Date: September 4
Hours: 2
What I Did: Researched and conceived up of multiple ideas but ultimately landing on possibly doing a prediction ai, where I can possibly train an ai to predict what a user will do. Maybe either a game or word guesser
What I learned/achieved: Not much since it was more of me looking at what I wanted to do, but found interesting information on predicative ai and possibly training it with if it gets information correctly or not

Date: September 7, 2025
Hours: 6 hours
What I Did:

- Created a local SQLite database using Python.
- Implemented schema creation for two main tables:
    - sentences → stores complete text inputs for future training or testing.
    - ngrams → stores combinations of word sequences (context, next_word) used for prediction logic.

What I Learned/Achieved:

- Learned how to structure a database schema for language processing tasks, specifically for n-gram prediction systems.
- Realized the importance of initializing PRAGMA foreign_keys = ON; to maintain data integrity when working with multiple related tables.
- Learned that SQLite lets me create a database within my project using python rather than needing to manually create my own database
- Learned how n-grams represent small chunks of text (for example, bigrams are two-word sequences like “thank you” or “good morning”), and how counting their frequencies helps predict what word is likely to appear next.
- Understood that this forms the foundation for a predictive typing model — by storing how often each word follows another, the system can later make educated guesses about the next word a user might type.

Date: September 12
Hours: 2
What I Did/ Learned:

- Began writing a Python script to insert sentences into the SQLite database.
- Attempted to connect to typing.db and insert basic example sentences like “I like pizza” and “I am learning.”
- Tested multiple insert commands and encountered issues with duplicate entries and incorrect row retrieval.
- At first, the script would re-insert the same sentences every time it ran, creating duplicates.
- Faced confusion with how to properly handle database connections and commits — some data didn’t appear to save because commits weren’t executed after inserts.

- Realized that small details — like query output format or missing commits — can completely break database logic.

Date: September 14
Hours: 2
What I Did/Learned:

- Fixed previous issues with duplicate entries and missing commits.
- Added logic to skip already-existing sentences before inserting new ones.
- Learned how to use executemany() with parameterized queries to securely and efficiently add multiple records at once.
- So lets understand how we did this

```
# Avoid inserting duplicates
existing = {r["text"] for r in conn.execute("SELECT text FROM sentences")}
to_add = [(s,) for s in SENTENCES if s not in existing]

if to_add:
    conn.executemany("INSERT INTO sentences(text) VALUES (?)", to_add)
    conn.commit()
    print(f"Inserted {len(to_add)} new sentences.")
else:
    print("No new sentences")

rows = conn.execute("SELECT id, text FROM sentences ORDER BY id DESC LIMIT 10").fetchall()
print("\nRecent sentences:")
for r in rows:
    print(f"  #{r['id']:>3}  {r['text']}")
```

-
- I wanted to make sure we don't insert the same sentence twice into the database. When you're training language models, repeating data skews results
- The first line goes through every row in the sentences table and builds a Python set of the text values. Sets are great for this because they automatically avoid duplicates.
- The following line then loops through your existing sentences and checks each one to see if it's not already in that set.
    - If it's new, it's prepared for insertion.
- If there are new sentences to add, we execute a "conn.executemany" which basically saves us time depending on how many new sentences are added, like if i wanted to take in large text. Else we would do it one by one
- We then just show the 10 most recent rows from the table
- Learned that (which you can see in the final line) the ":>" simply allows the ids to line up evenly which makes it easily more visually readable

Date: September 16
Hours: 1
What I learned/Did:

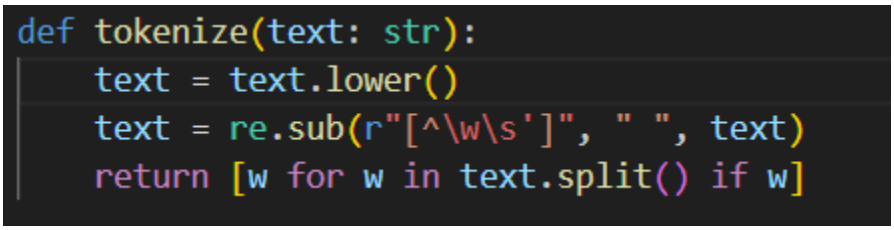
- We have a way to store sentences and we have a basic understanding of everything, now time for actual ways to build our n-grams and then afterwards, a possible front-end for a user to use to store the sentences via typing
- Learned how we could possible build up our ngrams - Example: “I Like Pizza”
    - Unigram
        - I
            - Has no previous context
    - Bigram
        - “I like” and “like pizza”
            - Has a context of previous word
    - Trigram
        - ‘I like pizza”
            - Context uses combination of the two previous words to predict the next

Date: September 17
Hours: 1.5
What I Did and Learned:

- What is an Upsert and Why?
    - Make sure the database doesn’t create duplicate entries when counting n-grams. Every time the builder runs, it may encounter the same n-gram again — for example, “I like” could appear in multiple sentences.
    - We needed a way to either insert a new record or update the existing one if it’s already in the database.
    - Using something called an “upsert” — a combination of update and insert.
    - In SQL, this is written as INSERT ... ON CONFLICT (...) DO UPDATE.
        - It means:
            - If the row (based on its unique key) doesn’t exist, insert it.
            - If it does exist, update the existing one (in our case, increase its count).
    - Learned that upserting is an efficient and safe way to keep n-gram counts accurate across multiple runs.
    - Understood that SQLite’s ON CONFLICT makes it perfect for word frequency tasks since we often encounter overlapping data.

Date: September 20
Hours: 5
What I Did/Learned

- Tokenize/Regex
    - Prepare each sentence in our database for n-gram generation by breaking it down into clean, lowercase words (called tokens).
    - We also needed to remove unwanted characters like punctuation, symbols, etc. that don’t contribute to language prediction.

- This process is called tokenization, and it's the first step in alot of Natural Language Processing (NLP) systems.
- 
```
def tokenize(text: str):
    text = text.lower()
    text = re.sub(r"[^\w\s']", " ", text)
    return [w for w in text.split() if w]
```
- We first set the text to lowercase via .lower()
    - Converts everything to lowercase so that "Pizza" and "pizza" are treated as the same word.
    - This prevents unnecessary duplication in our n-gram data.
- The next line uses regex expression
    - The pattern [^\w\s'] means: match anything that is not a word character, space, or apostrophe.
    - Everything else gets replaced with a space.
    - Example: "I like pizza!!!" → "i like pizza"
- Then text.split() just splits the text into a list of words
- If w → means that remove any empty strings, such as multiple empty spaces

- Why this matters
    - Predictive typing depends on exact word sequences. Without consistent tokenization, "Pizza!" and "pizza" would count as two different words.
    - Regex filtering removes unwanted characters that do not give us any contribution to our data and make it messy — punctuation marks, emojis, or special symbols that don't carry predictive meaning.
- Learned
    - Learned that regex is a powerful tool for defining text-cleaning rules in one line, instead of manually looping through every character.
    - Understood that tokenization transforms language into structured data the computer can analyze and count.
    - Saw how even small things (keeping apostrophes, lowercasing, spacing) directly affect data accuracy.

September 24
Hours: 5 hours
What I Did/Learned:

- Transforming our sentences into n-grams via a Build/Store function
    - We create three counters
        - uni for unigrams (single words)
        - bi for bigrams (pairs of words)
        - tri for trigrams (three-word sequences)
    - Using Counter automatically handles incrementing frequencies with each occurrence.

- Here is how we count and is the fundamental understanding of how predicative ai works on taking in data
    - Count n-grams
        - Unigrams:
            - No context (""), only the word itself.
            - Example: “pizza” → ("", "pizza")
        - Bigrams:
            - The current word predicts the next one.
            - Example: “I like pizza” → ("I", "like") and ("like", "pizza")
        - Trigrams:
            - Two previous words predict the third.
            - Example: “I like pizza” → ("I like", "pizza")

```
# Unigrams (context = "")
for i in range(len(words)):
    uni[("", words[i])] += 1

# Bigrams (context = previous 1 word)
for i in range(len(words) - 1):
    ctx, nxt = words[i], words[i + 1]
    bi[(ctx, nxt)] += 1

# Trigrams (context = previous 2 words)
for i in range(len(words) - 2):
    ctx = f"{words[i]} {words[i + 1]}"
    nxt = words[i + 2]
    tri[(ctx, nxt)] += 1
```

    - After all counting is done:.
        - For each entry in our counters, we call upsert().
        - If the record exists → its count increases.
        - If it doesn’t → it gets inserted as a new entry.
        - Once all n-grams are processed, conn.commit() saves everything permanently.

-

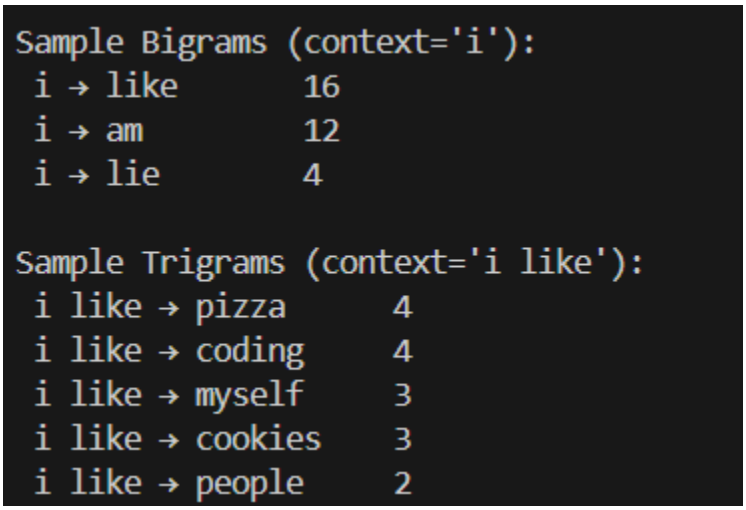
```
with get_conn() as conn:
    def upsert(n, ctx, nxt, c):
        conn.execute(
            """
            INSERT INTO ngrams(n, context, next_word, count)
            VALUES (?, ?, ?, ?)
            ON CONFLICT(n, context, next_word)
            DO UPDATE SET count = count + excluded.count
            """,
            (n, ctx, nxt, c),
        )

    for (ctx, nxt), c in uni.items(): upsert(1, ctx, nxt, c)
    for (ctx, nxt), c in bi.items():  upsert(2, ctx, nxt, c)
    for (ctx, nxt), c in tri.items(): upsert(3, ctx, nxt, c)
    conn.commit()
```

- Importance:
    - Turns our text into data that allows our very simple ai to learn and predict the next words
    - Saw how upserting helps work with our counter and not insert new sentences into the database that are already there and logically makes sense with what we are trying to achieve with counting n-grams

Date: September 25
Hours: 1
What I Did/Learned:

- Create a quick and simple way to see what's actually inside our ngrams table after building and storing all those word combinations.
    - This helps confirm that the n-gram building and database storage logic are working correctly.
    - It also gives us a look at which words appear most often and which patterns the model has started to learn.
- We Display Top Unigrams
    - Have a query grabs the 10 most frequent individual words (n=1) from the database.
    - Which are ordered by highest count → lowest, then alphabetically if counts are equal.
- We Display Sample Bigrams
    - Filters for n=2 (two-word combinations).
    - Specifically looks for all bigrams where the context is 'i'.
    - This lets us check what typically follows "I" in our sentences (e.g., "like," "am," "love").
- We Display Sample Trigrams
    - Filters for n=3 and context 'i like'.
    - Shows what words usually come after "I like," such as "pizza" or "coding."

- Each section helps ensure different levels of the model (unigram, bigram, trigram) are all updating and storing correctly.
    - 

    - 

- Really good visualizer on if our data is is being handled correctly

Date: September 27
Hours: 2
What I Did/Learned:

- Streamlit
    - Build a user interface for our language-model system using Streamlit, a lightweight Python framework that lets you turn scripts into interactive web apps without needing any HTML, CSS, or JavaScript.
    - This allows anyone—even non-programmers—to type sentences, store them in the database, and instantly see predicted next-word suggestions.
    - It runs entirely in Python: you just write normal code, and it automatically builds a clean, interactive webpage.
    - How It Works

        - Each time the user interacts (submits a form, checks a box, types text), Streamlit re-runs the entire script from top to bottom, caching values as needed.
        - Functions like st.text_area, st.checkbox, and st.form_submit_button create interactive input fields.
        - Live connection:
            - Because our database code and model functions live in the same script, Streamlit directly connects the frontend (user typing) and backend (SQLite + predictive model).
        - You can run locally with streamlit run app.py or deploy to cloud platforms such as Streamlit Cloud, making it instantly shareable.
- Why?
    - It provides an interactive app for our project, no separate framework required.
    - Lets us test predictions in real time: as soon as we type a sentence and submit it, the app updates n-grams and displays suggestions.
    - Streamlit keeps everything pure Python, avoiding frontend complexity.

Date: September 28
Hours: 4
What I Did/Learned:

- Types of smoothing
    - Learned about smoothing, which is a way to make predictions more reliable when the model encounters words or word combinations that weren't seen before in training.
    - Without smoothing, our predictive system could assign a zero probability to unseen phrases — which breaks language models, since it implies that something "can never happen."
        - LaPlace
            - Adds +1 to every word count before calculating probabilities.
            - Ensures no word has zero probability, even if unseen.
            - Works well for small datasets like ours where many word combinations may be rare.
        - Kneser-Ney
            - Doesn't overestimate words and takes context more importantly
                - Say we have San Fransisco and it was typed so much to the point where a non-Kneser-Ney algorithm would think San would be a common word to predict if say we had to backoff to unigrams (will explain backing off in a bit), San wouldn't be as popular as a choice if the words that proceeds is less dynamic in the amount of different words
        - These allow the model to generalize new predictions where we want no word to be seen as impossible to predict
    - Backing Off

- A technique used in predictive text and language modeling to make intelligent guesses even when certain patterns haven't been seen before.
- It allows our system to "fall back" to a simpler model (with less context) when higher-level data (like trigrams) is missing.
- In language models, we often predict a word based on the last few words (context).
- But what if a certain phrase has never been seen before?
    - Example: if "I love pizza" was seen but "I love sushi" wasn't, then the trigram ("I love sushi") has no data.
    - Without a fallback system, the model would assign a zero probability and stop predicting.
- How Backoff Works
- 1. Try the highest n-gram level first (most specific)
    - The model first looks for a trigram (context of 2 words, like "I love").
    - If that exists, it uses it for prediction.
- 2. If not found → fall back to a lower level
    - If no trigram match is found, the model "backs off" to the bigram level — using only one word of context (like "love").
    - If that still fails, it finally backs off to unigram — predicting based on overall word frequency regardless of context.
- 3. Combine results with smoothing
    - Each level's probabilities are adjusted using smoothing so the total still sums to 1.

Date: September 29
Hours: 1
What I Did/Learned;

- These past days have been more research in what I might try or use and this one seems a bit more complicated, used in more complicated neural networks wheras mine is more simple and that is Embedding and Vectors
    - In simple terms, an embedding is a vector (a list of numbers) that represents a word's meaning based on the context it appears in.
    - Instead of storing words as text strings, an embedding turns each one into a point (for example, 100-300).
    - Words with similar meanings are placed close together in that space.
        - Ex: (cat: 122 and dog: 130)
    - If I decided to integrate embeddings later:
    - I could replace n-gram probabilities with similarity scores between word vectors.
        - For example, if the user types "I enjoy," embeddings could find that "enjoy" is close in meaning to "like," and suggest "pizza" even though "I enjoy pizza" never appeared in training.
    - This would make predictions semantic rather than statistical.

Date: October 1 - 3
Hours: Around 5 hours
What I did/learned:

- Handled laplace smoothing in code
  - ```
    def laplace_scores(cands, total: int, V: int, alpha: float = 1.0):
        denom = total + alpha * V
        return [(w, (c + alpha) / denom) for (w, c) in cands]
    ```
  - Where alpha is defined as 1, handling unseen words, later realized this doesn't actually work since our model is so basic, it can't possibly no other words to suggest
- Have a fetch candidates functions that looks at what is most likely to appear after a given context via "next_word", will appear later in our predict next function
  - ```
    def fetch_candidates(conn: sqlite3.Connection, n: int, context: str):
        rows = conn.execute(
            "SELECT next_word, count FROM ngrams WHERE n=? AND context=?",
            (n, context),
        ).fetchall()
        return [(r["next_word"], int(r["count"])) for r in rows]
    ```
- Predict_next function - takes a string
  - Tokenizes it,
    - Tries trigram candidates first (most specific),
      - If none, tries bigram,
        - If none, falls back to unigram,
  - At whichever level returns candidates, it applies Laplace smoothing, sorts by probability, and returns the top-k suggestions.
- We also finished setting up our streamlit with our textbox
  - We run it in the terminal via
  - streamlit run app.py

Date: October 13
Hours: 3
What I did and learned:

- A GRU is a type of Recurrent Neural Network (RNN) architecture designed to handle sequence data, like text or time series.
- Unlike simple RNNs (which forget information quickly), GRUs use gates to control the flow of information — deciding what to keep from the past and what to update with new data.
- Each GRU cell has two main gates:
  - Update Gate (z): decides how much of the previous state should be kept.
  - Reset Gate (r): decides how much of the old information to forget.

- The model processes one word (or token) at a time and updates its internal “memory,” allowing it to learn patterns like grammar, context, and dependencies between words over long sequences.
- Learned layering
    - In neural networks, layering means stacking multiple layers of neurons or GRU units to increase learning complexity.
    - Each layer captures different levels of abstraction:
        - Lower layers detect simple patterns (word order, local dependencies).
        - Higher layers capture meaning, tone, and long-range relationships between words.
    - Layer Example
        - Embedding Layer: turns each word into a dense vector representation.
        - GRU Layer(s): learn how words depend on each other across time steps.
        - Conversion Layer: converts the GRU output into probabilities for the next word.
    - Our current model (n-grams) uses explicit counting, while GRUs learn these patterns automatically.
    - With GRUs, the model can understand longer phrases and non-repetitive structures without manually defining how many words to look at.

October 20
Hours: 3
What I Did/Learned:

- Pytorch
    - PyTorch is an open-source deep learning framework developed by Meta
    - It’s mainly used for:
        - Creating and training neural networks
        - Running tensor operations (multi-dimensional math like matrices and vectors)
    - Using GPUs for faster computation
    - PyTorch tracks tensors (the data) and operations (what you do with that data).
        - A tensor is just a fancy word for a multi-dimensional array of numbers — the same concept as a vector (1D) or matrix (2D).
    - Every time you do an operation (like add, multiply, or pass through a layer), PyTorch builds a dynamic graph that records it.
    - When it’s time to train, PyTorch can automatically compute gradients for all those operations, allowing your model to learn.
        - Gradient is basically how a weight changes so like if the weight of a word is -36 and it goes to -32 then the gradient is basically how much something changed by.
        - What are weights

        - Weights are the adjustable numbers inside a neural network that the model learns during training.
        - Each neuron connection has a weight that controls how strongly one input influences the output.
        - If a word is important, the weight attached to it will grow larger. If it's unimportant, the weight will shrink or approach zero.

Date: October 24
Hours: 3
What I Did/Learned:

- Learned about Optimization Loops or optim loops
    - An optimization loop is the repeated process that makes a neural network learn from data.
    - It's made up of five main steps:
        - Forward pass — Send input data through the model to make predictions.
        - Calculate loss — Measure how far those predictions are from the true answers.
        - Backward pass — Use backpropagation to calculate gradients for every weight.
        - Update weights — Adjust weights using those gradients.
        - Repeat — Do this many times until the model performs well.
    - Each loop is one training cycle, and over thousands of cycles, the model learns better weight values.
- Backpropagation
    - Backpropagation is the algorithm that teaches the model which weights caused the errors and how to fix them.
        - The model makes a prediction.
        - We compare it to the real answer → get a loss (error).
        - Backpropagation sends that loss backward through the network, calculating gradients for each weight.
    - But luckily for me Pytorch does this already, because otherwise there is a lot of steps and math in doing this
        - Audograd
            - Pytorch's system that automatically figures out the gradients of your tensors and model weights for you.
            - It is basically a built-in "math engine" that watches everything you do to your data, so it can later undo it step by step to figure out how each value changed the final result (loss).
            - Every time you perform an operation on a tensor that has requires_grad=True,
                - PyTorch:
                    - Records the operation in a computation graph.
                    - Remembers which other tensors were involved.

        - Knows how to calculate the derivative (gradient) of that operation.
    - Then, when you call .backward() on the final result (like your loss),
    - PyTorch traces the graph backward and calculates all the gradients automatically.

October 28 - 6 hours

- My next step is to develop the “training engine” for my capstone project. All the sentences I type into my Streamlit apps get saved into a SQLite database, and this file is responsible for taking that raw text, turning it into usable numeric data, feeding it into a GRU neural network, and then saving the trained model so the prediction UI can actually use it.
- Remade a tokenizer for this file, because it is such an important piece to my vocabulary and how it affects my entire model. It was a good thing to already have approached earlier in my n-grams file but now when I build my vectors and tensors, punctuations, and essentially all unnecessary things that come up in sentences will be nice to remove
- Today I spent time really understanding the heart of the text-processing pipeline: the build_vocab() function. Even though it looks simple at first glance, this one function entirely determines how the neural network sees language. If this step is wrong or shallow, everything downstream — the GRU embeddings, the training data, the predictions — becomes unstable or meaningless.

```python
def build_vocab(min_freq: int = 1):
    if not DB_PATH.exists():
        raise SystemExit("Database not found. Run init_schema.py first.")

    conn = sqlite3.connect(DB_PATH)
    conn.row_factory = sqlite3.Row

    counter = Counter()
    rows = conn.execute("SELECT text FROM sentences").fetchall()
    for r in rows:
        words = tokenize(r["text"])
        counter.update(words)

    conn.close()

    word_to_id = {
        "<pad>": 0,
        "<unk>": 1,
    }

    for word, freq in counter.items():
        if freq >= min_freq:
            word_to_id[word] = len(word_to_id)

    id_to_word = {i: w for w, i in word_to_id.items()}
    print(f"Vocab size: {len(word_to_id)} words")
```

- Purpose of the function
  - This function scans through all sentences stored in the database, extracts every unique word, counts how often each appears, filters out rare words if requested, and assigns each remaining word a unique numerical ID.
  - This numeric mapping is the foundation of how the neural network understands language.
- Code Bits
  - min_freq tells the function the minimum number of times a word must appear in the dataset to be included.
  - Right now we keep everything (min_freq=1), but later upping this up to 2, 3, or 5 helps remove unnecessary words.
  - <pad> and <unk>
    - Before I let the model learn anything, I have to initialize the vocabulary dictionary with two special words: <pad> and <unk>. Even though these

aren’t real English words, they serve structural purposes inside the neural network.
    - <pad> represents “nothing.”
        - It stands for empty positions when sequences need to be aligned to the same length.
        - Setting padding_idx=0 tells PyTorch:
        - The embedding vector for <pad> should ALWAYS be zero
        - Never update this vector during training
        - Mask it out so it doesn't influence the model’s learning
- <unk> stands for unknown words — any word that the model did not see during vocabulary building.
    - This prevents the model from crashing whenever the user types a new word.
    - Example:
        - If the vocabulary does not contain "blueberry" (maybe I never trained on fruit sentences), then:
        - "blueberry" → <unk> → ID 1
        - Why is this necessary?
            - Users always type new words
            - There is no way I can include every English word in the training data.
            - Prevents lookup failures
            - Without <unk>, the code:
                - self.word_to_id.get(word)
                    - would return None, crashing the model.
                    - “Unknown” gets its own embedding
                    - The model learns a vector that roughly represents “this is something unfamiliar.”
                    - This is better than deleting the word or ignoring it.
    -

November 2 - 5 hours

- Process of turning our n-grams into training samples for GRU
    - I already had an ngrams table in my database, where each trigram row looks like:
        - context = "the cat"
        - next_word = "meows"
        - count = how many times "the cat meows" appeared
    - The job of TrigramDataset is to convert those rows into the exact format the GRU needs during training:
        - Input: two context words → Output label: the next word
    - In neural network terms:
        - X: [id("the"), id("cat")] → tensor of shape
        - y: id("meows") → scalar tensor

-
```
ctx_ids = [
    self.word_to_id.get(w, self.word_to_id["<unk>"])
    for w in ctx_words
]
target_id = self.word_to_id.get(next_word, self.word_to_id["<unk>"])
```
- This is where it connects back to build_vocab:
    - Each context word is looked up in word_to_id.
    - If a word is not found (maybe it was filtered out or rare), I map it to <unk>.
- Same for the target word.
    - So "i like" → "pizza" might become something like:
        - ["i", "like"] → [12, 53]
        - "pizza" → 91
- Now it's in terms the GRU can actually handle integers instead of strings.

November 5 - 3 hours

- Devloped a len function and getItem function
    - Len function
        - This tells PyTorch how many samples exist.
    - GetItem function
        - This method is everything.
        - When the GRU trains, it never touches text, never touches English, never sees actual words.
        - Instead, the GRU learns from what __getitem__() returns.
        - What does this function do?
            - It takes one sample from my trigram dataset and turns it into tensors:
                - Input tensor: [ID_of_word1, ID_of_word2]
                - Target tensor: ID_of_next_word
        - If the trigram was:
            - "my dog" → "barks"
        - and my vocabulary mapping is:
            - my → 17
            - dog → 9
            - barks → 42
        - Input becomes tensor([17, 9])
        - Target becomes tensor(42)
        - Why tensors? Why not just numbers?
            - PyTorch models cannot train on Python integers or lists.
            - They need data in optimized format called tensors.
            - Tensors are like NumPy arrays but with GPU support, automatic differentiation, and the ability to track gradients.

November 8 - 4 hours

- Up until now, everything was vocab building, tokenizing, counting n-grams, and wiring up a dataset. But this section is where the model actually learns patterns from language. I focused on two core components: the Embedding layer and the GRU.
- Embedding Layer
    - If I simply fed raw IDs (like 10, 54, 77), the model would accidentally assume that 77 > 10 in some meaningful way, which is completely wrong. The embedding layer fixes this by learning a small vector for each word, like:
    - cat → [0.12, -0.55, 0.98, ...]
    - dog → [0.10, -0.51, 1.02, ...]
    - These vectors get shaped during training so that similar words cluster together. The cool thing I learned is that the embedding matrix is literally just a lookup table the model updates. It is basically the neural network's evolving "dictionary."
- GRU
    - the GRU (Gated Recurrent Unit) is the part of the model that gives it memory.
    - It reads the two-word context (like "the dog") one timestep at a time:
        - take word 1 embedding
        - update hidden state
        - take word 2 embedding
        - update hidden state
    - return the final state representing the whole phrase
    - This final hidden vector is the model's "understanding" of the context.
    - Then we use that to guess what the next word might be.

November 12 - 3 hours

LayerNorm

- GRUs produce hidden states that can sometimes explode or drift in scale. LayerNorm takes the last hidden vector and smooths it so all dimensions have a consistent distribution. This leads to:
    - more stable training
    - cleaner gradients
    - better generalization
- less "clutter" in predictions
- Before adding LayerNorm, my model occasionally produced weirdly skewed logits. After normalization, the outputs became noticeably more stable.
- A mistake I had early on was confusing LayerNorm with BatchNorm. BatchNorm works across the batch dimension, which doesn't fit well with sequential word embeddings. LayerNorm fixes scale per token, not per batch, which is better aligned with language modeling.
-

```
self.norm = nn.LayerNorm(hidden_dim)
self.fc_out = nn.Linear(hidden_dim, vocab_size)
```

- The second line basically takes the GRU's hidden state (a vector of size hidden_dim) and maps it onto a vector the size of the vocabulary.
- This means every single word gets a score — a "logit" — representing how likely it is to be the next word.
- Example:
    - If vocab size = 400, this layer outputs a vector of length 400.
    - The highest logit becomes the predicted next word.

November 15 - 4.5 hours
Forward Pass

- Today I focused solely on the forward pass, which I realized is basically the model's entire "thought process" condensed into a single function. Everything I built up to this point — the vocabulary, datasets, embeddings, GRU, normalization, etc. — all comes together here.

```
def forward(self, x):
    embedded = self.embedding(x)
    output, _ = self.gru(embedded)
    last_hidden = output[:, -1, :]
    last_hidden = self.norm(last_hidden)
    logits = self.fc_out(last_hidden)
    return logits
```

- First Line
    - the moment the model converts word IDs into dense, meaningful vectors.
- Second Line
    - the model tries to interpret the sequence. Each embedding is fed into the GRU one timestep at a time.
- Third Line
    - Using this representationj → (batch_size, sequence_length, hidden_dim)
        - This produces a vector that is the model's understanding of the input context.
- Fourth Line
    - A small line of code that makes our predictions more stable and smoother
- Fifth Line
    - This is the final transformation. The model now converts the hidden representation into a vector of size vocab_size, assigning a score to each word.

November 16 - 2 hours
Training Loop/Optim Loops

- Today I started dissecting the actual training loop — the part of the project where the neural network “learns” from data. I quickly realized how many things can quietly go wrong before the model even updates a single weight
- One of the first mistakes I made was calling model.eval() earlier for testing, then forgetting to switch it back.
    - There was no error thrown, but no learning was being done
- Before a model can even start learning, there are dozens of small details that must all align.
- The model must be in training mode via .train()
- The device must match between model and tensors.
- The DataLoader must return properly-shaped tensors.

November 17 - 6 hours
So this part of the code although small for today was incredibly hard to get around for how fragile it was

```
optimizer.zero_grad()
logits = model(ctx)        # (B, V)
loss = criterion(logits, target)
loss.backward()
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

- First line resets all the gradients before the next update. I knew this in theory, but today I saw firsthand what happens if you forget it or misplace it:
    - At first I did not even have this in my code, but as the training went on and after many many epochs, my predictions were so bad and chaotic that I spent a couple hours trying to figure out why
- Second line
    - Essentially, my model architecture
- Fourth Line does our backpropagation for us, which I learned almost a month ago that PyTorch does this for us
- A lot of failures or mistakes I saw were not blatant, and I needed some lines like the last line to make my training more stable as without that last line, loss between epochs were drastic, where one epoch had normal loss and then the next had way more loss
-

November 22
Developed my main for this file
This main() function basically does the following:
- Figure out where to run (CPU vs GPU).
- Build the vocabulary from your sentences.
- Build the dataset and DataLoader from your n-grams.
- Create the GRU model, loss function, and optimizer.
- Train for several epochs.

- Save the trained model + vocab so the Streamlit app can use it.
-
```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)
```
- PyTorch tensors and models don't just float in abstract memory – they live either:
    - on the CPU (normal system RAM), or
    - on a GPU (CUDA device) if I have one and PyTorch can talk to it.
- torch.device(...) is basically a tag that says "this thing belongs on the CPU" or "this thing belongs on the GPU".
- If I don't have this and I start mixing locations, PyTorch complains hard:
    - Model on GPU, data on CPU →
        - RuntimeError: Expected all tensors to be on the same device…
    - Model on CPU, data accidentally moved to CUDA → same type of error.
    - By centralizing the decision in this line and reusing device everywhere, I avoid:
        - hard-coding "cuda" and then crashing on a laptop with no GPU, or
        - hard-coding "cpu" and never using the GPU even if it's available.
-
```
model = NextWordGRU(vocab_size=vocab_size, embed_dim=64, hidden_dim=128, pad_idx=pad_idx)
model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-3)
```
    - Takes raw logits from the model → (batch_size, vocab_size)
    - Takes target word IDs → (batch_size,)
    - Computes how wrong the model is for each training sample.
    - This is your teaching signal — the lower it goes, the more the model aligns its predictions with the actual next words in your corpus.
    - Optimizer:
        - AdamW with learning rate 3e-3:
        - Adjusts weights in the direction that reduces the loss.
        - W version (AdamW) includes weight decay which can help generalization.

November 29 - 6-7 hours
Made another Streamlit app to let users see both the n-gram predictions and neural predictions after submitting their data into the other app.
Reminder that my first app was a sentence feeder and then we had a predictor at the bottom using our n-gram model which was as basic as we can get, here we can still visually show those n-gram predictions but now we also show our neural network and the most likely words to come next.

Here we visually can see different predictions for each one, but we know why'
N-grams are the actual words that followed after that particular unigram or bigram that was stored into our database
Our neural predictor essentially looks at our vectors and based off our GRU model that we made, predicts the next word based off the context of the previous word(s). Early on, with less

data, these predictions looked familiar, but as more data was loaded in, we saw them become very different as shown above.
BUT
We need so much more data, the data looks okay here with ‘The Cat” but because of how vast the English language is and the unlimited possibilities, you would essentially need thousands of data sets to be used to train the model to get a better representation of how a typical auto-complete function on your phone might work.

December 1 - 5 hours
All data at this point
The project is essentially finished, but to get better results on my predictions I either have ChatGPT conjure me up paragraphs after paragraphs or even go online to find free books or PDFS to upload to my sentence feeder upon which I re-train my model and re-analyze my findings.