

**Department of Electrical and Electronics Engineering**

**EEE 443: Neural Networks**

*Class Project VIII Report*

**Name and Surname:** Ali Aral Takak

**Student ID:** 22001758

**Department:** EEE

**Instructor:** Erdem Koyuncu

**Introduction**

The eighth project of the course focuses on experimenting with different aspects of large language models (LLMs), using nanoGPT; a minimal implementation of GPT-2 large language model. In this project, we will explore the architecture of the GPT-2, and by tuning its parameters, observe the effects of parameters adjustment and fine-tuning.

**Assignment Questions**

**Q2)** Run the code given in the assignment file.

The figure provided below displays the output of the sample code given in the instruction file:

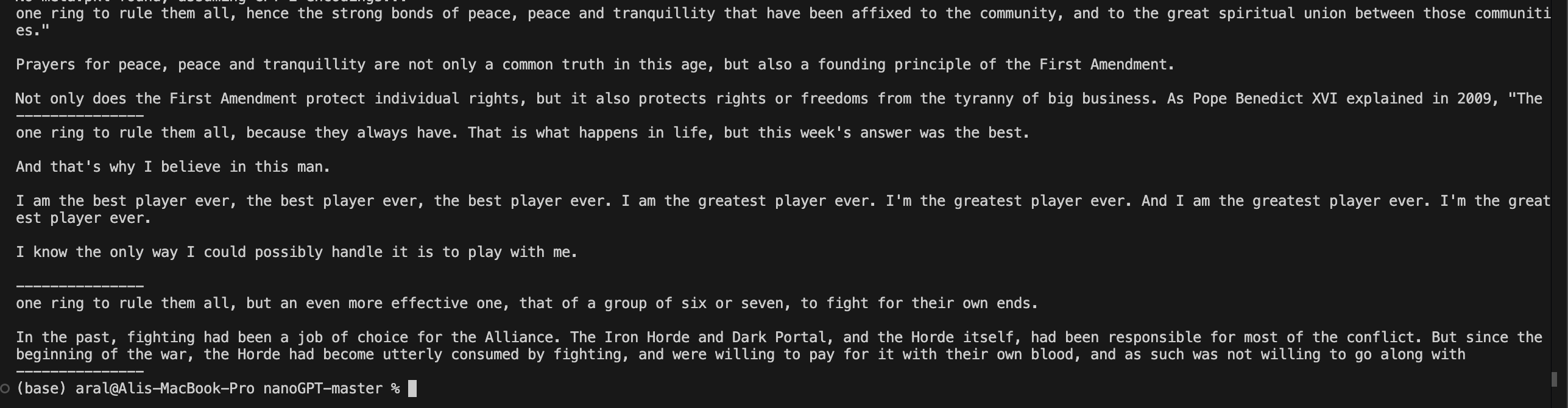


Figure 1: Output of the code given in the instructions.

Hence, we infer that the GPT setup is handled properly and works as intended.

**Q3)** Briefly explain how the model generates new words in terms of model architecture, number of layers, activation functions etc. You may assume the reader knows what a multihead attention block is, so you do not have to explain the very basics.

GPT2 generates text in an autoregressive fashion by repeatedly feeding its own outputs back into the model. At each step, the sequence of tokens generated up to that point is first converted into continuous representation via learned token and positional embeddings. These embedding then pass through a stack **of 12 identical Transformer blocks, each comprising a causal multi-head self-attention layer with 12 heads and residual connections, followed by a position-wise feedforward network of inner dimension of 3072 layers that uses Gaussian Error Linear Unit (GELU) activation [1].** The final hidden-state corresponding to the most recent token is linearly projected to a 50 257-dimensional vector of logits, which are normalized via softmax to yield a probability distribution over the next token in the vocabulary. Hence, a token is sampled from this distribution, appended to the sequence, and the process repeats until generation completes.

**Q4)** Briefly explain the purpose of “start", “num samples" and “max\_new\_tokens" parameters.

The three parameters given in the question control how generation begins, how many outputs are produced, and how long each continuation can grow. The “start” argument defines the initial text prompt that directs the model the context of generation. The “num\_samples” parameter determines how many independent continuations from the script will be produced, allowing comparison of multiple outputs from the same prompt. The “max\_new\_tokens” parameter sets a cap on the number of tokens added beyond the prompt in order to ensure that generation stops once this limit is reached.

**Q5)** Choose a prompt of your choice, change the temperature to 0.1, and include the outputs of your results in your report. What is the purpose of the temperature? Why do the outputs behave the way they do?

For this question, we chose the prompt “A long time ago in a galaxy far, far away...”. The results of the prompt can be observed in the figure given below, with temperature parameter set to 0.1:

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 2:** Text generation results with temperature = 0.1.

The temperature parameter regulates the model’s next token distribution by dividing the raw logits before the softmax application. If are the unnormalized log-probabilities for each token , then temperature rescales them as:

When , the distribution becomes more amplified, amplifying the highest-scoring log-probabilities and suppressing lower-scoring ones; on the other hand, flattens the distribution, making lower probability tokens relatively more likely.

**Q6)** Choose a prompt of your choice, change the temperature to 10, and include the outputs of your results in your report. Why do the outputs behave the way they do?

The figure given below displays the results when the temperature is set to 10, with the same prompt in the question 5:

A screenshot of a computer screen

AI-generated content may be incorrect.

**Figure 3:** Text generation results with temperature = 10.

The temperature parameter regulates the model’s next token distribution by dividing the raw logits before the softmax application. If are the unnormalized log-probabilities for each token , then temperature rescales them as:

When , the distribution becomes more amplified, amplifying the highest-scoring log-probabilities and suppressing lower-scoring ones; on the other hand, flattens the distribution, making lower probability tokens relatively more likely.

**Q7)** Keep the temperature a 10 but change top\_k to 2. Describe the reasons for the change in outputs as compared with the previous subquestion.

The figure given below displays the results of text generation when the temperature is set to 10 and top\_k parameters is reduced to 2 from 200:

A screenshot of a computer

AI-generated content may be incorrect.

**Figure 4:** Text generation results with temperature = 10 and top\_k = 2.

Top-k sampling is a decoding strategy, in which at each generation step, the model first identifies the k tokens with the highest predicted probabilities, discards other options, and then renormalizes the remaining probabilities to sum to one.

In our case, the randomness of the generation comes from the high temperature value. Also, since we have set the top\_k parameter to 2, the model can only alternate between the two high-scoring options rather than exploring the full vocabulary.

**Q8)** Restore the original settings of the parameters. Inspect the token embeddings used by the language model.

**Q8.a)** Locate the token embedding matrix wte in the model. This is a matrix of size (vocab\_size, embedding\_dim).

The token embedding matrix wte can be found in model definition. The figure given below displays the code block:

A computer screen shot of a program code

AI-generated content may be incorrect.

**Figure 5:** Token embedding matrix definition in GPT class.

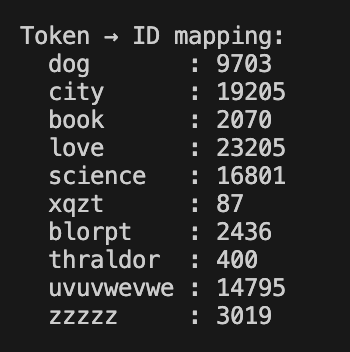
**Q8.b)** Pick five common English words and five rare or made-up tokens.

As a ground reference, we will use the words given in the instruction file.

* **Common words:** Dog, City, Book, Love, Science
* **Rare or made-up words:** Xqzt, Blorpt, Thraldor, Uvuvwevwe, Zzzzz

**Q8.c)** Use the tokenizer to obtain their token IDs. For each token, retrieve its embedding vector.

The figure given below displays the token IDs for every word determined and used:



**Figure 6:** Token - ID mapping for every word used.

The figure given below displays the embedding vectors for every word determined and used:

A black screen with white text

AI-generated content may be incorrect.

**Figure 7:** Embedding vectors of words.

**Q8.d)** Compute and compare:

* The norms of each embedding vector.

The figure given below displays the norms of each embedding vector:

A graph of blue bars with white text

AI-generated content may be incorrect.

**Figure 8:** Token embedding L2 norms.

* The pairwise cosine similarities among the five common tokens.

The figure given below displays the cosine similarities among the five common tokens:

A yellow and purple squares

AI-generated content may be incorrect.

**Figure 9:** Cosine similarity of common tokens.

* The pairwise cosine similarities among the five rare tokens.

The figure given below displays the cosine similarities among the five rare or made-up tokens:

A yellow and blue squares

AI-generated content may be incorrect.

**Figure 10:** Cosine similarity of rare or made-up tokens.

**Q8.e)** What differences do you observe between common and rare tokens in terms of norm and similarity? What do you think this reveals about how the model allocates capacity in its embedding space?

Common tokens like “dog,” “city,” “book,” “love,” and “science” have higher L₂ norms (around 3.9–4.2) than most made-up tokens. This means the model gives more space in its embeddings to words it sees often. When we look at cosine similarities, common words spread out moderately (similarities around 0.29–0.37), so they’re distinct but still related. Rare tokens, however, cluster more tightly without clear patterns. Thus, GPT2 dedicates richer, more varied embeddings to frequent words and compresses rare or unknown strings into a smaller, less detailed part of its embedding space.

**Q9)** Experiment with layer pruning as discussed below.

**Q9.a)** Modify the inference-time loop of the GPT model so that only every other transformer block is applied. That is, instead of applying all n layers, apply only layers 0, 2, 4, … (even numbered layers).

See **Appendix B** for the script.

**Q9.b)** Run the model with this reduced-depth inference on the same prompt. Compare the output to the original model output (without pruning).

The figure given below displays the text generation results of unmodified GPT2 and even number layer pruning:

A screenshot of a computer screen

AI-generated content may be incorrect.

Figure 11: Text generation results of various layer modifications on GPT2.

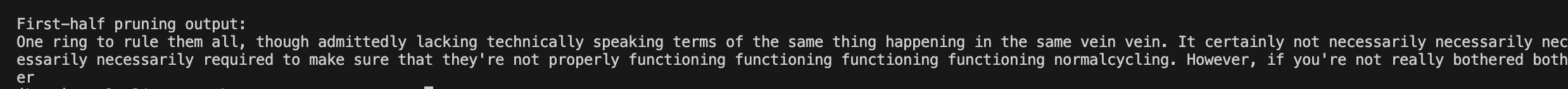
When running the pruned model on the same prompt, the generated text diverges markedly from the full-depth output. The standard GPT2 continuation builds a short narrative with new clauses and sensible progression. On the contrary, the even-layer-pruned model produces a single looping phrase, repeating it dozens of times instead of developing a coherent story or adding fresh content.

**Q9.c)** Discuss how the output changes. Is it shorter? More repetitive? Less coherent? What does this tell you about how model depth affects fluency and semantics?

From this comparison, it is clear that skipping every other layer led to highly repetitive, low-coherence text. Although the pruned model can still generate a similar length of tokens, it lacks the depth needed to refine context and suppress repetition. This shows that each additional transformer block contributes crucial non-linear transformations that prevent looping and support the gradual buildup of narrative structure.

**Q9.d)** Repeat the above, removing the last n/2 layers. Comment on the differences.

The figure given below displays the text generation results of the last n/2 layers:



**Figure 12:** First half pruning output.

Removing the last half of the layers yields only slightly better initial fluency: the text begins with grammatically plausible phrases but soon descends into word-level repetition. This indicates that early layers capture basic syntax and local dependencies, while deeper layers are responsible for maintaining long-range coherence, semantic richness, and varied expression.

**Conclusion**

In summary, our experiments showed how GPT2’s depth and settings shape its output. We saw that token embeddings travel through twelve GELU-activated layers to predict each next word, and that start, num\_samples, and max\_new\_tokens control the prompt and length of generation. Changing temperature and top-k revealed how randomness and candidate selection affect diversity. Examining embeddings confirmed that common words get larger, more distinct vectors than rare ones. Finally, pruning layers—either skipping every other block or cutting the last half—made the text more repetitive and less coherent, demonstrating that full model depth is key to meaningful generation.

**References**

**[1]** A. Radford et al., Language models are unsupervised multitask learners, https://cdn.openai.com/better-language-models/language\_models\_are\_unsupervised\_multitask\_learners.pdf (accessed May 18, 2025).

**Appendices**

**Appendix A:** Python Script for Question 8

# Import required libraries.

import torch

import torch.nn.functional as F

import matplotlib.pyplot as plt

import numpy as np

from transformers import AutoModelForCausalLM, AutoTokenizer

# Compute pairwise cosine similarity matrix.

def pairwiseCosine(mat: torch.Tensor) -> torch.Tensor:

matNorm = F.normalize(mat, dim=1)

return matNorm @ matNorm.T

# Retrieve first sub-token ID for each token in list.

def getFirstId(tokenizer: AutoTokenizer, tokenList: list[str]) -> list[int]:

encoded = tokenizer(tokenList, add\_special\_tokens=False).input\_ids

return [ids[0] for ids in encoded]

# Plot L2 norms as a bar chart.

def plotL2Norms(tokens: list[str], norms: torch.Tensor) -> None:

x = np.arange(len(tokens))

norms\_np = norms.detach().numpy()

plt.figure()

plt.bar(x, norms\_np)

plt.xticks(x, tokens, rotation=45, ha='right')

plt.ylabel("L2 Norm")

plt.title("Token Embedding L2 Norms")

plt.tight\_layout()

# Plot cosine similarity matrix as a heatmap.

def plotCosineMatrix(tokens: list[str], sims: torch.Tensor, title: str) -> None:

sims\_np = sims.detach().numpy()

plt.figure()

plt.imshow(sims\_np, interpolation='nearest')

plt.colorbar()

n = len(tokens)

plt.xticks(np.arange(n), tokens, rotation=45, ha='right')

plt.yticks(np.arange(n), tokens)

plt.title(title)

plt.tight\_layout()

# Load GPT-2 model and tokenizer.

modelName = "gpt2"

print(f"Loading {modelName} model and tokenizer.")

tokenizer = AutoTokenizer.from\_pretrained(modelName)

model = AutoModelForCausalLM.from\_pretrained(modelName)

model.eval()

# Extract token embedding matrix.

wte = model.transformer.wte.weight

# Define tokens.

commonTokens = ["dog", "city", "book", "love", "science"]

rareTokens = ["xqzt", "blorpt", "thraldor", "uvuvwevwe", "zzzzz"]

# Tokenize and get first sub-token IDs.

commonIds = getFirstId(tokenizer, commonTokens)

rareIds = getFirstId(tokenizer, rareTokens)

# Print token to ID mapping.

print("\nToken → ID mapping:")

for tok, idx in zip(commonTokens + rareTokens, commonIds + rareIds):

print(f" {tok:10s}: {idx}")

# Lookup embeddings.

commonEmbs = wte[commonIds]

rareEmbs = wte[rareIds]

# Print embedding vectors for each token.

print("\nEmbedding vectors (first 10 elements):")

for tok, emb in zip(commonTokens + rareTokens, torch.cat([commonEmbs, rareEmbs], dim=0)):

emb\_np = emb.detach().numpy()

print(f" {tok:10s}: {emb\_np[:10].tolist()}")

# Compute L2 norms.

commonNorms = torch.norm(commonEmbs, dim=1)

rareNorms = torch.norm(rareEmbs, dim=1)

# Print L2 norms.

print("\nL2 norms:")

for tok, norm in zip(commonTokens, commonNorms):

print(f" {tok:10s}: {norm:.4f}")

for tok, norm in zip(rareTokens, rareNorms):

print(f" {tok:10s}: {norm:.4f}")

# Compute pairwise cosine similarities.

commonSims = pairwiseCosine(commonEmbs)

rareSims = pairwiseCosine(rareEmbs)

# Generate and display plots.

allTokens = commonTokens + rareTokens

allNorms = torch.cat([commonNorms, rareNorms])

plotL2Norms(allTokens, allNorms)

plotCosineMatrix(commonTokens, commonSims, "Cosine Similarity of Common Tokens")

plotCosineMatrix(rareTokens, rareSims, "Cosine Similarity of Rare and Made-Up Tokens")

plt.show()

**Appendix B:** Python Script for Question 9

# Import required libraries.

import torch

from transformers import AutoModelForCausalLM, AutoTokenizer

# Set device.

device = torch.device("cpu")

# Load GPT-2 model and tokenizer.

modelName = "gpt2"

print(f"Loading {modelName} model and tokenizer.")

tokenizer = AutoTokenizer.from\_pretrained(modelName)

model = AutoModelForCausalLM.from\_pretrained(modelName).to(device)

model.eval()

# Get combined token and position embeddings.

def getEmbeddings(inputIds: torch.Tensor) -> torch.Tensor:

# Compute token embeddings and add position embeddings.

tokenEmb = model.transformer.wte(inputIds)

posEmb = model.transformer.wpe(torch.arange(inputIds.size(-1), device=device))

return tokenEmb + posEmb

# Forward pass using only even-indexed transformer layers.

def forwardPrunedEven(inputIds: torch.Tensor) -> torch.Tensor:

# Apply every other layer starting from layer 0.

x = getEmbeddings(inputIds)

for i, block in enumerate(model.transformer.h):

if i % 2 == 0:

x = block(x)[0]

x = model.transformer.ln\_f(x)

return model.lm\_head(x)

# Forward pass using only the first half of transformer layers.

def forwardPrunedFirstHalf(inputIds: torch.Tensor) -> torch.Tensor:

# Apply only the first n/2 layers.

x = getEmbeddings(inputIds)

half = len(model.transformer.h) // 2

for i, block in enumerate(model.transformer.h):

if i < half:

x = block(x)[0]

x = model.transformer.ln\_f(x)

return model.lm\_head(x)

# Generate text given a custom forward function.

def generateWithForward(prompt: str, forwardFn, maxNewTokens: int = 200) -> str:

# Tokenize prompt and iteratively sample next tokens.

inputIds = tokenizer(prompt, return\_tensors="pt").input\_ids.to(device)

for \_ in range(maxNewTokens):

logits = forwardFn(inputIds)

nextToken = torch.argmax(logits[:, -1, :], dim=-1, keepdim=True)

inputIds = torch.cat([inputIds, nextToken], dim=-1)

return tokenizer.decode(inputIds[0])

# Define the prompt.

prompt = "One ring to rule them all,"

# Generate and print full-depth output.

print("\nFull-depth GPT-2 output:")

fullIds = tokenizer(prompt, return\_tensors="pt").input\_ids.to(device)

fullOut = model.generate(fullIds, max\_new\_tokens=50, do\_sample=False)

print(tokenizer.decode(fullOut[0]))

# Generate and print even-layer pruned output.

print("\nEven-layer pruning output:")

print(generateWithForward(prompt, forwardPrunedEven, maxNewTokens=50))

# Generate and print first-half pruned output.

print("\nFirst-half pruning output:")

print(generateWithForward(prompt, forwardPrunedFirstHalf, maxNewTokens=50))