CARTE-Enbridge Bootcamp

Lab 5-1

How to generate text: using different decoding methods for language generation with Transformers

Introduction

In recent years, we've seen a lot of progress in the field of natural language processing, especially in generating text that looks like it was written by a human.

The ability to generate human-like text isn't just because these models are big and powerful. The way they choose the next word to generate—known as "decoding"—also plays a big role.

In this notebook, we'll walk you through how to generate text using some of the most popular models out there, like GPT-2. We'll also introduce you to different techniques for choosing the next word in a sentence, including methods like "Greedy search," "Beam search," "Top-K sampling," and "Top-p sampling."

Don't worry if these terms sound complicated! We'll break them down into simpler language as we go along.

Before we get started, we'll set up the 'transformers' library, which many of you might have heard of. It's a handy tool that makes text generation a breeze. For our examples, we'll use a well-known model called GPT-2.

In [1]:
```python
# Test whether we are running in Google Colab
IN_COLAB = "google.colab" in str(get_ipython())
print("Running in Colab: ", IN_COLAB)
```

```
Running in Colab:  False
```

In [2]:
```python
!pip install -q git+https://github.com/huggingface/transformers.git
#
# if not IN_COLAB:
#     !pip install -q tensorflow>=2.1.0
```

In this cell, we're setting up the tools we'll need. We import TensorFlow, a library for machine learning, and specific functions from the 'transformers' library. Then we choose a pre-trained GPT-2 model and a 'tokenizer'—a tool that helps us prepare text for the model.

Here's what each line does:

- Import necessary libraries: `tensorflow` for machine learning, `TFGPT2LMHeadModel` and `GPT2Tokenizer` from 'transformers'.
- Specify which model to use: here, it's 'gpt2'.
- Load the tokenizer: prepares text for the model.
- Load the model: fetches the pre-trained GPT-2 model.

Once we've run this cell, we'll be all set to start generating text!

In [3]:
```python
import tensorflow as tf
from transformers import TFGPT2LMHeadModel, GPT2Tokenizer

# Specify the model name
model_name = "gpt2"

# Load pre-trained tokenizer
tokenizer = GPT2Tokenizer.from_pretrained(model_name)

# Load pre-trained model and specify EOS token as PAD token to avoid warning
model = TFGPT2LMHeadModel.from_pretrained(
    model_name, pad_token_id=tokenizer.eos_token_id
)
```

```
2023-11-13 09:55:25.507098: I tensorflow/core/util/port.cc:110] oneDNN custo
m operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn t
hem off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2023-11-13 09:55:25.535927: I tensorflow/core/platform/cpu_feature_guard.cc:
182] This TensorFlow binary is optimized to use available CPU instructions i
n performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AVX512F AVX512_
VNNI FMA, in other operations, rebuild TensorFlow with the appropriate compi
ler flags.
2023-11-13 09:55:27.159351: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.176218: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.176375: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.177491: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.177597: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.177676: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.246153: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.246294: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.246377: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_gpu_executor.cc:995] successful NUMA node read from SysFS had negative v
alue (-1), but there must be at least one NUMA node, so returning NUMA node
zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/
ABI/testing/sysfs-bus-pci#L344-L355
2023-11-13 09:55:27.246444: I tensorflow/core/common_runtime/gpu/gpu_device.
```

```
cc:1639] Created device /job:localhost/replica:0/task:0/device:GPU:0 with 10
392 MB memory:  -> device: 0, name: NVIDIA GeForce RTX 3060, pci bus id: 000
0:02:00.0, compute capability: 8.6
2023-11-13 09:55:27.486642: I tensorflow/compiler/xla/stream_executor/cuda/c
uda_blas.cc:606] TensorFloat-32 will be used for the matrix multiplication.
This will only be logged once.
All PyTorch model weights were used when initializing TFGPT2LMHeadModel.

All the weights of TFGPT2LMHeadModel were initialized from the PyTorch mode
l.
If your task is similar to the task the model of the checkpoint was trained
on, you can already use TFGPT2LMHeadModel for predictions without further tr
aining.
```

Greedy Search

Imagine you're building a sentence and, at each step, you can only choose the next word that is most likely to come next. This is the basic idea behind Greedy Search. It's like playing a word game where you always pick the safest, most obvious next word.

In this example, we start with the word "The," and the next word that is most likely to follow is "nice," and so on. So, our final sentence might be "The nice woman."

Now, let's try generating some text using Greedy Search. We'll start with the phrase "I enjoy walking with my cute dog" and see what comes next when using this method.

In [4]:
```python
def generate_text(model, tokenizer, text, max_length=50):
    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text
    output = model.generate(input_ids, max_length=max_length)

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


input_text = "I enjoy walking with my cute dog"
print(generate_text(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to w
alk with my dog. I'm not sure if I'll ever be able to walk with my dog.

I'm not sure if I'll
```

Great, we've generated our first piece of text! However, you might have noticed that the generated text can sometimes be repetitive. This is a common issue when using Greedy Search.

The main limitation of this method is that it can miss out on better word choices that come after less obvious ones. In simpler terms, it might skip a great word just because it's focused on picking the most likely next word at each step.

Don't worry, though; there are other methods that help us overcome these limitations. Next, we'll explore one called 'Beam Search.'

Beam Search

If Greedy Search is like playing a word game by always picking the safest bet, then Beam Search is like having a few bets running at the same time. Instead of just keeping the single most likely next word, Beam Search keeps track of a few possibilities, known as 'beams.'

Imagine we start with the word 'The.' Beam Search might keep two possibilities going at once—like 'The nice' and 'The dog.' As it moves on to the next word, it picks the combination that overall makes the most sense. In this way, it can discover word choices that Greedy Search might miss.

Now, let's see how to use Beam Search in practice. We'll change a couple of settings to make sure it works well for our example.

In [5]:
```python
def generate_text_beam_search(
    model, tokenizer, text, max_length=50, num_beams=5, early_stopping=True
):
    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using beam search
    output = model.generate(
        input_ids,
        max_length=max_length,
        num_beams=num_beams,
        early_stopping=early_stopping,
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_beam_search(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to w
alk with him again.

I'm not sure if I'll ever be able to walk with him again. I'm not sure if
I'll
```

Even though Beam Search gives us a more fluent result, we still run into the problem of repeating words or phrases. One way to tackle this is by using something called 'n-grams,' which are basically just sequences of words that appear next to each other.

In simpler terms, we can tell the model to avoid repeating the same sequence of words twice. This helps make the generated text more varied and interesting.

Let's see how this works in practice by setting a rule that prevents the same two words from appearing next to each other more than once.

In [6]:
```python
def generate_text_no_repeat_ngram(
    model,
    tokenizer,
    text,
    max_length=50,
    num_beams=5,
    no_repeat_ngram_size=2,
    early_stopping=True,
):
    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using beam search with no_repeat_ngram_size
    output = model.generate(
        input_ids,
        max_length=max_length,
        num_beams=num_beams,
        no_repeat_ngram_size=no_repeat_ngram_size,
        early_stopping=early_stopping,
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_no_repeat_ngram(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog, but I'm not sure if I'll ever be able to w
alk with him again.

I've been thinking about this for a while now, and I think it's time for me
to take a break
```

Great, the generated text looks much better without the repetitions! However, we should be careful when using this technique. For instance, if we're writing about 'New York,' we wouldn't want the name to appear only once in the entire text.

Another useful feature is that we can generate multiple versions of the text and then pick the one that fits our needs the best. This is similar to beam search from before, but here we are looking at more than one of the generated outputs, instead of just picking the single best verison.

Let's see how we can do this next!

```
In [7]: def generate_multiple_sequences(
            model,
            tokenizer,
            text,
            max_length=50,
            num_beams=5,
            no_repeat_ngram_size=2,
            num_return_sequences=5,
            early_stopping=True,
        ):
            # encode input context
            input_ids = tokenizer.encode(text, return_tensors="tf")

            # generate multiple sequences using beam search with no_repeat_ngram_siz
            output = model.generate(
                input_ids,
                max_length=max_length,
                num_beams=num_beams,
                no_repeat_ngram_size=no_repeat_ngram_size,
                num_return_sequences=num_return_sequences,
                early_stopping=early_stopping,
            )

            # decode and return the text for each sequence
            return [tokenizer.decode(ids, skip_special_tokens=True) for ids in outpu


        for i, sequence in enumerate(generate_multiple_sequences(model, tokenizer, i
            print(f"{i}: {sequence}\n")
```

```
0: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able t
o walk with him again.

I've been thinking about this for a while now, and I think it's time for me
to take a break

1: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able t
o walk with him again.

I've been thinking about this for a while now, and I think it's time for me
to get back to

2: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able t
o walk with her again.

I've been thinking about this for a while now, and I think it's time for me
to take a break

3: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able t
o walk with her again.

I've been thinking about this for a while now, and I think it's time for me
to get back to

4: I enjoy walking with my cute dog, but I'm not sure if I'll ever be able t
o walk with him again.

I've been thinking about this for a while now, and I think it's time for me
to take a step
```

As you can see, the different versions of the generated text are quite similar to each other. While Beam Search is useful, it has some limitations, especially for more creative tasks like storytelling or dialogue generation. Here's why:

1. **Predictability**: Beam Search works well when we know the approximate length of the text we want to generate. But that's not always the case.
2. **Repetitiveness**: We've seen that it can sometimes get stuck and generate repetitive text. Fine-tuning to avoid this is tricky.
3. **Lack of Surprise**: Good storytelling or dialogue isn't just about picking the most likely next word. Sometimes, the most interesting choices are the unexpected ones.

So, how about we add a bit of randomness to make things more interesting? Let's try that next!

Sampling

So far, we've talked about methods that pick the next word based on how likely it is to follow the previous words. Now, let's introduce some randomness into the mix! Sampling is like rolling a weighted die to pick the next word. This makes the generated text less predictable and more interesting.

Instead of following a set path, we'll let chance guide us a bit. The result is that the generated text can take on different styles and tones, making it more dynamic.

In technical terms, we'll change a setting to `do_sample=True` to enable this random sampling. Let's see what happens when we add a sprinkle of randomness!

In [8]:
```
def generate_text_with_sampling(model, tokenizer, text, max_length=50, top_k
    # Set the seed for reproducibility
    tf.random.set_seed(0)

    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using sampling
    output = model.generate(
        input_ids, do_sample=True, max_length=max_length, top_k=top_k
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_with_sampling(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog Reyn: on the platform in a bowling alley in
Brooklyn, mint one," the partner admitted, laughing at the furred hair pounc
ed on her collar. "Really? I just sat down and watched the rainbow bear
```

Interesting, isn't it? Sampling adds a fun element of randomness, but you might notice that the text doesn't always make sense. It's like shaking a bag of words and picking them out at random—sometimes you get a coherent sentence, and sometimes you get gibberish.

So how do we make it better? Imagine the randomness as a dial that you can turn up or down. This dial is what we call 'temperature.' A lower temperature means the text will stick closer to common phrases, making it more coherent but maybe less creative.

Let's try tweaking this temperature setting and see how it impacts our generated text.

In [9]:
```
def generate_text_with_temperature(
    model, tokenizer, text, max_length=50, top_k=0, temperature=0.7
):
    # Set the seed for reproducibility
    tf.random.set_seed(0)
    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using sampling and temperature
    output = model.generate(
        input_ids,
        do_sample=True,
        max_length=max_length,
        top_k=top_k,
        temperature=temperature,
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_with_temperature(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog and I like having a nice view of the world,
including the one I'm walking through, but at the same time, I've got a litt
le bit of an anxiety about it. And I'm trying to be
```

Great, adjusting the temperature made our generated text more coherent! But be careful—if you turn down the 'temperature' too much, you'll end up with the same issues we saw with Greedy Search: repetitive and too predictable text.

In other words, finding the right balance is key to getting text that's both coherent and interesting.

Top-K Sampling

Here's another interesting way to pick the next word: Top-K Sampling. In this method, we don't consider all possible next words. Instead, we make a shortlist of the 'K' most likely next words and pick from that list. This helps us avoid weird or irrelevant words while still keeping things interesting.

Imagine you're writing a story and the sentence starts with 'The cat.' The model might consider the top 6 most likely next words to be 'sat,' 'jumped,' 'ran,' 'meowed,' 'slept,' and 'purred.' It then randomly picks one of these to continue the story.

Let's try this out with a setting of `top_k=50` to see how it works!

In [10]:
```
def generate_text_with_top_k(model, tokenizer, text, max_length=50, top_k=50
    # Set the seed for reproducibility
    tf.random.set_seed(0)

    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using top-k sampling
    output = model.generate(
        input_ids, do_sample=True, max_length=max_length, top_k=top_k
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_with_top_k(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog and I miss having to walk with my dog in th
e middle of one of the city's busiest hours at dusk. I was also inspired by
the idea of creating an interactive map of the city but I think they really
```

That turned out pretty well, didn't it? The text sounds quite human-like. However, Top-K Sampling has its quirks. For example, it uses the same 'K' most likely words for every step, which can sometimes limit creativity or include words that don't fit well.

Imagine you're writing about a 'garden.' In one step, Top-K might not consider words like 'rose' or 'tree,' which could fit well. In another step, it might include less relevant words like 'run' or 'jump.'

To address these issues, researchers came up with another method called 'Top-p' or 'nucleus' sampling. Let's explore that next!

Top-p (Nucleus) Sampling

Here's another way to add variety to our text: Top-p or 'nucleus' sampling. Instead of picking from a fixed number of top choices like in Top-K, this method gives us more flexibility. It selects words from a dynamic list that changes based on how likely each word is to appear next.

Think of it like this: If you're writing about a 'car,' words like 'drive,' 'road,' and 'speed' would be high on the list. But if you're writing about 'food,' words like 'tasty,' 'cook,' and 'eat' would be the top choices.

Let's see how this works in practice. We'll activate Top-p sampling with a specific setting and observe the results.

In [11]:
```
def generate_text_with_top_p(
    model, tokenizer, text, max_length=50, top_p=0.92, top_k=0
):
    # Set the seed for reproducibility
    tf.random.set_seed(0)

    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate text using top-p sampling
    output = model.generate(
        input_ids, do_sample=True, max_length=max_length, top_p=top_p, top_k
    )

    # decode and return the text
    return tokenizer.decode(output[0], skip_special_tokens=True)


print(generate_text_with_top_p(model, tokenizer, input_text))
```

```
I enjoy walking with my cute dog and how on Earth this actually works, altho
ugh I was reminded of one particular episode where Captain America comes to
Vancouver Island to visit Agent Coulson at his house, for $1.50 to get him t
o act kind
```

That result is quite impressive! It almost feels like it was written by a person. Both Top-K and Top-p sampling have their strengths, and you can even combine them to get the best of both worlds.

For example, using both can help us avoid odd word choices while keeping the text lively and dynamic. And guess what? If you want to generate multiple versions of the text, you can do that easily by changing a setting.

Let's explore some more options next!

In [12]:
```
def generate_multiple_sequences_with_sampling(
    model, tokenizer, text, max_length=50, top_k=50, top_p=0.95, num_return_
):
    # Set the seed for reproducibility
    tf.random.set_seed(0)

    # encode input context
    input_ids = tokenizer.encode(text, return_tensors="tf")

    # generate multiple sequences using top-k and top-p sampling
    output = model.generate(
        input_ids,
        do_sample=True,
        max_length=max_length,
        top_k=top_k,
        top_p=top_p,
        num_return_sequences=num_return_sequences,
    )

    # decode and return the text for each sequence
    return [tokenizer.decode(ids, skip_special_tokens=True) for ids in outpu


input_text = "I enjoy walking with my cute dog"
for i, sequence in enumerate(
    generate_multiple_sequences_with_sampling(model, tokenizer, input_text)
):
    print(f"{i}: {sequence}\n")
```

```
0: I enjoy walking with my cute dog and I like having a fun time, so I was g
oing to stop in the restaurant and get some coffee."

The second person I saw who didn't wear makeup at the restaurant was her ex-
boyfriend

1: I enjoy walking with my cute dog, and the time you spent with my cats was
wonderful and truly worth it. I also found it really funny that my dog gets
all upset and irritable before he goes to bed, and is even more upset when

2: I enjoy walking with my cute dog, so it has been such an inspiration to m
e in her life."

Morton's story was shared on social media.

The mother-of-two said it was about the "trem
```

So, we've explored various ways to generate text, like Top-K and Top-p sampling, which often produce smoother and more natural-sounding text than older methods like Greedy and Beam Search. However, it's important to note that no method is perfect. For instance, all methods can sometimes produce repetitive text.

Recent research suggests that the limitations of Greedy and Beam Search might be due to how the AI model is trained, rather than the method itself. Some studies even show that with the right training, Beam Search can outperform other methods.

The field of text generation is evolving quickly, and there's no one-size-fits-all approach. The good news is you can try all these different methods right here in this notebook, thanks to the transformers library.