STATS-413-HW-5

Author: Hochan Son

UID: 206547205

Date: @November 26, 2025

Problem 1

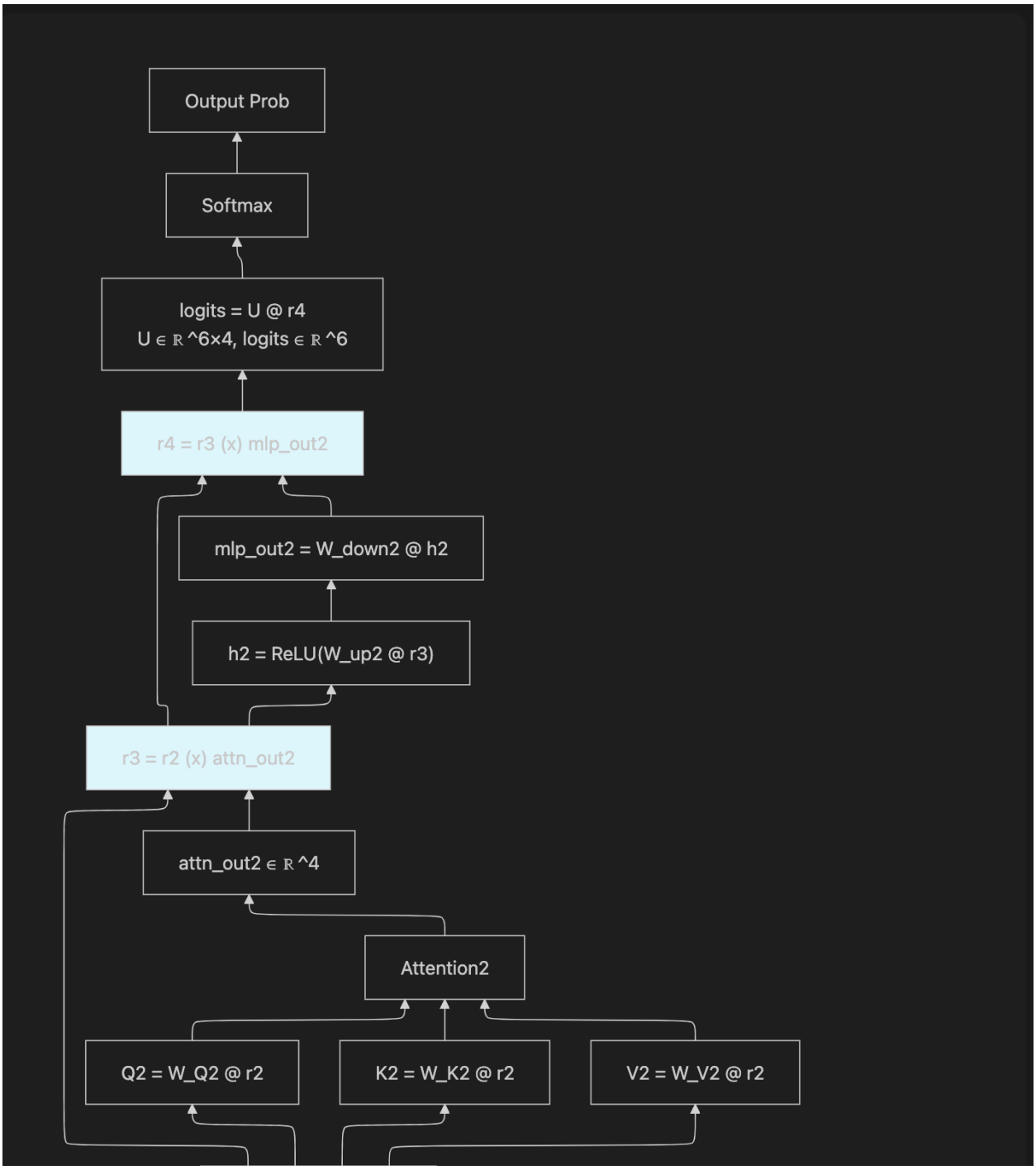
- Core Architecture and Information Flow:

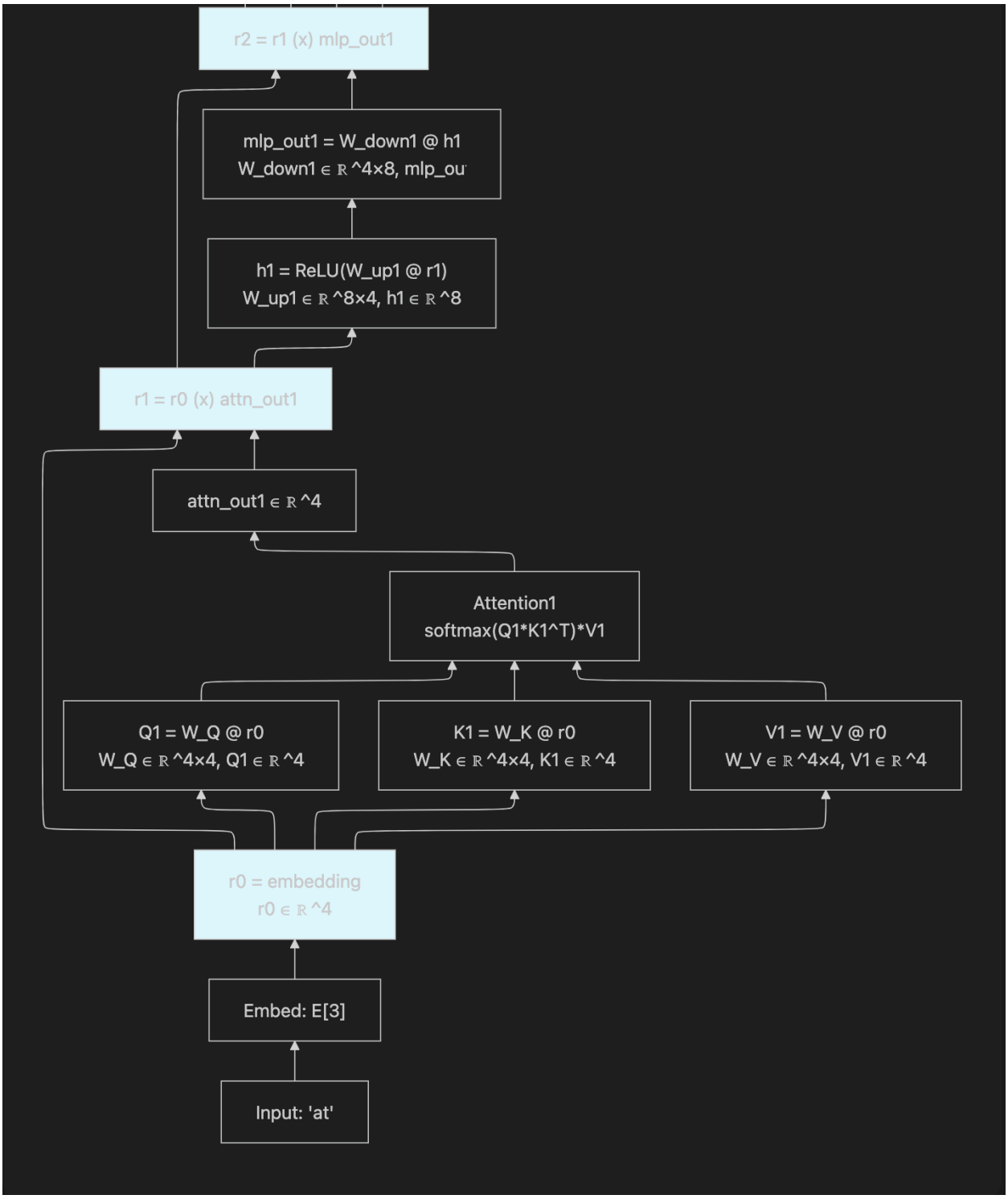
This illustrates how a GPT model processes a single token ("at" in "UCLA is at Westwood") through alternating layers of context retrieval (attention) and memory retrieval (MLP), with a residual stream as the central highway for accumulating information. The residual stream maintains an 8-dimensional "thought vector" that is progressively enriched—starting from a token embedding, then augmented by attention mechanisms that look back at previous tokens, and further refined by MLPs that inject learned patterns. This architecture exemplifies the assembly-line concept: each layer reads the current residual state, performs two independent transformations (attention and MLP), and writes the enriched vector back through residual connections. The final thought vector is then unembedded to predict the next token's logits across a 1,024-word vocabulary.

- Parallelization and Computational Efficiency:

A key insight is that despite the apparent sequential flow of "UCLA → is → at → Westwood," the backward pass through backpropagation factors cleanly and parallelizes across both positions and attention heads. Matrix multiplications can be batched using vectorized BLAS operations, attention gradients for different positions are positionwise independent once weights are known, and MLP gradients exploit the linear structure with pointwise nonlinearities. This parallelizability—enabled by the residual stream's structure and the decoupling of spatial positions in self-attention—is fundamental to why Transformers can scale to billions of parameters on modern GPUs/TPUs. The passage underscores that Transformers trade sequential dependency for global receptive fields and massive parallelism, making them superior to RNNs for long-range reasoning while remaining computationally efficient during training and inference.

- Diagrams


Output Prob
Softmax
logits = U @ r4
U ∈ ℝ^6×4, logits ∈ ℝ^6
r4 = r3 (x) mlp_out2
mlp_out2 = W_down2 @ h2
h2 = ReLU(W_up2 @ r3)
r3 = r2 (x) attn_out2
attn_out2 ∈ ℝ^4
Attention2
Q2 = W_Q2 @ r2
K2 = W_K2 @ r2
V2 = W_V2 @ r2



1. input → embed
2. layer 2-3: attention layer and MLP layer
3. unembed → softmax(s) → prob (out)

```
graph BT
    A["Input: 'at'"] --> B["Embed: E[3]"]
    B --> C["r0 = embedding<br/>r0 ∈ ℝ^4"]

    C --> Q1["Q1 = W_Q @ r0<br/>W_Q ∈ ℝ^4×4, Q1 ∈ ℝ^4"]
    C --> K1["K1 = W_K @ r0<br/>W_K ∈ ℝ^4×4, K1 ∈ ℝ^4"]
    C --> V1["V1 = W_V @ r0<br/>W_V ∈ ℝ^4×4, V1 ∈ ℝ^4"]

    Q1 --> ATT1["Attention1<br/>softmax(Q1*K1^T)*V1"]
    K1 --> ATT1
    V1 --> ATT1

    ATT1 --> ATTN_OUT1["attn_out1 ∈ ℝ^4"]

    ATTN_OUT1 --> SKIP1["r1 = r0 (x) attn_out1"]
    C --> SKIP1

    SKIP1 --> MLP_UP1["h1 = ReLU(W_up1 @ r1)<br/>W_up1 ∈ ℝ^8×4, h1 ∈ ℝ^8"]
    MLP_UP1 --> MLP_DOWN1["mlp_out1 = W_down1 @ h1<br/>W_down1 ∈ ℝ^4
×8, mlp_out1 ∈ ℝ^4"]

    MLP_DOWN1 --> SKIP2["r2 = r1 (x) mlp_out1"]
    SKIP1 --> SKIP2

    SKIP2 --> Q2["Q2 = W_Q2 @ r2"]
    SKIP2 --> K2["K2 = W_K2 @ r2"]
    SKIP2 --> V2["V2 = W_V2 @ r2"]

    Q2 --> ATT2["Attention2"]
    K2 --> ATT2
    V2 --> ATT2

    ATT2 --> ATTN_OUT2["attn_out2 ∈ ℝ^4"]

    ATTN_OUT2 --> SKIP3["r3 = r2 (x) attn_out2"]
```

```
SKIP2 --> SKIP3

SKIP3 --> MLP_UP2["h2 = ReLU(W_up2 @ r3)"]
MLP_UP2 --> MLP_DOWN2["mlp_out2 = W_down2 @ h2"]

MLP_DOWN2 --> SKIP4["r4 = r3 (x) mlp_out2"]
SKIP3 --> SKIP4

SKIP4 --> UNEMB["logits = U @ r4<br/>U ∈ ℝ^6×4, logits ∈ ℝ^6"]
UNEMB --> SOFTMAX["Softmax"]
SOFTMAX --> OUTPUT["Output Prob"]

style C fill:#e1f5ff
style SKIP1 fill:#e1f5ff
style SKIP2 fill:#e1f5ff
style SKIP3 fill:#e1f5ff
style SKIP4 fill:#e1f5ff
```

Problem2

Please play with the PyTorch code provided by the following webpage:
https://github.com/karpathy/nanoGPT
Write a brief explanation of the code and show your results. You can explore the code by
varying the design parameters.

- nanoGPT Explanation

nanoGPT is a minimal, production-oriented implementation of GPT language models created by Andrej Karpathy. Unlike educational frameworks that prioritize pedagogy, nanoGPT prioritizes simplicity and efficiency—condensing the entire training pipeline into ~300 lines of code each for training and model definition. The project enables researchers and
practitioners to train GPT-2-scale models (124M parameters) from scratch or fine-tune pre-trained weights on custom datasets without requiring deep infrastructure expertise.

Core Components:

1. model.py - Defines the GPT Transformer architecture: token/position embeddings, stacked attention and MLP layers with residual connections, and the final unembedding for logit predictions. Optionally loads pre-trained OpenAI weights.
2. train.py - Implements the training loop: mini-batch SGD with AdamW optimizer, distributed data parallelism across GPUs, learning rate scheduling, and gradient checkpointing for memory efficiency. Supports ~4-day training of GPT-2 on 8 A100 GPUs using OpenWebText.
3. sample.py - Generates text by autoregressive decoding: starting from a prompt, the model repeatedly samples next tokens from its predicted distribution, demonstrating the learned language patterns.
4. Data Pipeline - Tokenizes raw text (Shakespeare, OpenWebText, etc.) into integer sequences for efficient batching and training.

Key Insights for the Chapters from the Book.pdf:

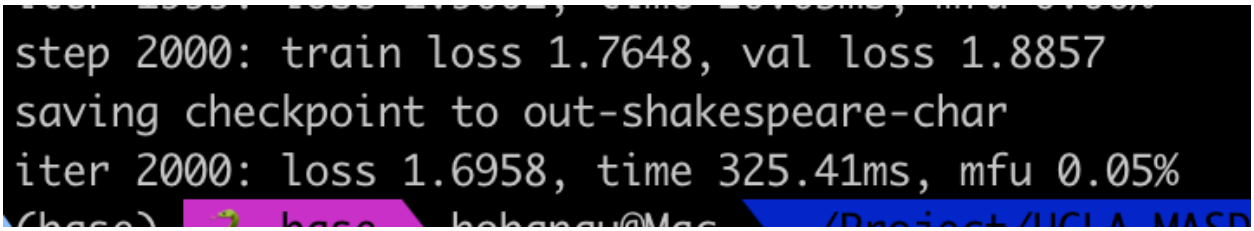
- Chapter 2 (MLPs): The model stacks residual blocks alternating attention and MLP, consistent with the assembly-line concept where each layer enriches the residual stream.
- Chapter 3 (CNNs): While not a CNN, the attention mechanism resembles local computation scaled globally—every token attends to all prior tokens (in autoregressive mode), replacing convolution's local receptive fields with full context.
- Chapter 4 (RNNs): Unlike RNNs that compress history into a single hidden state, nanoGPT caches all prior key-value pairs, enabling perfect long-range recall—the trade-off Transformers make for parallelizability.
- Chapter 5 (Transformers): nanoGPT is a direct, minimal implementation of the GPT decoder stack: each block reads the residual stream, applies context retrieval (attention) and memory retrieval (MLP), and writes back.

What this practice about?

The codebase democratizes LLM training—users can train on custom text (poetry, code, domain-specific data) with modest compute, fine-tune existing checkpoints, and experiment with architectural modifications. Its ~600 lines of core logic make it an ideal sandbox for understanding how Chapter 5's concepts (embeddings, attention, residual streams, decoding)
translate to working code.

1. training shakespeare_char.py using CPU

```
python train.py config/train_shakespeare_char.py --device=cpu --compile=Fal
se --eval_iters=20 --log_interval=1 --block_size=64 --batch_size=12 --n_layer
=4 --n_head=4 --n_embd=128 --max_iters=2000 --lr_decay_iters=2000 --dro
pout=0.0
```



2. generate sample like:

```
/opt/anaconda3/lib/python3.12/site-packages/torch/cuda/__init__.py:63: Futur
eWarning: The pynvml package is deprecated. Please install nvidia-ml-py inst
ead. If you did not install pynvml directly, please report this to the maintainers
of the package that installed pynvml for you.
  import pynvml  # type: ignore[import]
Overriding: out_dir = out-shakespeare-char
Overriding: device = cpu
number of parameters: 0.80M
Loading meta from data/shakespeare_char/meta.pkl...

I by done what leave death,
And aproposely beef the are and sors blate though wat our fort
Thine the aftior than whating bods farse dowed
And nears and thou stand murs's consel.
```

MEOF:
Sir, should and then thee.

GRICHARD one, enceling:
Or house
Your to do the all in despery I parve porge,
And shall not aple bralance the my plife friend.
Why should not seed liSian be me,
A light strought weart a king here shall
And so arttion I breatio:
Trale time him the witHer's bout thing thy a moy
A to therr

To furd of sonce on it my lord my prove.

ICIRIIA:
He had and I clann what would sy pile.

MYUKE MIOF OYCE:
Shall
Pule are wen way bane and thought I a the thems
Thet me staing and ous the broted a well.

MUDINE:
Have was he sir, tye as tor thee o, the beod
And I sto be in to do bun a go.

BORWARDch:
That a heaven my cournamants, wo that art is way.

QUEEN CARWIUS:
For dight erge masters: lold the had in but sway faulter thing miring conserly.
Why thouf than souch low the kings, bot here,

'I tha

To lack evice in spray.

PORIOLEN:
Come?

JULIET:
You ! it dother?

That lo, that sur, Securtion; as and I that in
A call in a world intree hous parcious thought the
less lordy this high stater own for of dight;
But twith she have shoul pray wave to to spice here such;
The be the to done do infelf anate.

JULIET:
If you speak:
Thoug you heart my outh be.

BUTHOK:
I powel, tor you leath a it thought helf;
And not take waul us like! Would I have never?
There and fuar souring stay frear in clifest,

Thire trech theirssing fily with good a be Romate
stanted I all swand you garely faten!
Thy onse lidy for hibe.

SISABELLA:
If we it deay is word wherepak.

LLEO:
But deeds murige have you should you shoulds detle,

So would he are recuren forth heir of,
Of this that fair my kending, my be reave shee.
That have heer dedy sing the the couse bease.

MERDIEL:
I heard be ladick in death in the my weard,
I should the are good you that one bothan were
To good dight, should is mouse enem your Claiin
Tha

To the is rarne that fath persising with's not of his me?
The love you love word the him him,
As hat worth the lest the a harter presart and lord;
Bet lordow Rick, and muwil I neave warn theirs in death,
And more his think shalt coors you huse say.

VELDO:
As I wail of you a day all bepence the pard,
Bust that a love king a gace man,
As in my later sood I coursues my the time.
ISA ward I dus we thouse the condeling him man.

KING YLOK:
Fy doou have king a which warty the of the our
Lades, this b

I were they to dower thy lought sur,
Fer hight mother us on toth, the not a more
Forcal the server the preat and heaver where as he
and a a with hast my the hibst the wril not
Whoo the undring to poon a the sore, wough the are
The be and so proond lought proung and and our bouness
On off the thou, the sir but the takes?

Whild be fort them: pourt, therefolt: shall stay I
As you mall should and there to faid's here the teverd:
What he pristing with to thight take no their!

ARCENIO:
We me the hav

That of the Emand remporrant tell dlep sheed
Met murder him his repustred bid but punnaroughter.

LUCIO:
The sie, art what a carse so forgelied3 this the kning,
Thue in he are amewaring of that fort you Wast in create,
That beet he's shanged ward the exty
But seand shand and Nupe abuly, I wartice so vee
The counsed word mophe hold own so inde the under:
He should my grief pouns the lade this I know,
All me, are would couses to with me ere's sone;
Off a infeared with in the when we repories,
That

And yeat shall which hath good deefing arm gull that the heave fach
That the made this be low mard,
That his all he be me oun toought pardunges not
A Commoned heir by the whom is proud,
I sill and and for in she were the had.

CARIO:
Should and perven hy herser named.
What I will me shall I do hopse net of your cout
Hat hus face the worse popes the lie
The done brow hous infeer in manes him not:
Which with should in dies you:
With Varrent your not you wardand,

And he not han worging dight nor th

MENIO:
I would it gear; and my hands wome
Thin, he hadse as may beed the beatt be that of
My sorder agant nempone him be I sujuse ofse
That sway be rest, no he proppur and be of shall go
Onge, onry house hemon the course,
The is a fiend to shall Levoule quores'd onsel
That I comed a my kid not angead of treant
the fail the of coursier groam'dly
The a look him the provereds you him the Prase.

COLUS:
Vord:
Then sir, not whom ame lest there pastarm'd doge and
That here had for peast. I sain, the t

Romest and the facll by lords him,
Not that lop, thou are frien when,
Which nought ye fair safy same, wrow you,
what not the do con the sunry coursel
The ill hims urn this one arme our too?

LADO:
You beand our king upor teept more put thein;
You do mile spord of unfent all of death lied,
As a that he beck his not with tie my ret our
Than with the wommer not lonks consomes'd?

MORCUSE:
That you have bod love mort are sead the how.

```
KING RIC& Eymewing:
Away hat baine whem ske fall be thee leath,
```

3. hyper parameter explained

```
--block_size=64
- Context window for language modeling
- nanoGPT training:
  * Read character stream from file
  * Create overlapping windows of 64 characters
  * Predict next character after each 64-char window

- Example with Shakespeare:
  Input (64 chars): "ROMEO: But, soft! what light through yonder window b"
  Target:           "OMEO: But, soft! what light through yonder window br"
                    (shifted by 1 position)

- Shakespeare analysis:
  * Avg English word ≈ 5 chars
  * 64 chars ≈ ~12 words of context
  * Reasonable for character-level prediction

--batch_size=12
- During training iteration:
  * Load 12 independent text chunks, each of length 64
  * Total tokens processed: 12 × 64 = 768 chars
  * All 12 chunks processed in parallel (if GPU)
  * On CPU: Sequential, slower

- For Shakespeare:
  * Epoch ≈ (file_size / (batch_size × block_size)) iterations
  * Shakespeare ~1MB = ~1M chars
  * Iters per epoch ≈ 1M / (12 × 64) ≈ 1300 iterations
  * 2000 iters ≈ 1.5 epochs (moderate underfitting)
```

```
--eval_iters=20
- Validation loop:
  * nanoGPT splits data: 90% train, 10% val
  * Eval: Run 20 batches on validation set
  * Compute avg validation loss
  * Print every log_interval

--log_interval=1
- Print training loss every iteration
- On CPU this is VERY VERBOSE and slows things down
- Consider: --log_interval=10 or 50 for CPU

--lr_decay_iters=2000
- nanoGPT default: Uses cosine decay
- Formula: lr(t) = lr_min + 0.5 * (lr_max - lr_min) * (1 + cos(π * t / T))
  where T = lr_decay_iters = 2000

- Learning rate schedule:
  iter 0:    LR at maximum (typically 0.001 or 6e-4)
  iter 500:  LR at ~90% of max
  iter 1000: LR at ~50% of max
  iter 1500: LR at ~10% of max
  iter 2000: LR at minimum (~1e-5)

- Effect on convergence:
  * Start high: Fast initial learning
  * End low: Fine-tuning, stabilization
  * With max_iters=2000 = 1.5 epochs: Good annealing strategy

Vocabulary:
├─ Character-level (not word-level!)
├─ vocab_size ≈ 65 (ASCII printable + newline)
│  (a-z, A-Z, 0-9, punctuation, space, newline)
└─ Each char → embedding vector ∈ ℝ^128

--n_embd=128 (d_model)
```

- Each character embedded in 128-dimensional space
- Transformer processes sequences in this space
- Typical nanoGPT defaults:
  * Small: n_embd=64
  * Medium: n_embd=128 ✓
  * Large: n_embd=256-512

- Memory per embedding: 65 × 128 = 8,320 floats
- This is TINY compared to word embeddings

--n_head=4
- Attention heads: 4
- Head dimension: d_head = 128 / 4 = 32
- Per-head attention operates in ℝ^32
- Multi-head attention:
  ├─ Head 1 attends to dimension 0-31
  ├─ Head 2 attends to dimension 32-63
  ├─ Head 3 attends to dimension 64-95
  ├─ Head 4 attends to dimension 96-127
  └─ Results concatenated back to 128

- Why 4 heads?
  * Specialization: Each head learns different patterns
  * Example from language:
    - Head 1: Learn when to continue words vs. start new
    - Head 2: Learn punctuation patterns
    - Head 3: Learn grammatical structure
    - Head 4: Learn semantic patterns

--n_layer=4
- Transformer depth: 4 blocks
- Information flow:
  Layer 0: Extract low-level patterns (character sequences, words)
  Layer 1: Combine patterns (phrases, common sequences)
  Layer 2: Learn longer dependencies (clauses, scenes)
  Layer 3: High-level structure (dialogue, narrative flow)

```
- For Shakespeare:
  * 4 layers is reasonable for character-level
  * Can learn: characters speak, stage directions, acts/scenes
  * Might struggle with very long-range dependencies

--dropout=0.0
- NO dropout during training
- For Shakespeare (1MB):
  * 2000 iterations ≈ 1.5 epochs
  * Model likely underfits (not overfit) with 1.5 epochs
  * So dropout=0.0 is fine

- When to add dropout:
  * If training >5 epochs: Add dropout=0.1-0.2
  * If small dataset + long training: Add dropout
  * For your setting: dropout=0.0 ✓ OK
```

Problem 3

Please read book chapters 1-5. For each chapter, write a brief review.

Chapter 1:

Chapter 1 establishes the mathematical and conceptual foundations of machine learning. It begins with simple linear models ($s_i = x_i\beta$) and introduces core concepts like loss functions (MSE for regression, cross-entropy for classification), optimization methods (gradient descent and closed-form solutions), and maximum likelihood estimation. The chapter covers key activation functions (sigmoid for binary classification, ReLU for deep networks) and explains the geometric interpretation of linear regression as projecting data onto a feature subspace. It also introduces logistic regression as a probabilistic framework for classification, where predictions are mapped to probabilities via the sigmoid function.

The chapter then explores model complexity and generalization, addressing the counterintuitive double descent phenomenon—where heavily over-parameterized

models can still generalize well despite perfect memorization of training data. It discusses the bias-variance tradeoff, implicit regularization through training dynamics, and optimization techniques like gradient descent with momentum. The foundation established here is crucial: linear models with non-linear activations enable the hierarchical feature learning of deep neural networks, and understanding the balance between memorization and generalization is essential for building effective machine learning systems.

Chapter 2:

Chapter 2 bridges classical statistical models and neural networks by showing how logistic regression naturally becomes a perceptron layer, then extends to multi-layer perceptrons (MLPs) with non-linear activations. The chapter introduces single and multi-layer networks with ReLU units that function as piecewise-linear interpolants, then explains back-propagation—the efficient algorithm for computing gradients through the chain rule. It covers optimization techniques (mini-batch SGD and Adam), proper weight initialization strategies (Xavier and He), and multi-class classification via softmax. The key insight is that over-parameterized networks, when trained with gradient descent, act as implicit regularizers favoring smooth, minimum-norm solutions that generalize well despite perfect training accuracy.

Beyond basic architecture, Chapter 2 explores higher-level constructs built on embeddings: word embeddings (dense representations supporting analogical reasoning), associative memory systems, and recommender systems using user-item embeddings. It introduces crucial regularization and normalization techniques —RMS normalization for scale-invariant gradient flow and dropout for fault tolerance—and explains how semantic concepts are distributed across orthogonal subspaces in embeddings. These techniques form the foundation for modern deep learning systems, demonstrating how to scale networks effectively while maintaining robustness and generalization.

Chapter 3:

Chapter 3 presents Convolutional Neural Networks (CNNs) as vector-based computer programs that exploit the spatial structure of images through three core inductive biases: translation invariance, local connectivity, and hierarchical composition. The chapter shows how CNNs build representational hierarchies

from low-level edge detectors (3×3 receptive fields) to abstract object parts (7×7 to 15×15) and finally global semantic representations. Convolutional layers efficiently apply the same learned filters across all spatial positions, reducing parameters and data requirements. Fully connected layers then compress the spatial grid into high-level embeddings. The architecture is formalized through dimensionality analysis and channel-wise convolutions, with 1×1 convolutions enabling efficient cross-channel interactions and dimension manipulation.

The chapter derives efficient backpropagation algorithms for convolutional layers, showing how gradients flow through spatial positions and across strides, and explaining why CNNs parallelize naturally across spatial positions, feature channels, and batch items on modern GPU/TPU hardware. By enforcing the assumption that patterns are local and reusable, CNNs dramatically reduce the model's parameter count and generalization error on visual tasks compared to fully connected networks. This structure transforms raw pixels into hierarchical representations that capture increasingly abstract visual concepts, forming the foundation for modern computer vision applications.

Chapter 4:

Chapter 4 presents Recurrent Neural Networks (RNNs) as neural programs that evolve through two orthogonal highways: memory streams across time and residual streams across depth. Starting with the vanilla tanh recurrence for next-word prediction, the chapter diagnoses the vanishing gradient problem that arises from repeated multiplication by weight matrices with eigenvalues less than 1. It then introduces memory stream innovations—additive skip connections that preserve information horizontally—and formalizes this in LSTMs with gating mechanisms (forget, input, and output gates) that selectively control what information flows through the cell stream. The chapter shows how residual connections work vertically through layers, enabling stable gradient flow through depth in multi-layer RNNs. Both mechanisms act as assembly lines that accumulate updates while maintaining direct gradient pathways.

The chapter unifies RNNs with temporal CNNs through state space models (SSMs), which can be viewed equivalently as recurrences for sequential inference or as convolutions for parallel training. It extends this view to continuous-time dynamics and introduces Mamba, a selective SSM that adapts its parameters based on input to combine RNN efficiency with Transformer-like flexibility. The

chapter concludes with a novel perspective: quantum mechanics recast as a fixed RNN where the Schrödinger equation governs hidden state evolution and measurements act as input embeddings, suggesting that RNNs provide a universal computational framework spanning classical machine learning, iterative algorithms, and quantum physics.

Chapter 5:

Chapter 5 presents Transformers as residual assembly lines where high-dimensional thought vectors accumulate nearly orthogonal features across layers through two complementary retrieval mechanisms: attention for context-dependent lookups and MLPs for associative memory recall. Starting from distributed token embeddings decomposed as superpositions of semantic components, the chapter traces the full Transformer architecture from positional encodings through causal self-attention and MLP layers in the residual stream. Key-value caching enables efficient inference by maintaining persistent notes of prior context, explaining GPT's superior long-range dependency modeling compared to RNNs. The chapter contrasts encoder-decoder (translation) models with decoder-only GPT architectures, and provides architectural details and parameter counts for reference models from the original 2017 Transformer through GPT-3 with 96 layers and 12,288 dimensions.

The chapter details modern training pipelines and scaling trends, including empirical power laws relating loss to model parameters, data, and compute, along with optimal allocation strategies (Chinchilla scaling). It covers the two-stage training approach: unsupervised pre-training via next-token prediction followed by instruction tuning on curated task examples. Critically, it explains Reinforcement Learning from Human Feedback (RLHF), where a learned reward model scores responses via Bradley-Terry comparisons and policy gradients optimize the language model to maximize expected reward while using Proximal Policy Optimization for stability. The chapter closes with vision and multimodal variants: Vision Transformers (ViT) that patch images and apply standard Transformer blocks, and CLIP, which aligns vision and language encoders through contrastive learning on image-text pairs, enabling zero-shot classification and semantic search.