1. Introduction and Goal:
The objective was to configure a Jetson device running JetPack 4.6.1 (L4T R32.6.1 / Ubuntu 18.04) to execute modern machine learning models, particularly Large Language Models (LLMs). This required addressing the inherent constraints of the target hardware (likely Jetson Nano/TX1), the older JetPack software stack (Ubuntu 18.04, CUDA 10.2), and the complexities of installing the necessary Python ML ecosystem. An initial feasibility study involving TensorRT optimization was also conducted in a separate Colab environment.

2. Initial Exploration: TensorRT Optimization in Colab (Challenges Encountered):
Optimizing a BLOOM model (bloom-560m) using TensorRT in Colab revealed significant challenges applicable to resource-constrained deployment. While 8-bit quantization with bitsandbytes reduced the inference footprint in PyTorch, attempts to use optimum for TensorRT integration faced hurdles: evolving library APIs (optimum.tensorrt vs optimum.onnxruntime), complex installation dependencies (tensorrt, onnx, onnxruntime-gpu), and severe resource limitations (CPU RAM Out-Of-Memory) during the ONNX export and TensorRT engine build phases. This indicated that generating optimized TensorRT engines, especially for INT8 precision, demands substantial computational resources exceeding those available on the target Jetson with 2GB Memory.

3. Jetson Environment Setup: Core Compatibility and Dependency Issues:
Setting up the Python environment on the Jetson presented following challenges:

- **PyTorch/TorchVision Installation:** The primary obstacle was installing GPU-accelerated PyTorch and TorchVision. Standard PyPI repositories lacked compatible pre-compiled wheels for Jetson's aarch64 architecture and the specific CUDA/cuDNN versions in JetPack 4.6.1. Success mandated locating and installing official NVIDIA-provided wheels (.whl files) specifically built for JP4.6.1 and the chosen Python version (ultimately Python 3.6). This involved navigating NVIDIA's distribution channels, handling outdated links, and ensuring the correct wheel file (identified by tags like cp36) was installed locally. Attempts to use wheels built for different Python versions or downloaded from unofficial sources consistently failed validation or led to runtime errors. PyTorch 1.10 (cp36) and TorchVision 0.11 (cp36) were identified as the compatible versions and successfully built from source.
- **Build Toolchain and Dependencies:** Installing required libraries often necessitated building them from source, uncovering missing build dependencies:

    Numpy required iterative installation of specific Cython (<3.0) and setuptools versions, plus the packaging library, to overcome build failures related to compiler requirements and internal setuptools/distutils incompatibilities, before a suitable pre-built wheel was eventually found and installed (numpy==1.21.6).
    Tokenizers (Transformers Dependency) required installing the Rust toolchain

(curl, rustup, cargo) and the setuptools-rust Python package to allow compilation of the Rust-based library.

- **System Library Conflicts (MPI):** A critical incompatibility arose when attempting to use an unofficial PyTorch 1.11 wheel built for Python 3.8. It failed at runtime due to a missing dependency (OSError: libmpi_cxx.so.40 not found), indicating it required OpenMPI 4.x, whereas JetPack 4.6.1 provides OpenMPI 2.1.1. This could not be resolved by installing the system's libopenmpi-dev package and strongly reinforced the necessity of using PyTorch wheels built specifically against the target JetPack's library set. This incompatibility was a key factor in standardizing on the Python 3.6 environment for which official, compatible wheels (like PyTorch 1.10) were available.

4. Challenges with Advanced Model Deployment (BLOOM ONNX Example):
Attempting to run a more complex deployment scenario using a BLOOM ONNX model highlighted further runtime and model-specific issues pertinent to the Jetson platform:

- **Tokenizer Loading Mechanism:** Differences between the tokenizers library (Tokenizer.from_pretrained, lacks Hub download) and the transformers library (AutoTokenizer, supports Hub download) required careful selection of the loading method based on whether the tokenizer files were local or needed downloading.
- **Library Version Skew:** The feasible version of the transformers library for the older Python 3.6 environment potentially lacked full compatibility with the newest BLOOM tokenizer configurations, presenting a risk that required careful version management or acceptance of using slightly older model configurations.
- **ONNX Runtime & Execution Providers:** Successful execution of ONNX models relies on correctly installed and configured backends like onnxruntime-gpu and its execution providers (CUDA, TensorRT). Issues arose if TensorRT was improperly set up or if the model contained unsupported operations for the Jetson's TRT version, causing failures or silent fallback to less efficient providers.
- **ONNX IR Version Incompatibility:** The BLOOM ONNX model used a newer ONNX Intermediate Representation (IR) version than supported by the available ONNX Runtime for JetPack 4.6.1. This necessitated a manual IR downgrade using the onnx version_converter before the model could be loaded.
- **Local File Management:** Running inference from local files demanded not only the ONNX model but also the complete set of corresponding tokenizer configuration files (tokenizer.json, vocabularies, merges) to be present in the correct structure.
- **Autoregressive Generation Logic:** Correctly generating text with stateful models like BLOOM requires handling past_key_values explicitly within the generation loop, rather than just performing simple token prediction based on input_ids and logits. Failing to do so results in incoherent output and potential runtime errors.

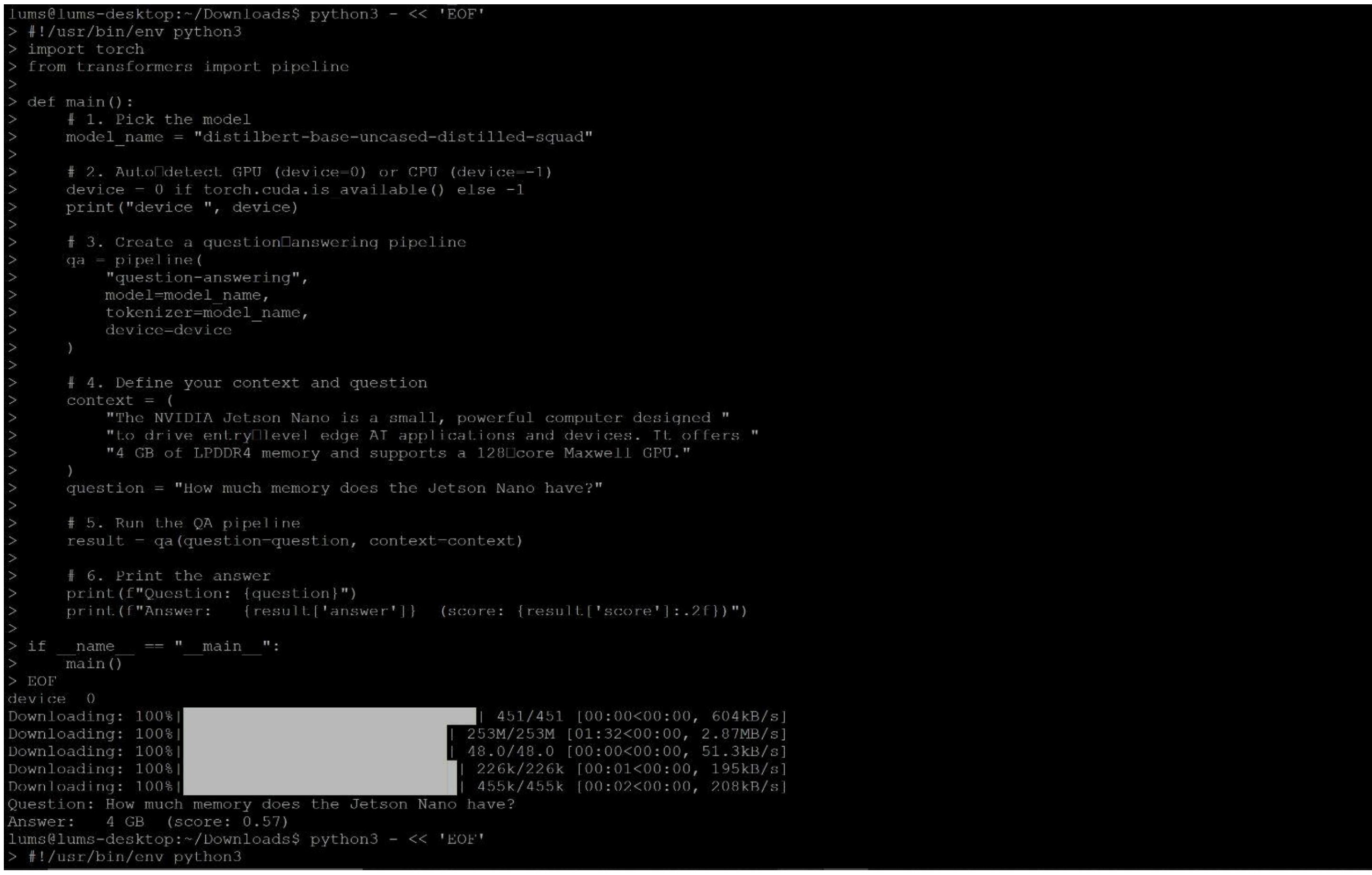
5. Final Working State and LLM Execution:
Ultimately, a stable Python 3.6 environment was established on the Jetson JP4.6.1. PyTorch 1.10 and TorchVision 0.11 (with CUDA support via NVIDIA wheels) were installed, along with necessary dependencies like NumPy, Transformers, Optimum, ONNX tools, and Pillow. Basic inference using small generative models (distilgpt2, gpt2) was demonstrated successfully via the Hugging Face transformers pipeline API. For improved efficiency on the resource-limited hardware, alternative approaches using ctransformers (requiring careful dependency versioning, e.g., huggingface-hub<0.20) or direct llama.cpp execution with quantized GGUF models may help to improve performance.

6. Key Takeaways:

- Software versions (PyTorch, TorchVision, CUDA, cuDNN, system libraries) must align with the specific JetPack release. Using official NVIDIA-provided wheels is crucial for GPU acceleration and stability.
- Installing libraries often involves building from source, necessitating the manual installation and version management of build tools (compilers, Cython, Rust, setuptools, packaging).
- Python wheels may have implicit dependencies on specific versions of system libraries (like OpenMPI) that must be met by the host OS environment. Mismatches lead to runtime errors.
- Deployment via ONNX requires compatible runtimes, attention to IR versions, support for model operations by the target execution provider (especially TensorRT on Jetson), and careful management of associated files (tokenizers). TensorRT engine building itself is highly resource-intensive.
- Successful inference depends on correct tokenizer handling and appropriate generation logic (e.g., managing state for autoregressive models). Model size and quantization are critical for performance on constrained hardware. CPU-centric inference engines (llama.cpp, ctransformers) often offer a practical path for smaller models on older Jetsons.

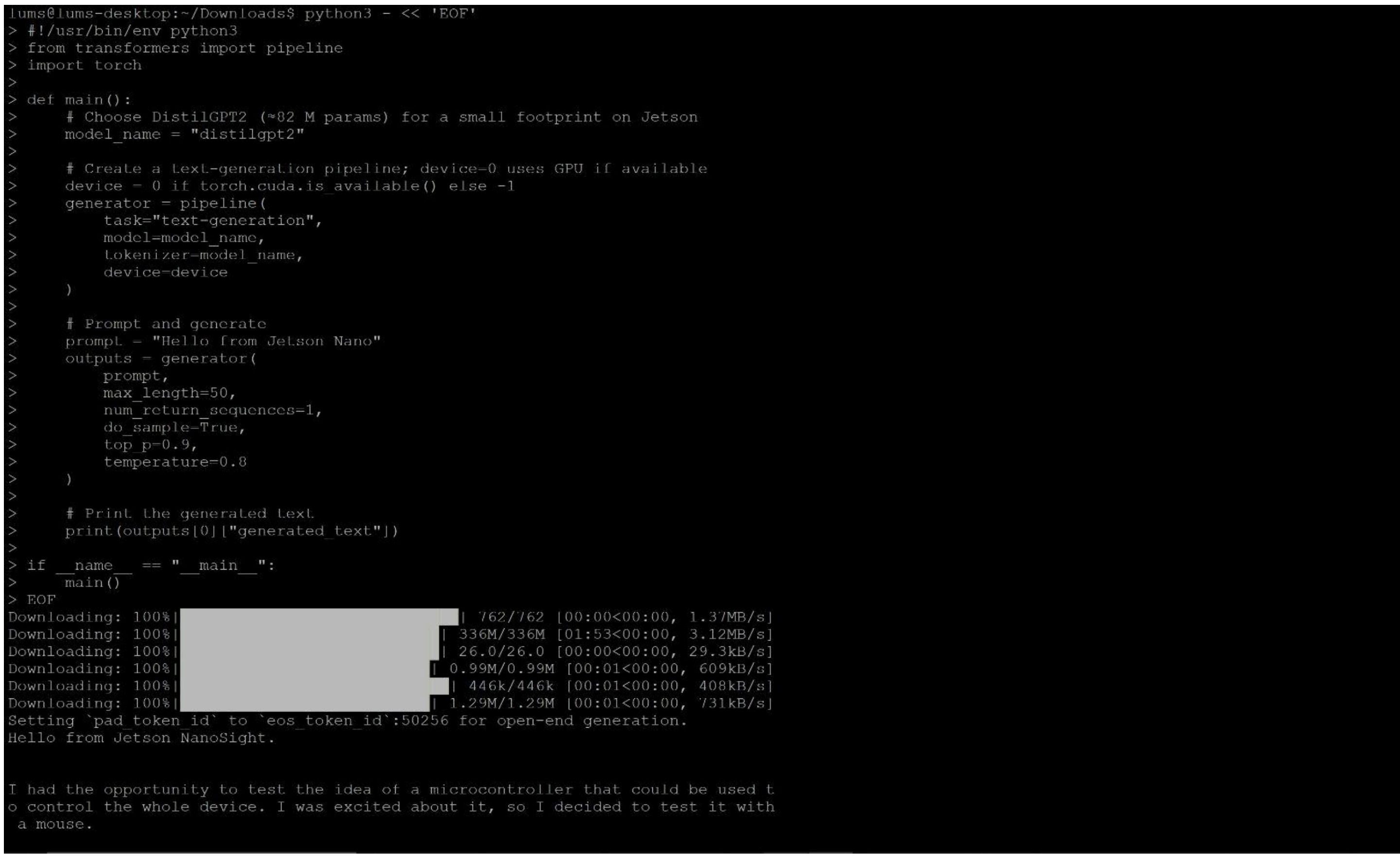
SCREENSHOTS OF MODEL INFERENCES ON JETSON BELOW

Distilled-Squad



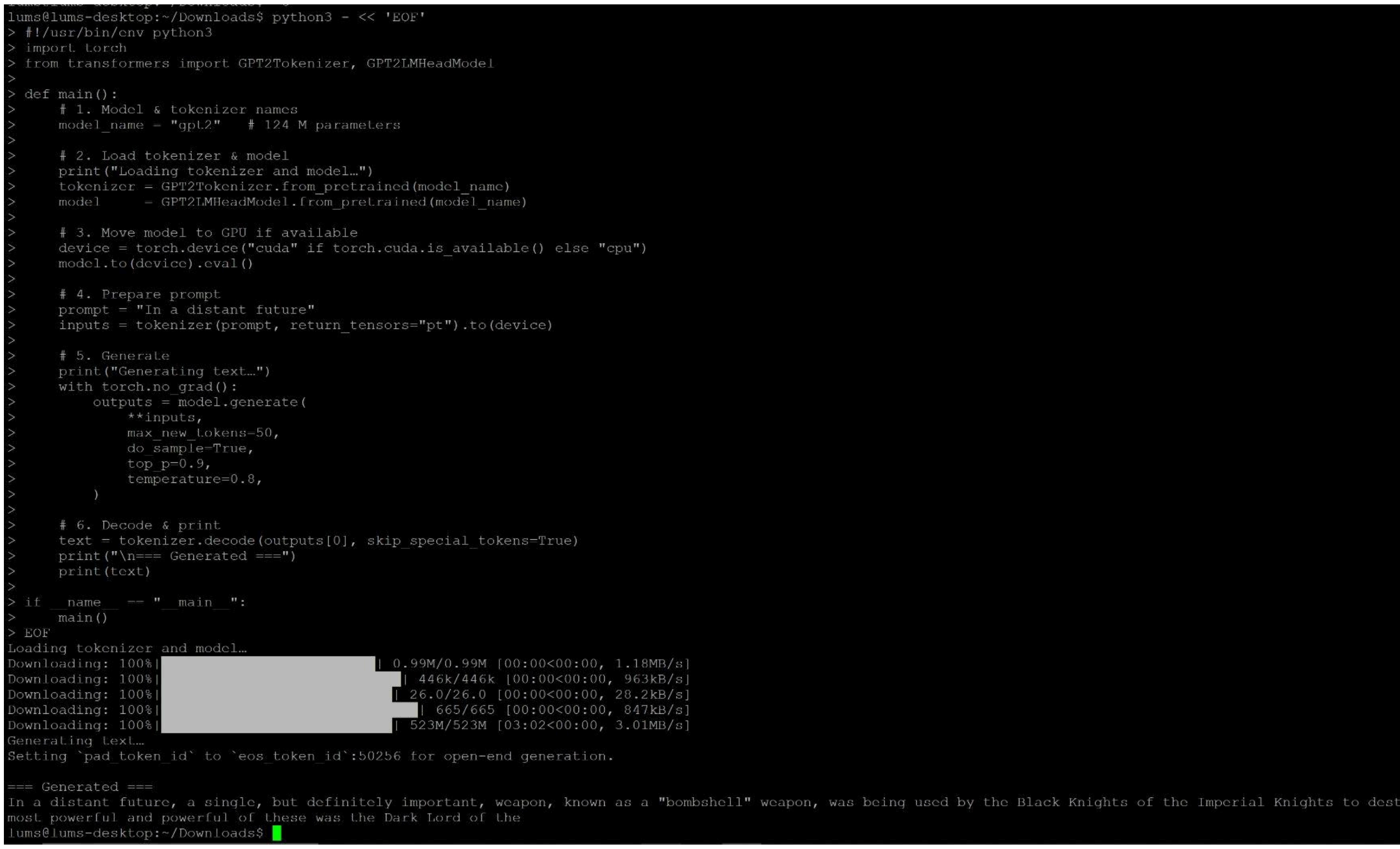
```
lums@lums-desktop:~/Downloads$ python3 - << 'EOF'
> #!/usr/bin/env python3
> import torch
> from transformers import pipeline
>
> def main():
>     # 1. Pick the model
>     model_name = "distilbert-base-uncased-distilled-squad"
>
>     # 2. Auto□detect GPU (device=0) or CPU (device=-1)
>     device = 0 if torch.cuda.is_available() else -1
>     print("device ", device)
>
>     # 3. Create a question□answering pipeline
>     qa = pipeline(
>         "question-answering",
>         model=model_name,
>         tokenizer=model_name,
>         device=device
>     )
>
>     # 4. Define your context and question
>     context = (
>         "The NVIDIA Jetson Nano is a small, powerful computer designed "
>         "to drive entry□level edge AI applications and devices. It offers "
>         "4 GB of LPDDR4 memory and supports a 128□core Maxwell GPU."
>     )
>     question = "How much memory does the Jetson Nano have?"
>
>     # 5. Run the QA pipeline
>     result = qa(question=question, context=context)
>
>     # 6. Print the answer
>     print(f"Question: {question}")
>     print(f"Answer:   {result['answer']}  (score: {result['score']:.2f})")
>
> if __name__ == "__main__":
>     main()
> EOF
device  0
Downloading: 100%|██████████████████████████████| 451/451 [00:00<00:00, 604kB/s]
Downloading: 100%|███████████████████████████| 253M/253M [01:32<00:00, 2.87MB/s]
Downloading: 100%|███████████████████████████| 48.0/48.0 [00:00<00:00, 51.3kB/s]
Downloading: 100%|█████████████████████████████| 226k/226k [00:01<00:00, 195kB/s]
Downloading: 100%|█████████████████████████████| 455k/455k [00:02<00:00, 208kB/s]
Question: How much memory does the Jetson Nano have?
Answer:   4 GB  (score: 0.57)
lums@lums-desktop:~/Downloads$ python3 - << 'EOF'
> #!/usr/bin/env python3
```

Distilled-GPT2



```
lums@lums-desktop:~/Downloads$ python3 - << 'EOF'
> #!/usr/bin/env python3
> from transformers import pipeline
> import torch
>
> def main():
>     # Choose DistilGPT2 (≈82 M params) for a small footprint on Jetson
>     model_name = "distilgpt2"
>
>     # Create a text-generation pipeline; device=0 uses GPU if available
>     device = 0 if torch.cuda.is_available() else -1
>     generator = pipeline(
>         task="text-generation",
>         model=model_name,
>         tokenizer=model_name,
>         device=device
>     )
>
>     # Prompt and generate
>     prompt = "Hello from Jetson Nano"
>     outputs = generator(
>         prompt,
>         max_length=50,
>         num_return_sequences=1,
>         do_sample=True,
>         top_p=0.9,
>         temperature=0.8
>     )
>
>     # Print the generated text
>     print(outputs[0]["generated_text"])
>
> if __name__ == "__main__":
>     main()
> EOF
Downloading: 100%|██████████████████████████████| 762/762 [00:00<00:00, 1.37MB/s]
Downloading: 100%|████████████████████████████| 336M/336M [01:53<00:00, 3.12MB/s]
Downloading: 100%|████████████████████████████| 26.0/26.0 [00:00<00:00, 29.3kB/s]
Downloading: 100%|███████████████████████████| 0.99M/0.99M [00:01<00:00, 609kB/s]
Downloading: 100%|█████████████████████████████| 446k/446k [00:01<00:00, 408kB/s]
Downloading: 100%|███████████████████████████| 1.29M/1.29M [00:01<00:00, 731kB/s]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
Hello from Jetson NanoSight.


I had the opportunity to test the idea of a microcontroller that could be used t
o control the whole device. I was excited about it, so I decided to test it with
 a mouse.
```



```
lums@lums-desktop:~/Downloads$ python3 - << 'EOF'
> #!/usr/bin/env python3
> import torch
> from transformers import GPT2Tokenizer, GPT2LMHeadModel
>
> def main():
>     # 1. Model & tokenizer names
>     model_name = "gpt2"   # 124 M parameters
>
>     # 2. Load tokenizer & model
>     print("Loading tokenizer and model…")
>     tokenizer = GPT2Tokenizer.from_pretrained(model_name)
>     model     = GPT2LMHeadModel.from_pretrained(model_name)
>
>     # 3. Move model to GPU if available
>     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
>     model.to(device).eval()
>
>     # 4. Prepare prompt
>     prompt = "In a distant future"
>     inputs = tokenizer(prompt, return_tensors="pt").to(device)
>
>     # 5. Generate
>     print("Generating text…")
>     with torch.no_grad():
>         outputs = model.generate(
>             **inputs,
>             max_new_tokens=50,
>             do_sample=True,
>             top_p=0.9,
>             temperature=0.8,
>         )
>
>     # 6. Decode & print
>     text = tokenizer.decode(outputs[0], skip_special_tokens=True)
>     print("\n=== Generated ===")
>     print(text)
>
> if __name__ == "__main__":
>     main()
> EOF
Loading tokenizer and model…
Downloading: 100%|█████████████████████████| 0.99M/0.99M [00:00<00:00, 1.18MB/s]
Downloading: 100%|████████████████████████████| 446k/446k [00:00<00:00, 963kB/s]
Downloading: 100%|███████████████████████████| 26.0/26.0 [00:00<00:00, 28.2kB/s]
Downloading: 100%|██████████████████████████████| 665/665 [00:00<00:00, 847kB/s]
Downloading: 100%|███████████████████████████| 523M/523M [03:02<00:00, 3.01MB/s]
Generating text…
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.

=== Generated ===
In a distant future, a single, but definitely important, weapon, known as a "bombshell" weapon, was being used by the Black Knights of the Imperial Knights to dest
most powerful and powerful of these was the Dark Lord of the
lums@lums-desktop:~/Downloads$ 
```

GPT2