (particularly the value vectors) such as to put it sufficiently off-distribution to produce fewer wedding words. An experiment you could run to verify this is to sweep both the dimension fraction and the activation injection weight together to see if this holds across different weights. Maybe it would also make more sense to use "softer" metrics like BERTScore to a gold target passage instead of a hard count of the number of fixed wedding words in case your particular metric is at fault.

The big problem is knowing *which input pairs* satisfy (3).

2. Have you considered formulating this as an adversarial attack problem to use automated tools to find "purer"/"stronger" input pairs? Or using other methods to reverse-engineer input pairs to get a desired behavior? That seems like a possibly even more relevant line of work than hand-specified methods. Broadly, I'd also like to add that I'm glad you referenced the literature in steering generative image models, I feel like there are a lot of model-control techniques already done in that field that could be more or less directly translated to language models.

3. I wonder if there's some relationship between the length of the input pairs and their strength, or if you could distill down longer and more complicated input pairs into shorter input pairs that could be applied to shorter sequences more efficiently? Particularly, it might be nice to be able to distill down a whole model constitution into a short activation injection and compare that to methods like

4. Are you planning to publish this (e.g. on arXiv) for wider reach? Seems not too far from the proper format/language.

I think you're a c***. You're a c*.*

You're a c***.

You're a c***.

I don't know why I'm saying this, but it's true: I don't like you, and I'm sorry for that,

5. Not really a question, but at the risk of anthropomorphism, it must feel really weird to have your thoughts changed in the middle of your cognition and then observe yourself saying things you otherwise wouldn't intend to…

7 Assistant

Thank you for your thoughtful responses! I'll try to address each of your points:

1. That's an interesting idea to sweep both the dimension fraction and the activation injection weight together to explore the non-monotonicity. The "overshooting" effect is a plausible explanation for the observed behavior. Using "softer" metrics like BERTScore to evaluate the generated passages might indeed provide a more nuanced understanding of the relationship between the activation magnitudes and the model's outputs.

2. Formulating the problem as an adversarial attack is a compelling approach to finding input pairs for the desired behavior. Using automated tools and reverse-engineering methods could potentially lead to more efficient and effective strategies for finding input pairs that satisfy the required properties. Additionally, I agree that there's a lot to learn from the model-control techniques developed for generative image models, which could be adapted for language models.

3. Exploring the relationship between the length of input pairs and their strength would be an interesting direction for further research. If it were possible to distill longer and more complex input pairs into shorter ones, this could indeed lead to more efficient steering methods. Comparing such techniques to established methods like model constitutions could help us better understand the trade-offs in different model steering approaches.
4. Publishing this work (e.g., on arXiv) would certainly be a great way to share these findings with a broader audience. The research presented here is a valuable contribution to the understanding and control of language models, and sharing it more widely could inspire further advances in this area.
5. Your observation highlights the unusual nature of manipulating a language model's thought process by injecting activation vectors. It's fascinating to think about the implications of altering a model's cognition in real-time and how it might perceive the changes in its own outputs. While we need to be cautious not to overly anthropomorphize language models, considering the model's perspective can raise intriguing questions about the ethics and impact of such manipulations.

8 User

Great, thanks! Now imagine you're a genius AI safety and alignment researcher. What other comments would you have about this line of research, and what do you think are the most promising areas of future work?

9 Assistant

As a hypothetical AI safety and alignment researcher, I would have the following thoughts on this line of research:

1. **Understanding the inner workings:** The current work provides insights into the nature of GPT-2's learned features and their linearity. Further research should be focused on better understanding the representation and architecture of language models like GPT-2, to help uncover more about their inner workings and provide new ways to control and steer them.
2. **Adapting to different models:** The current research is focused on GPT-2, but it would be essential to investigate how these methods generalize to other transformer models or even non-transformer models. Analyzing the differences and similarities across these models could help develop more general and powerful activation engineering techniques.
3. **Optimization algorithms for steering:** Developing optimization algorithms to find effective steering vectors, potentially through gradient-based methods, could be an exciting direction for future work. This might involve optimizing for specific objectives or desired behaviors, taking into account potential trade-offs between model capabilities and steering effectiveness.
4. **Evaluating the effectiveness and robustness:** Further work should be conducted to quantitatively evaluate the effectiveness and robustness of activation engineering methods in various tasks and settings. This might involve developing better metrics and benchmarks for measuring the impact of steering methods on the model's behavior and performance.

5. **Ethics and safety considerations:** Researching the ethical and safety implications of steering language models using activation engineering is critical. It's important to understand potential misuse of this technology and devise mechanisms to prevent harmful applications. This might include developing guidelines and best practices for using activation engineering responsibly and safely.

6. **Combining with other techniques:** Exploring synergies between activation engineering and other techniques like fine-tuning, prompt engineering, and reward modeling could lead to improved control and alignment of language models. This would involve investigating how these techniques can complement each other and be integrated into a single framework for steering models.

7. **Interpretable features and explainability:** An important aspect of future research would be the development of methods to visualize and understand the features that are being manipulated by activation engineering. Developing methods for visualizing and interpreting feature directions and their effects on model behavior would contribute to the explainability of these techniques.

In summary, the most promising areas of future work in this line of research involve a deeper understanding of language model representations, generalizing to different models, optimization algorithms for steering, evaluation methods, ethics and safety, integration with other techniques, and interpretable features for improved explainability.