Decoder Models and the Mechanics of Generation

Dr. Prof. Ivan Habernal
Yassine Thlija

1 What Decoder Models Actually Do

- Decoder models are not classifiers. They do not label text, extract spans, or return structured answers
- They do one thing:
  - Predict the next token, given all previous tokens
- Everything else (stories, answers, summaries, "reasoning") is a side effect of repeating this process many times
- This has consequences:
  - They are creative, not precise
  - They sound confident even when wrong
  - They will happily hallucinate if the prompt allows it

2 Two Ways to Use Decoder Models

- There are two valid ways to use decoder models (similar to what we did with encoders)
- Pipeline (easy mode):
  - Handles tokenization, generation, and decoding for you
  - Safe, concise, and hard to misuse
- Manual inference (no magic):
  - Exposes logits, probabilities, and sampling
  - Forces you to understand what the model is actually doing

3 The Pipeline Method

- The Hugging Face pipeline is a high-level wrapper
- Internally, it performs:
  - Tokenization
  - Model forward pass
  - Token generation loop
  - Decoding back to text
- Example (text generation):

```
gen_pipeline = pipeline(
    "text-generation",
    model="gpt2",
    tokenizer=tokenizer,
    device=0
)

result = gen_pipeline(
    "Once upon a time",
    max_new_tokens=40,
    do_sample=True,
    temperature=0.5
)[0]
```

- This is useful because:
  - It reduces boilerplate code
  - It handles padding and batching correctly
  - It prevents many common mistakes
- However:
  - It hides important implementation details
  - It can make generation feel "magical"

4 A Note on Padding

- GPT-style models do not define a padding token
- When batching inputs, Hugging Face assigns:
  - `pad_token_id = eos_token_id`
- You should set this explicitly:
  - To avoid warnings
  - To make model behavior explicit

```
tokenizer.pad_token = tokenizer.eos_token
model.config.pad_token_id = tokenizer.eos_token_id
```

5 Manual Text Generation (What the Pipeline Hides)

- Manual generation reveals how decoder models work internally
- The generation loop always follows the same steps:
  - Run the model
  - Extract logits for the last token
  - Convert logits to probabilities
  - Select the next token

    - Append the token and repeat
- Understanding this explains:
    - The role of temperature
    - The difference between sampling and greedy decoding
    - Why models can drift or loop

```
outputs = model(input_ids=generated_ids)
logits = outputs.logits[:, -1, :]
probs = softmax(logits / temperature)
next_token = sample(probs)
generated_ids = concat(generated_ids, next_token)
```

6 Generative Question Answering

- Decoder-based QA is generative, not extractive
- The model does not return a text span from the context
- Instead, it:
    - Reads the context
    - Interprets the question
    - Generates an answer token by token
- Prompt structure matters:
    - Instruction-tuned models expect explicit formats
    - Example: {`question: ..., context: ...`}

```
prompt = f"question: {question} context: {context}"
```

```
qa_pipeline = pipeline("text2text-generation", model="google/flan-t5-
    small")
result = qa_pipeline(prompt)[0]["generated_text"]
```

7 Manual Generative QA

- Manual generation uses `model.generate()`
- This still abstracts the token-by-token loop, but gives you control over:
    - decoding strategy
    - maximum generation length
    - determinism vs randomness
- Typical usage:
    - Tokenize the prompt
    - Call `model.generate(...)`

- Decode the output tokens back to text

```
output_ids = model.generate(
    input_ids,
    max_new_tokens=32,
    do_sample=False
)
answer = tokenizer.decode(output_ids[0], skip_special_tokens=True)
```

- Greedy decoding (`do_sample=False`) is useful when:
  - you want factual answers
  - reproducibility matters
  - you are debugging model behavior

8 Summarization

- Summarization is a fully generative task
- The model:
  - reads the entire input text
  - compresses the information
  - rewrites it in its own words
- A pipeline is usually sufficient:

```
summ_pipeline = pipeline("summarization", model="facebook/bart-large-cnn
   ")
result = summ_pipeline(text)[0]["summary_text"]
```

- Important caveat:
  - summaries may introduce information not explicitly stated
  - this is a property of generative models
  - it is not a bug