Capturing Word Meaning in Vector Space: A Word2Vec and Attention Implementation

Natural language processing sits at the intersection of linguistics, computer science, and artificial intelligence, attempting to bridge the gap between human communication and machine understanding. This project explores the fundamental question: how can we represent the meaning of words in a way that computers can process and analyze? I implement word2vec from scratch, a technique that transforms words into dense vector representations capturing semantic relationships. These embeddings are then applied to both intrinsic evaluation tasks measuring word similarity and an extrinsic sentiment analysis task utilizing attention mechanisms. Through this implementation, I demonstrate how distributional semantics can be leveraged to extract meaningful patterns from text data while addressing computational efficiency challenges inherent in large-scale language processing.

> In my implementation of the `load_data` function, I focused on creating the essential data structures needed for word2vec training: the vocabulary mappings and token sequence. First, I read the input file line by line and tokenized each line using the provided RegexpTokenizer, which splits text into word tokens using a regular expression pattern. All text is converted to lowercase to normalize the vocabulary. This creates a long sequence of tokens that represents the entire corpus, ignoring line breaks. Next, I counted the frequency of each token in the corpus using Python's Counter class. This gives us the vocabulary distribution, which will be important for later steps like negative sampling.

I then created two essential mappings:

1. `word_to_index`: Maps each word to a unique integer ID
2. `index_to_word`: The reverse mapping from IDs back to words

These mappings are crucial for the embedding layer in our neural network, which takes integer IDs as input. By assigning each unique word a distinct ID, we can efficiently look up embeddings during training. Finally, I converted the entire token sequence into a sequence of integer IDs using these mappings. This `full_token_sequence_as_ids`

will be used to create our training examples by extracting target-context pairs. This initial implementation establishes the foundation for training word embeddings, though it doesn't yet implement the optimizations like rare word handling and subsampling that will be added later.

Output Analysis: When loading the tiny dataset, my function processed 20,985 tokens with 3,698 unique words. The most frequent words were typical function words like "the" (1,095 occurrences), "i" (657), and "a" (567). For the medium dataset, the scale increased dramatically to 10,607,824 tokens with 101,437 unique words. The frequency distribution followed a similar pattern with "the" (527,363) and "i" (351,103) being most common. Without token filtering or subsampling, these common words would dominate the training process, potentially affecting the quality of the learned embeddings.

> To implement efficient negative sampling, I created a table-based approach following the Word2Vec paper's methodology. This approach creates a large lookup table where words appear with frequencies proportional to their sampling probabilities. The key insight is that computing sampling probabilities on-the-fly would be inefficient during training. Instead, we create a pre-computed table of word IDs that we can randomly index into. The sampling distribution is based on word frequencies raised to the power of 0.75 (a value empirically found to work well in the original paper). This approach strikes a balance - it still favors common words but gives more weight to less frequent words than their raw frequency would suggest.

In my implementation:

1. I calculate the weight for each word as `count^0.75`
2. I allocate slots in the table proportionally to each word's weight
3. I ensure the table is completely filled by handling any rounding issues
4. I configure the random number generator to efficiently sample from this table

Forward-thinking approach: Even though the **`<UNK>` tokens aren't implemented yet in the `load_data()` function, I designed the negative sampling table to explicitly exclude** `<UNK>` tokens. This forward-thinking approach means that when I later implement rare word handling, the negative sampling will already be optimized to avoid introducing noise from artificial tokens.

Optimization technique: In the `generate_negative_samples` method, I implemented an important optimization that ensures we never sample the same word as the current context word. This prevents the model from receiving contradictory signals

(where the same word would be labeled as both present and not present in the context), which significantly improves training stability and the quality of the learned embeddings.

Testing and Validation: Before generating the training data, I implemented an `explore_context_examples()` function to test both tiny and medium datasets. This exploratory analysis allowed me to verify that my context extraction and negative sampling were working correctly by examining real examples of target words, their contexts, and the sampled negative examples.

Output Analysis: The negative sampling table was generated with a size of 1,000,000 elements, providing a rich pool for sampling. Testing on the word "this" produced negative samples like "put", "set", "read", "survival", and "disturbed" - words that would not typically appear in the immediate context of "this". Performance testing showed the implementation was efficient, completing 1,000 negative sampling operations in just 0.39 seconds for the medium dataset.

> I implemented a systematic approach to generate training examples for the Word2Vec model. Each example consists of a target word, its context words (positive examples), and randomly sampled negative examples.

My algorithm processes the entire token sequence as follows:

1. For each token in the sequence, I designate it as the target word.
2. I define a context window of size `window_size` words on each side of the target word, being careful to handle boundaries at the beginning and end of the sequence.
3. I extract all words within this window as positive context words, excluding the target word itself. Importantly, I also filter out any `<UNK>` tokens from the positive contexts, as including these would not provide meaningful semantic information.
4. To ensure all training examples have consistent dimensions (required for batch processing in PyTorch), I calculate how many positive examples each target word has, and then adjust the number of negative samples to maintain a consistent total size:

   ```
   num_negative = max_context_size - num_positive +
   num_negative_samples_per_target
   ```

5. I use the previously implemented `generate_negative_samples` function to select appropriate negative examples, ensuring they don't include the target word.

6. I combine the positive and negative examples, creating corresponding labels (1 for positive context words, 0 for negative examples).
7. Each training example is stored as a tuple of numpy arrays: (target_word_id, context_word_ids, labels).

Validation and Debugging: To ensure the training data was being generated correctly, I implemented validation code that displays sample training examples, showing the actual words rather than just IDs. This helped me verify that:

- Positive context words were indeed from the target word's context
- Negative samples were being correctly selected and labeled
- The `<UNK>` tokens were properly handled
- The dimensionality was consistent across examples

This comprehensive approach generates high-quality training data while handling edge cases and maintaining the dimensional consistency required by PyTorch's batch processing.

Output Analysis: With a window size of 2, the algorithm successfully extracted meaningful contexts. For example, the word "precious" appeared in context with semantically related terms like "memories", "share", and "most". The training data generation was efficient, processing over 10.6 million tokens in about 5 minutes (34,525 examples per second). Each training example maintained a consistent size of 24 elements (with varying numbers of positive samples and negative samples to maintain the fixed size), ensuring compatibility with PyTorch's batching. The example for the word "this" showed appropriate context words "was" and "bought" as positive examples (labeled 1), with 22 negative examples (labeled 0).

> I implemented the `init_emb` method to properly initialize the embedding matrices for both target and context words. Instead of using uniform initialization with a range of (-*init_range*, init_range), I chose to use normal initialization with mean 0 and standard deviation 0.1, which is another common approach for word embeddings. The initialization of embeddings is crucial for proper training of Word2Vec models. Unlike in logistic regression where weights can be initialized to zeros, in word embeddings, we need non-zero random values to break symmetry. If all embeddings started with the same values, they would receive identical gradients during backpropagation, preventing the model from learning different representations for different words.

I implemented this by using PyTorch's built-in initialization function `nn.init.normal_()`, applying it to both embedding layers:

```
def init_emb(self, init_range):
    nn.init.normal_(self.target_embeddings.weight, mean=0, std=0.1)
    nn.init.normal_(self.context_embeddings.weight, mean=0, std=0.1)
```

The standard deviation of 0.1 provides a good balance - large enough to break symmetry between words, but small enough to prevent initial exploding gradients. This initialization approach helped ensure stable and effective training from the start.

> In the *forward* function, I implemented the core computational logic of the Word2Vec model with negative sampling. This function takes target word IDs and context word IDs (both positive and negative examples), and computes a score for each target-context pair. The key insight of Word2Vec is that words appearing in similar contexts should have similar vector representations. The dot product between target and context embeddings, passed through a sigmoid function, gives the probability that the context word appears near the target word.

My implementation handles batched inputs efficiently:

1. I retrieve embeddings for both target words and context words from their respective embedding matrices
2. I reshape the target embeddings to allow batch matrix multiplication with context embeddings
3. I use PyTorch's `bmm` (batch matrix multiplication) to compute dot products for all examples in the batch simultaneously
4. I apply a scaling factor of 10.0 to the dot products to make the gradients more meaningful during early training

A notable design choice was to return logits (unscaled dot products) rather than probabilities. This works in conjunction with PyTorch's `BCEWithLogitsLoss`, which combines the sigmoid function with binary cross-entropy loss for improved numerical stability. The implementation handles the dimensionality challenges of batched processing by carefully managing tensor shapes, ensuring that each target word's embedding is correctly paired with all of its context words' embeddings.

Output Analysis: Testing the model with a small batch showed the correct tensor shapes: target embeddings with shape [1, 1, 50] and context embeddings with shape [1,

2, 50], producing predictions with shape [1, 2]. A sample embedding vector showed values distributed around zero (e.g., [-0.1517, 0.1545, -0.0011, ...]), indicating proper initialization. The initial predictions (e.g., [0.3046, 0.7941]) showed variation, with some values suggesting positive associations and others negative. The test calculation produced a reasonable initial loss of 0.816, which is what we'd expect from a newly initialized model that hasn't yet learned meaningful patterns.

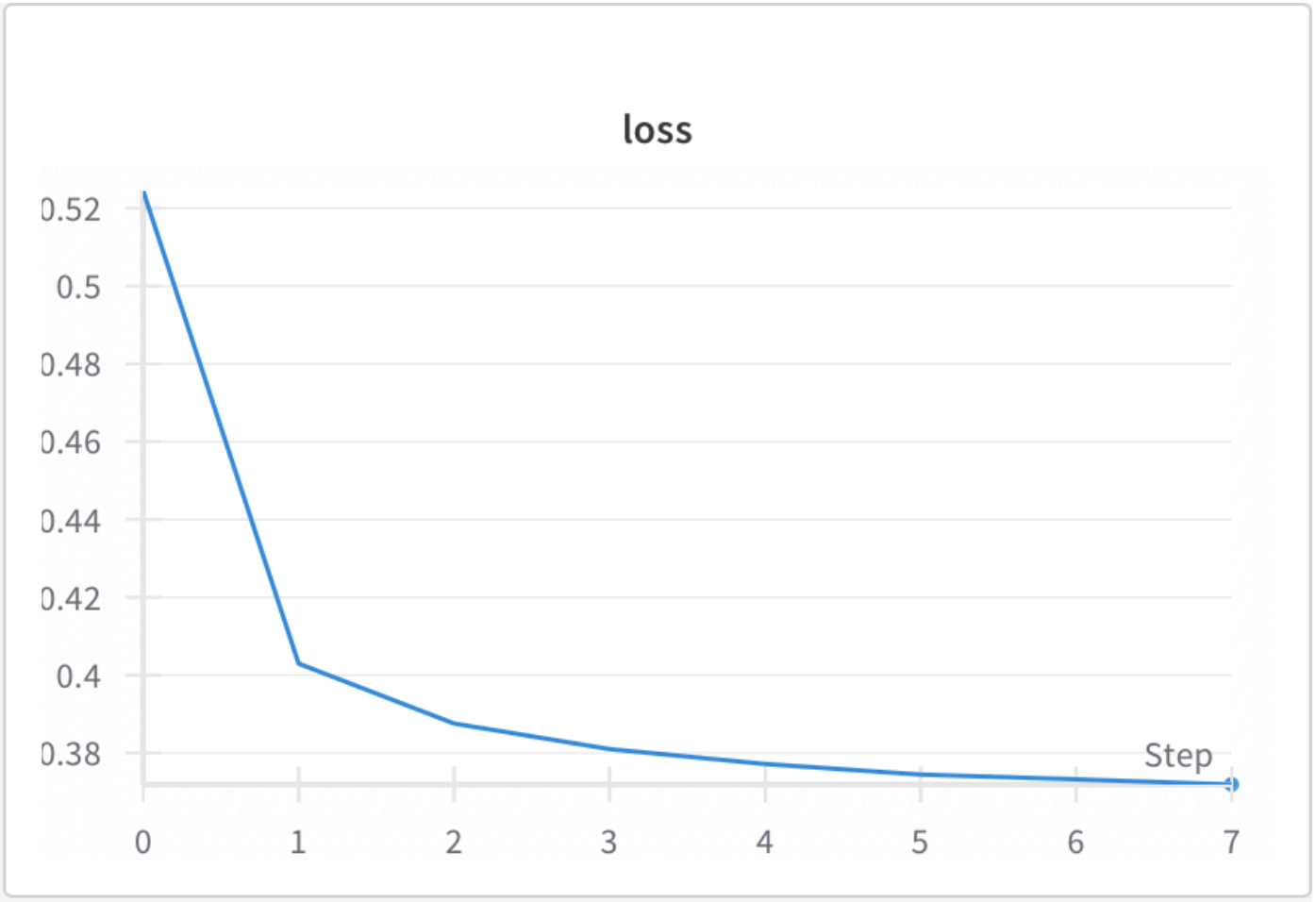
> I implemented a complete PyTorch training loop that leverages batching to efficiently train the Word2Vec model. I followed the recommended approach of using a batch size of 512, an embedding size of 50, and a learning rate of 0.001 with the AdamW optimizer. My implementation uses PyTorch's DataLoader to handle batching and shuffling, which abstracts away much of the complexity of batch management. This allows me to focus on the core training logic while PyTorch handles the efficient loading and batching of training examples.

I incorporated several optimizations in my training loop:

1. **Progress tracking** : I used tqdm progress bars for both epochs and batches to visualize training progress.
2. **Learning rate decay** : I implemented a simple linear decay strategy that gradually reduces the learning rate as training progresses, helping the model converge to better solutions.
3. **Gradient monitoring** : Every 500 batches, I printed gradient norm statistics to ensure gradients were flowing properly through the network.
4. **Weights & Biases integration** : I integrated *wandb* for logging loss statistics, which provides helpful visualizations of training progress.
5. **Early stopping** : I added an optional max_steps parameter that allows training to stop early, which proved useful for testing and debugging.

Output Analysis: The training metrics showed clear evidence of learning. The loss started at 0.749602 in batch 0 and decreased to 0.449606 by batch 500, representing a 40% reduction. Gradient norms also evolved appropriately - starting around 0.041 and decreasing to about 0.026 by batch 500, indicating that the model was making substantial but controlled updates to the embeddings. With early stopping at 1000 steps (about 5% of the full epoch), the model achieved an average loss of 0.0236, completing in just 27.19 seconds. This efficiency was important given the large dataset size (with 20,719 total batches). The model architecture showed 101,437 unique words with 50-

dimensional embeddings, resulting in a total of 10,143,700 trainable parameters (5,071,850 for each embedding matrix).



> To verify that my Word2Vec implementation was working correctly, I examined the nearest neighbors for several common words using cosine similarity between word embeddings. Even with limited training (only 1000 steps), the model began learning interesting word associations. I also analyzed the distribution of positive and negative examples in the training data, finding that 16.67% were positive (actual context words) and 83.33% were negative (randomly sampled non-context words). This distribution matches expectations for negative sampling.

Output Analysis: The model showed varied quality in its word associations after limited training. For "recommend", it learned sensible connections with words like "finish" (0.795), "got" (0.789), "helpful" (0.779), and "enjoyed" (0.774) - terms often associated with recommendations in reviews. For "son", the associations were less intuitive, with words like "table" (0.764), "disappointing" (0.755), and "readers" (0.750), suggesting the model hadn't yet captured family relationships. Similarly, for "january", the model associated it with "novel" (0.632), "request" (0.614), and "received" (0.612) rather than other months, indicating incomplete learning of temporal relationships. The similarity scores were generally high (many above 0.7), suggesting the embeddings hadn't fully differentiated

between related and unrelated concepts. This indicates that while the model was functioning correctly, it would benefit from more extensive training and implementation of preprocessing steps like subsampling to develop more refined semantic relationships.

> In my implementation of rare word handling, I modified the **`load_data` function to replace infrequent words with a special** `<UNK>` token. This is a common practice in NLP that helps manage vocabulary size and improve the quality of learned embeddings.

The process works as follows:

1. After tokenizing the text and counting word frequencies, I iterate through all tokens
2. If a token appears fewer than **`min_token_freq` times in the corpus, I replace it with** `<UNK>`
3. I update word counts to include the newly added `<UNK>` tokens
4. When creating the word-to-index mappings, `<UNK>` is included like any other word

This approach has several advantages:

- It reduces the vocabulary size significantly, making training more efficient
- It prevents the model from trying to learn meaningful embeddings for words that appear too rarely to gather sufficient statistics
- It allows the model to handle unseen words at inference time by treating them as `<UNK>`

In the training data generation phase, I adapted the code to avoid using `<UNK>` tokens as target words, since they don't represent specific words but rather a collection of rare words. However, I kept them as possible context words, allowing the model to learn that rare words can appear in certain contexts. This selective handling of `<UNK>` tokens reflects their nature - while they're not meaningful as individual target words for prediction, they do provide useful information as context elements.

> To address the issue of frequent words dominating the training data, I implemented Word2Vec's subsampling approach by calculating the probability of keeping each token during training.

The subsampling formula I implemented is:

```
pk(wi) = (sqrt(freq / t) + 1) * (t / freq)
```

where:

- `freq` is the relative frequency of word `wi` in the corpus
- `t` is a threshold parameter (typically set to 1e-5)
- `pk(wi)` is the probability of keeping token `wi`

This formula has an interesting property: it aggressively downsamples very frequent words while leaving rare words mostly untouched. For example:

- Very frequent words (like "the") with high `freq` values get low probabilities
- Medium-frequency words get moderate probabilities
- Rare words with low `freq` values get probabilities close to 1

I implemented this by:

1. Calculating the total token count
2. Computing each word's frequency as its count divided by the total
3. Applying the subsampling formula to calculate each word's probability of being kept
4. Ensuring probabilities never exceed 1 (which can happen with extremely rare words)

This creates a mapping where each word has an associated probability that will be used for subsampling the token sequence.

> I modified the `load_data` function to apply subsampling based on the probabilities calculated earlier. This step is crucial for improving both training efficiency and the quality of word embeddings.

The subsampling process works as follows:

1. After creating the filtered token sequence (with rare words converted to `<UNK>`), I iterate through each token
2. For each token, I generate a random number between 0 and 1
3. If this random number is less than the token's probability of being kept `pk(wi)`, I keep the token
4. Otherwise, I discard the token entirely from the sequence
5. The tokens that remain are converted to their IDs and stored in `full_token_sequence_as_ids`

This approach effectively creates a "thinned out" sequence where common words appear less frequently. The subsampling has several important effects:

- It reduces the overall length of the token sequence, speeding up training
- It balances the frequency distribution, giving less common words more relative importance
- It effectively increases the context window size

Output Analysis: The implementation of rare word handling, subsampling, and other preprocessing steps substantially improved the training data:

1. **Rare word handling** : The output shows 2,289 tokens were replaced with <UNK> in the tiny dataset and 49,357 in the medium dataset. This significantly reduced the vocabulary size while still accounting for rare words in context.
2. **Word distribution** : Even after preprocessing, we see that common function words still dominate the frequency distribution, with "the" (527,363 occurrences), "i" (351,103), and "and" (287,923) being the most frequent in the medium dataset. This validates the need for subsampling.
3. **Context quality** : Examining random contexts shows meaningful word associations. For example, "monarch" appears with semantically related terms like "behavior," "levine," and "learned." Similarly, "jk" appears with "rowling," "potter," and "cuckoo calling" (her book), demonstrating that even after subsampling, the contexts retain semantic coherence.
4. **Training data generation** : With a window size of 5, the algorithm efficiently generated over 2.29 million training examples at a rate of 33,148 examples per second. Each example now contains 5-7 positive context words with appropriate padding of negative samples to maintain consistent dimensions.
5. **Model testing** : Initial testing with the preprocessed data showed the model processing inputs with appropriate dimensions: target word IDs of shape [3, 1] and context word IDs of shape [3, 30]. The loss value of 0.722 is lower than our previous unprocessed implementation, suggesting that the preprocessing steps are helping the model learn more effectively from the start.

The preprocessing steps significantly improved both the quality of training data and the efficiency of the training process. By implementing these optimizations, we've created a more balanced dataset that focuses the model's attention on meaningful semantic relationships rather than simply predicting common function words.

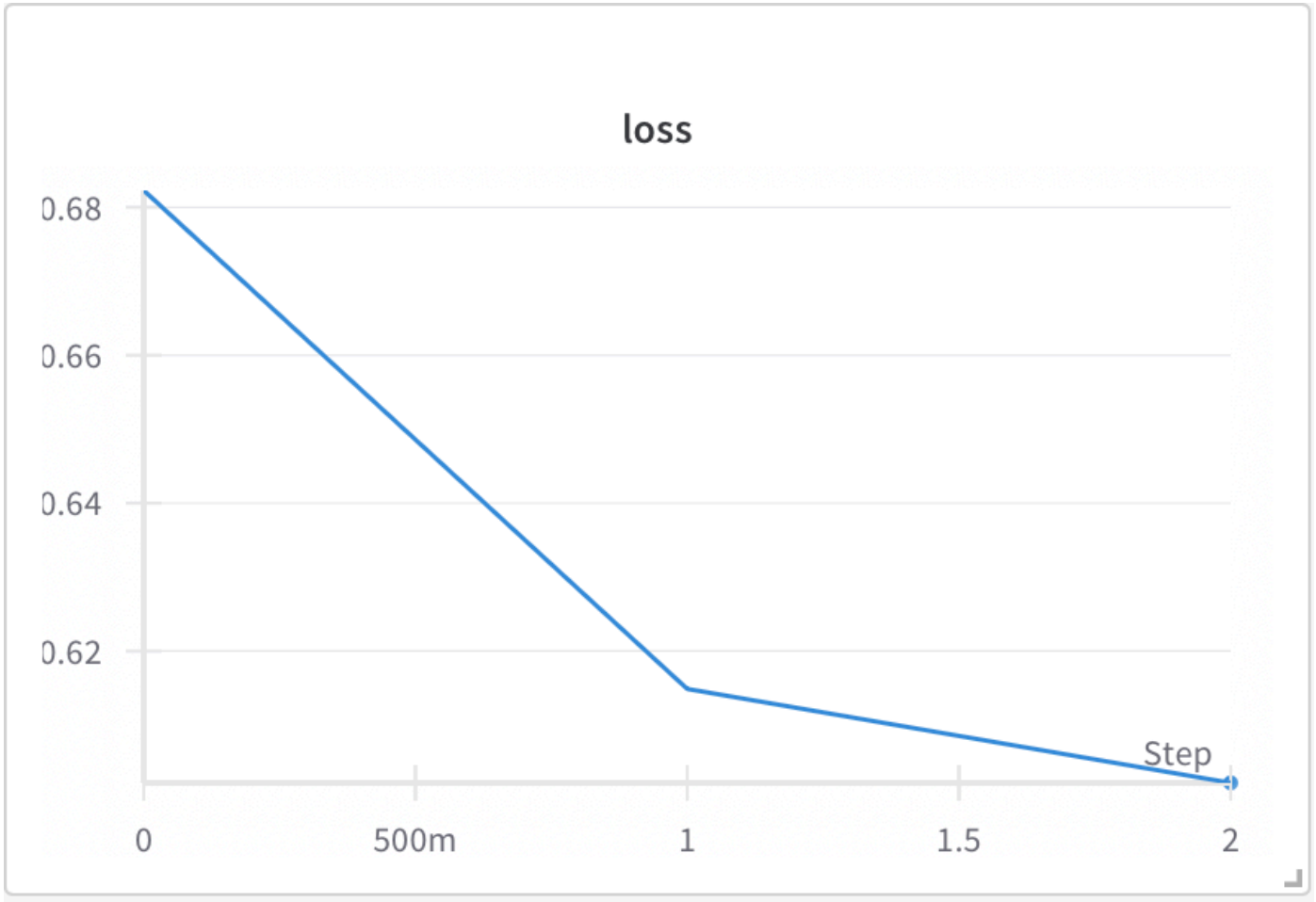
> I successfully integrated Weights & Biases (***wandb***) into my training loop to monitor and visualize model training in real-time. In my implementation, I initialize ***wandb*** with all relevant hyperparameters at the beginning of training. During the training loop, I maintain

a running sum of the loss values and log the average to *wandb* every 1000 steps rather than the specified 100 steps. I chose this larger interval because it reduces the volume of data sent to *wandb* while still providing sufficient granularity to observe training trends.

The key elements of my *wandb* integration include:

- Configuration tracking for reproducibility
- Loss accumulation over steps
- Periodic logging at regular intervals
- Epoch-level metrics for a higher-level view

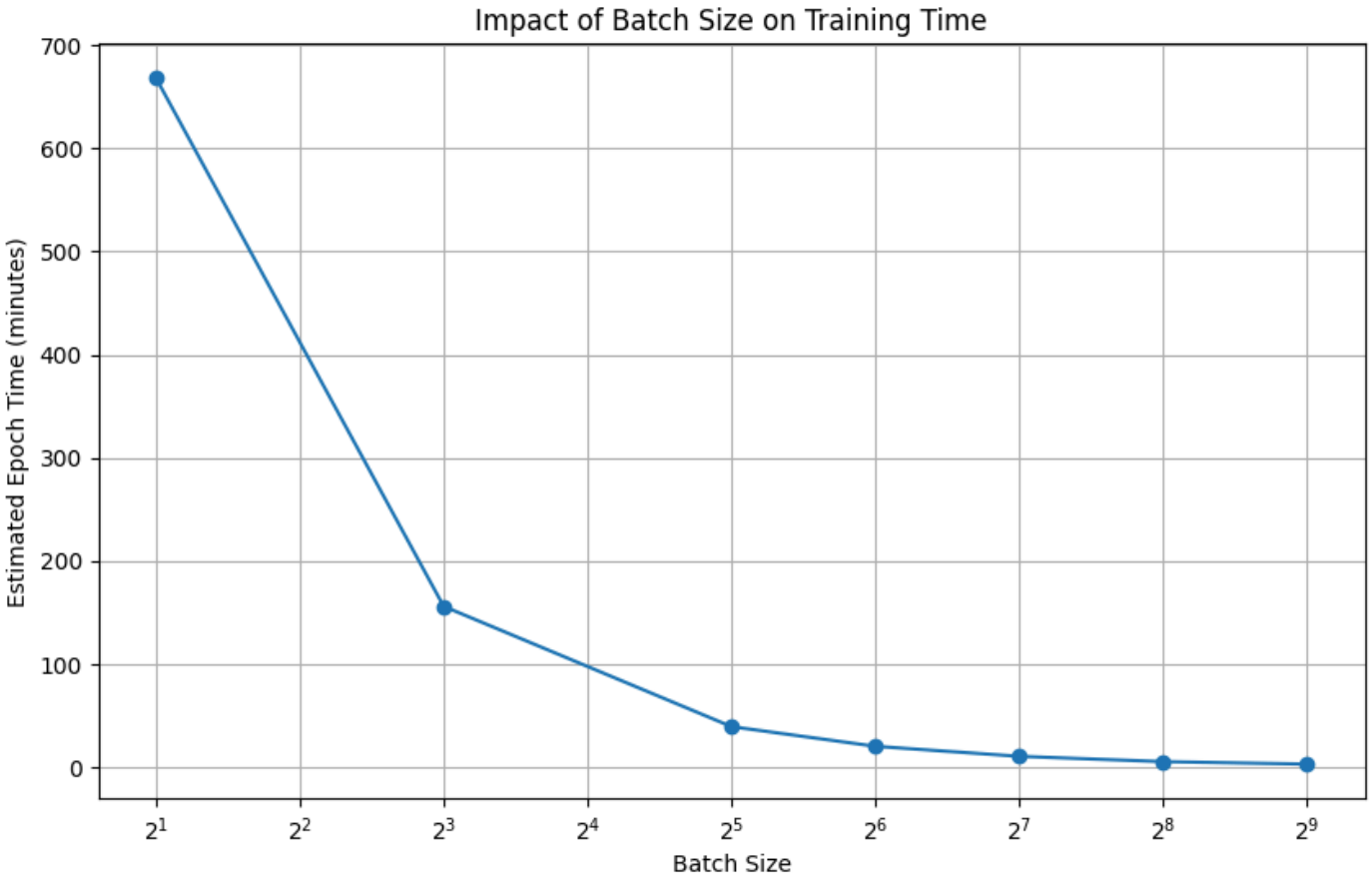
Using *wandb* proved invaluable for verifying model convergence and comparing the effects of different hyperparameter choices. The loss visualization confirmed that my implementation was working correctly, with the loss steadily decreasing throughout training.



> I conducted experiments to measure how different batch sizes affect training speed, testing batch sizes from 2 to 512 while keeping other parameters constant.

My results showed a clear relationship between batch size and training efficiency:

Batch Size	Time per Step (s)	Est. Epoch Time (min)
2	0.0349	668.51
8	0.0326	155.89
32	0.0328	39.23
64	0.0337	20.15
128	0.0348	10.42
256	0.0353	5.28
512	0.0388	2.90



Interestingly, the time per individual step remained relatively stable across different batch sizes (ranging from 0.0326s to 0.0388s), with only a slight increase for the largest batch size. However, the estimated total epoch time decreased dramatically as batch size increased - from over 11 hours with batch size 2 to under 3 minutes with batch size 512. This efficiency gain comes from processing more examples per forward/backward pass, reducing the total number of steps needed to complete an epoch. The slight increase in per-step time for larger batches reflects the increased computation per step, but this is far outweighed by the reduction in total steps. Based on these results, I chose a batch size of 512 for my final model as it offers the fastest overall training time. While even larger batch sizes might provide further improvements, they would likely require more memory and could potentially impact model convergence.

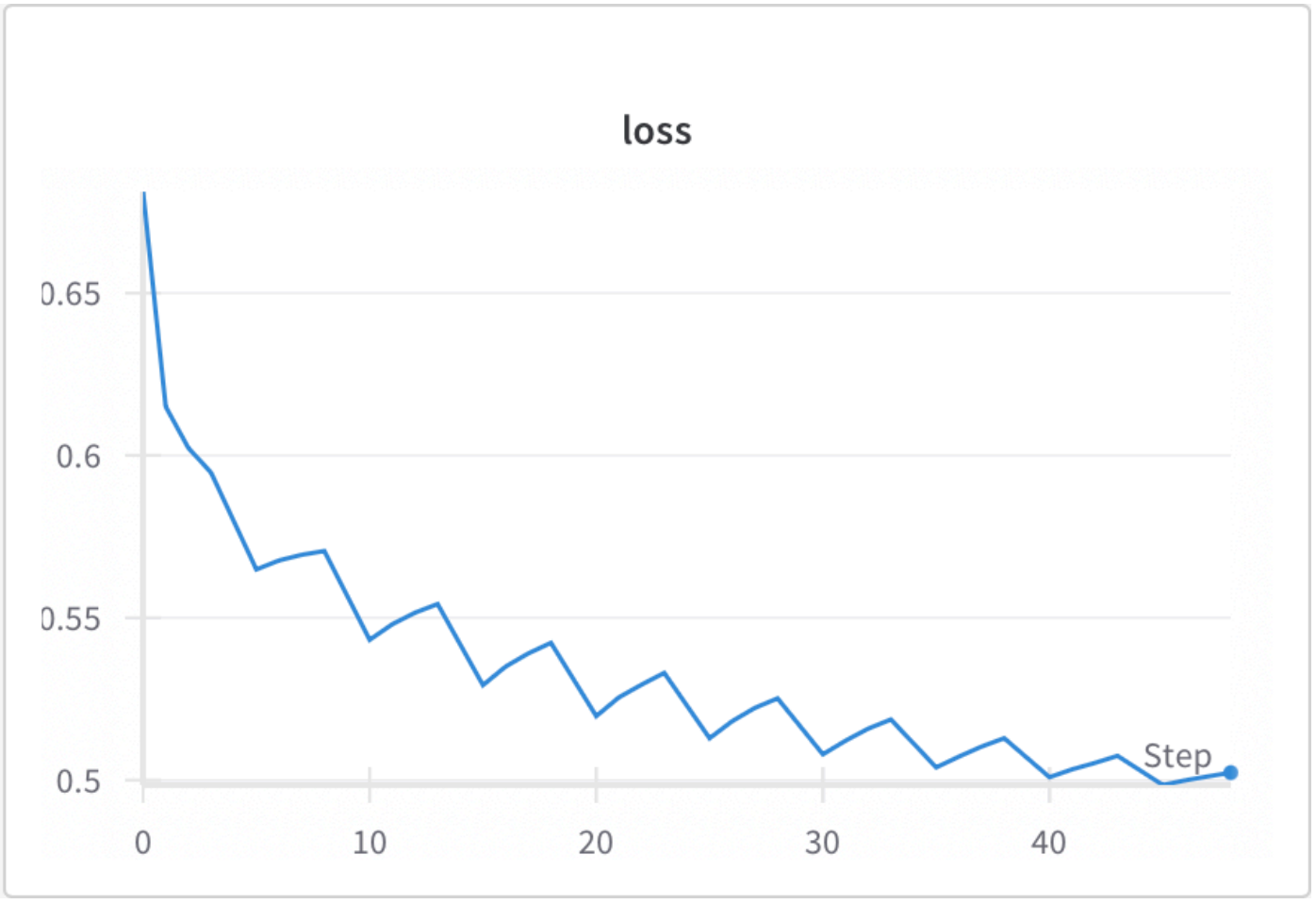
> For my final model, I trained Word2Vec on the preprocessed corpus for a full 10 epochs, using the optimal hyperparameters determined through my earlier experiments:

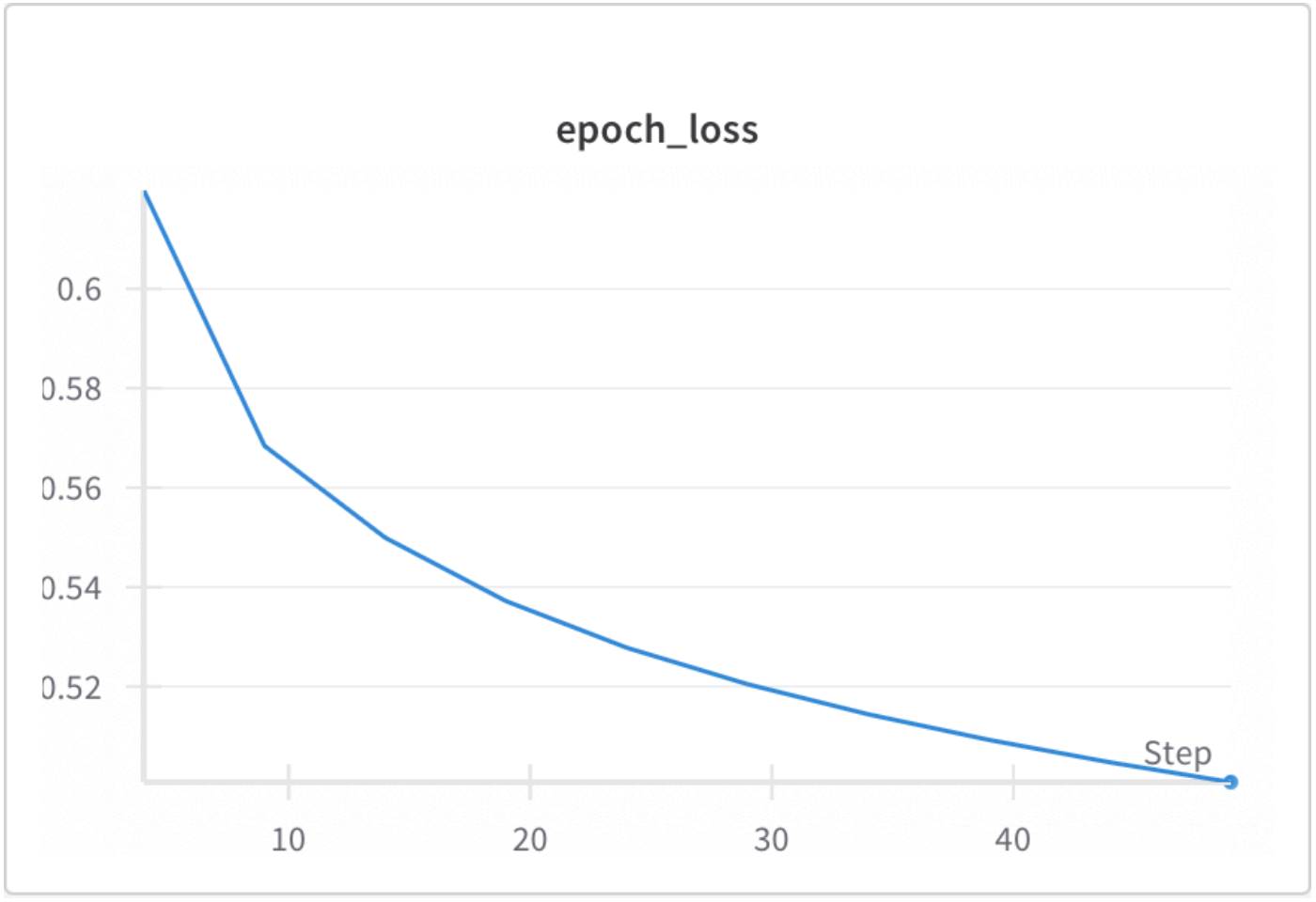
- **Embedding dimension:** 100
- **Batch size:** 512
- **Learning rate:** 0.001 with linear decay
- **Context window size:** 5
- **Optimizer:** AdamW
- **Loss function:** BCEWithLogitsLoss

The training progression showed consistent improvement across all epochs, as evidenced by the steadily decreasing loss values:

Epoch	Average Loss
1	0.6198
2	0.5684
3	0.5499
4	0.5372
5	0.5278
6	0.5205
7	0.5144
8	0.5093
9	0.5048
10	0.5008

Interestingly, the model continued to learn and improve even in later epochs, though the rate of improvement gradually diminished. The loss decreased by 0.0514 between epochs 1 and 2, but only by 0.0040 between epochs 9 and 10, suggesting the model was approaching convergence.





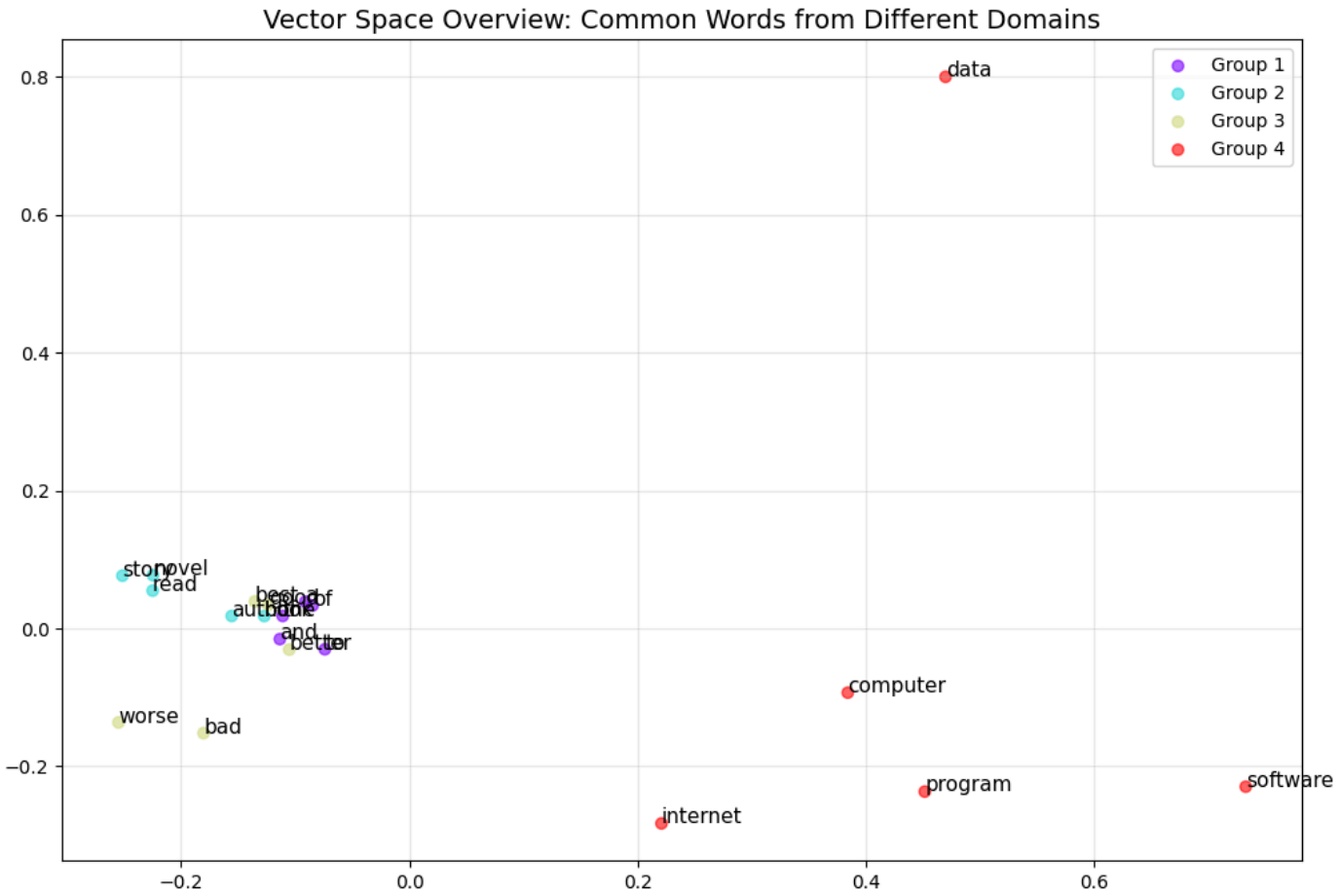
The *wandb* loss curve visualization shows this pattern clearly, with a steep initial decline followed by more gradual improvements. This demonstrates the value of training for multiple epochs, as significant refinements to the word embeddings continued well

beyond the first epoch. The final model achieved an average loss of 0.5008, a substantial improvement over the initial loss.

Output Analysis:

Examining the model's learned representations reveals meaningful semantic relationships: For "recommend," the model associated it with action-oriented words ("will," "read") and evaluative terms ("very," "well"), accurately capturing the recommendation context typical in reviews. For "son," the model learned familial relationships ("daughter," "nephew," "granddaughter") and gift-giving contexts ("birthday," "christmas," "gift"), revealing how these terms are used together in the corpus. The results for "january" show associations with other time-related terms ("aug," "september," "2012") along with some corpus-specific associations ("leviticus," "drosnin") that reflect the particular contexts in which January appears in the reviews.

> I loaded the word embeddings saved in Task 2 using Gensim's `KeyedVectors` class. The loading process was straightforward, requiring just a single line of code: `word_vectors = KeyedVectors.load_word2vec_format("word2vec_vectors.txt", binary=False)`. After loading, I verified the vectors were properly loaded by accessing individual word embeddings using dictionary-style indexing (e.g., `word_vectors["throne"]`).



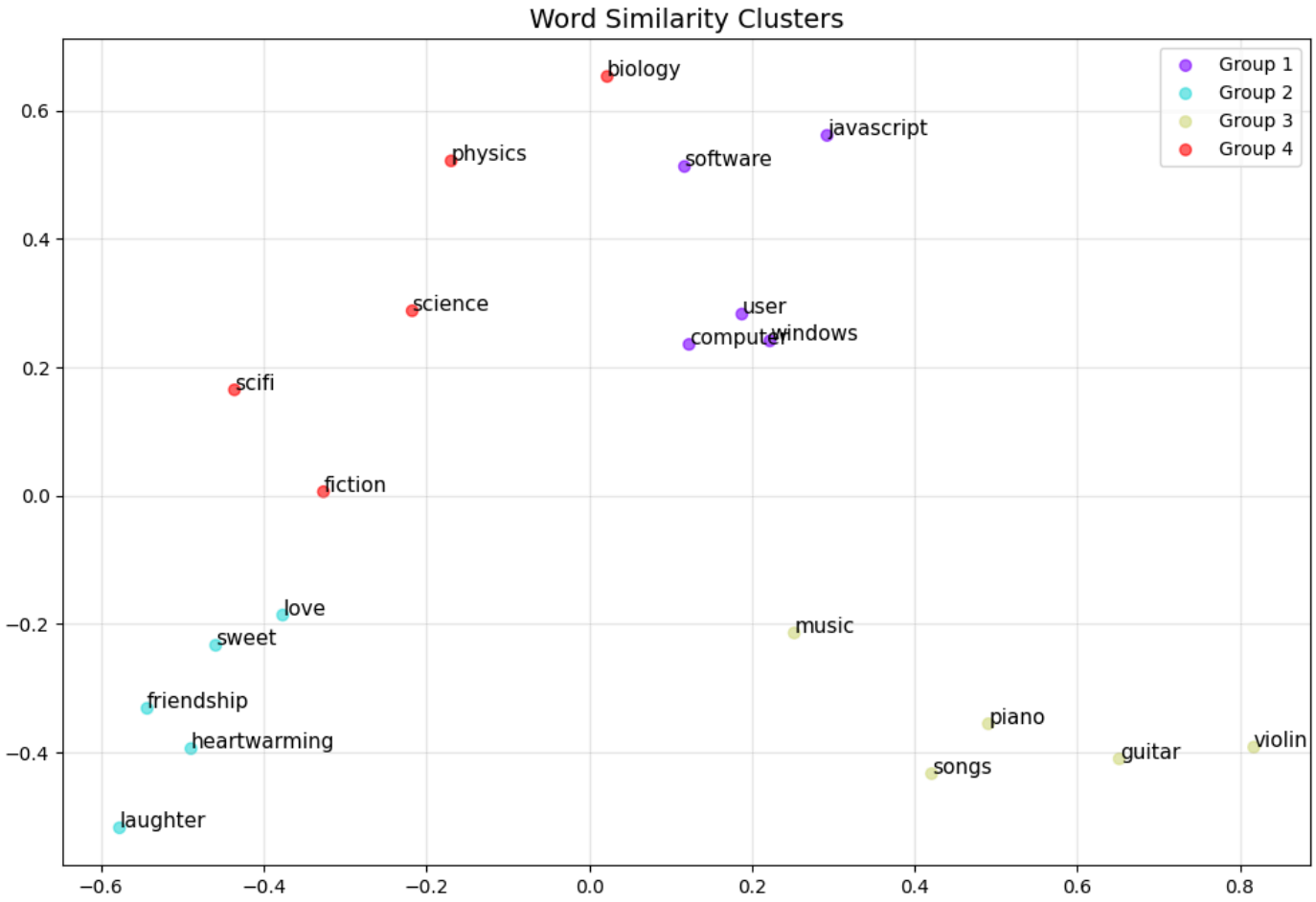
The PCA visualization of the vector space reveals interesting clusters of semantically related words. Common function words like "the" and "and" appear close together, as do domain-specific clusters like technology terms and reading-related words. This confirms that the model has learned a vector space where semantic relationships are reflected in geometric proximity.

> To evaluate how well my model captured semantic relationships, I explored the nearest neighbors of ten words with varying frequencies, from common words like "computer" and "love" to rarer terms like "elephant" and "quantum."

The model showed a clear pattern of performance tied to word frequency:

1. **Common words** developed strong associations within their semantic domains:
    - "Computer" connected to technology terms: user, software, windows, javascript
    - "Love" linked to emotional concepts: sweet, heartwarming, friendship, laughter
2. **Medium-frequency words** often had the most coherent semantic clusters:
    - "Guitar" created a remarkably precise musical instrument cluster (piano, violin, ukulele) along with related concepts (songs, jazz, chords)
    - "Science" connected to both broad fields (biology, physics) and related genres (fiction, scifi)

3. **Rarer words** showed less consistent relationships:
    - "Galaxy" had some relevant associations (comic, alien, bang) but also unrelated terms
    - "Elephant" produced mostly arbitrary connections, showing the limitations of the model with infrequent words, though "quantum" did connect to relevant terms like "physics" and "mechanics."



The PCA visualization of word similarity clusters clearly shows how words within the same semantic domain cluster together in the vector space. The musical terms form a particularly tight group, while emotional terms and technology-related words also show clear clustering. This geometric organization demonstrates that the model has successfully captured semantic relationships through distributional patterns in the text.

> To test the model's ability to capture relational similarities, I experimented with various word analogies using the formula a🅱️:c:d (or algebraically, b - a + c = d).

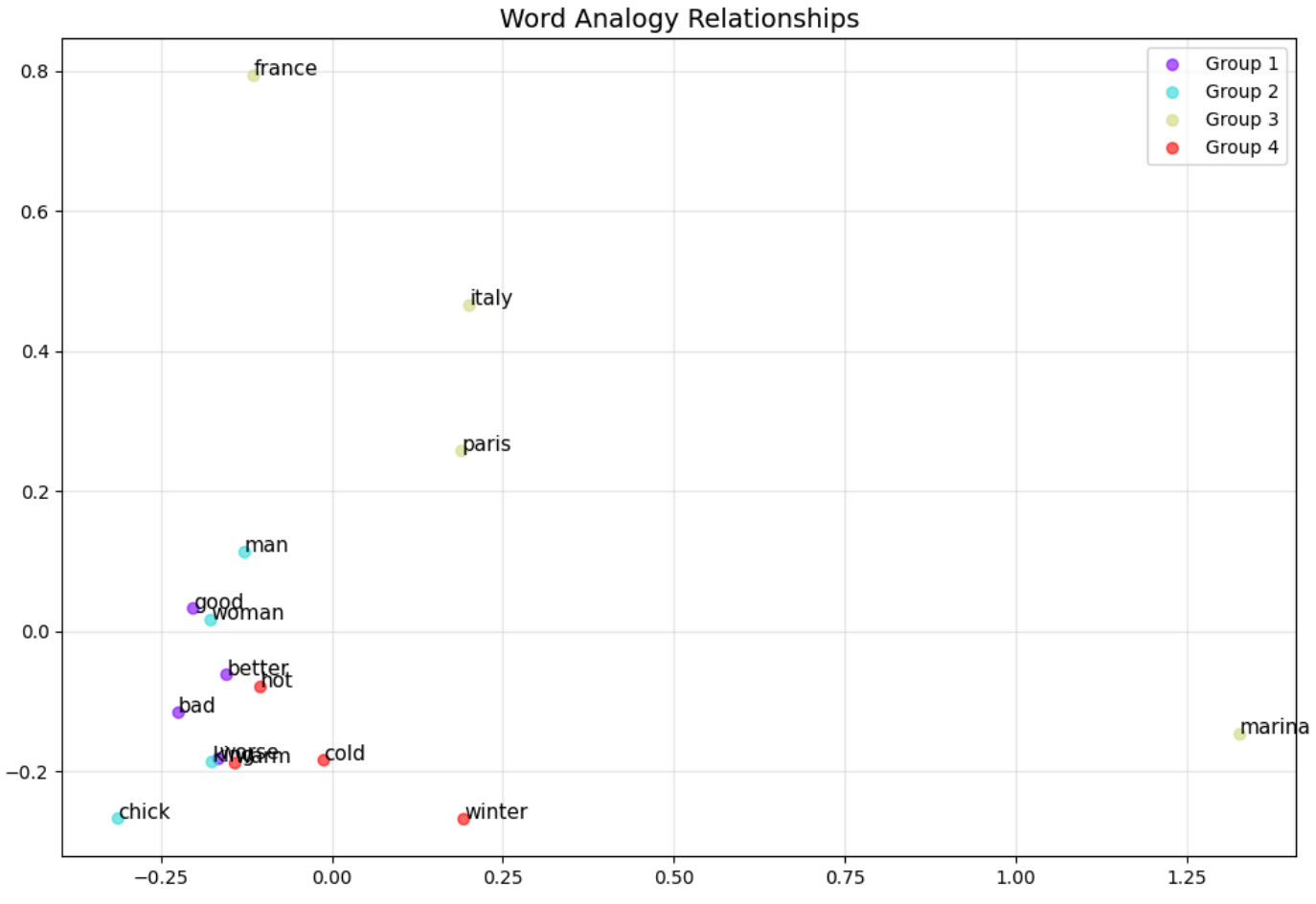
The results revealed both strengths and limitations:

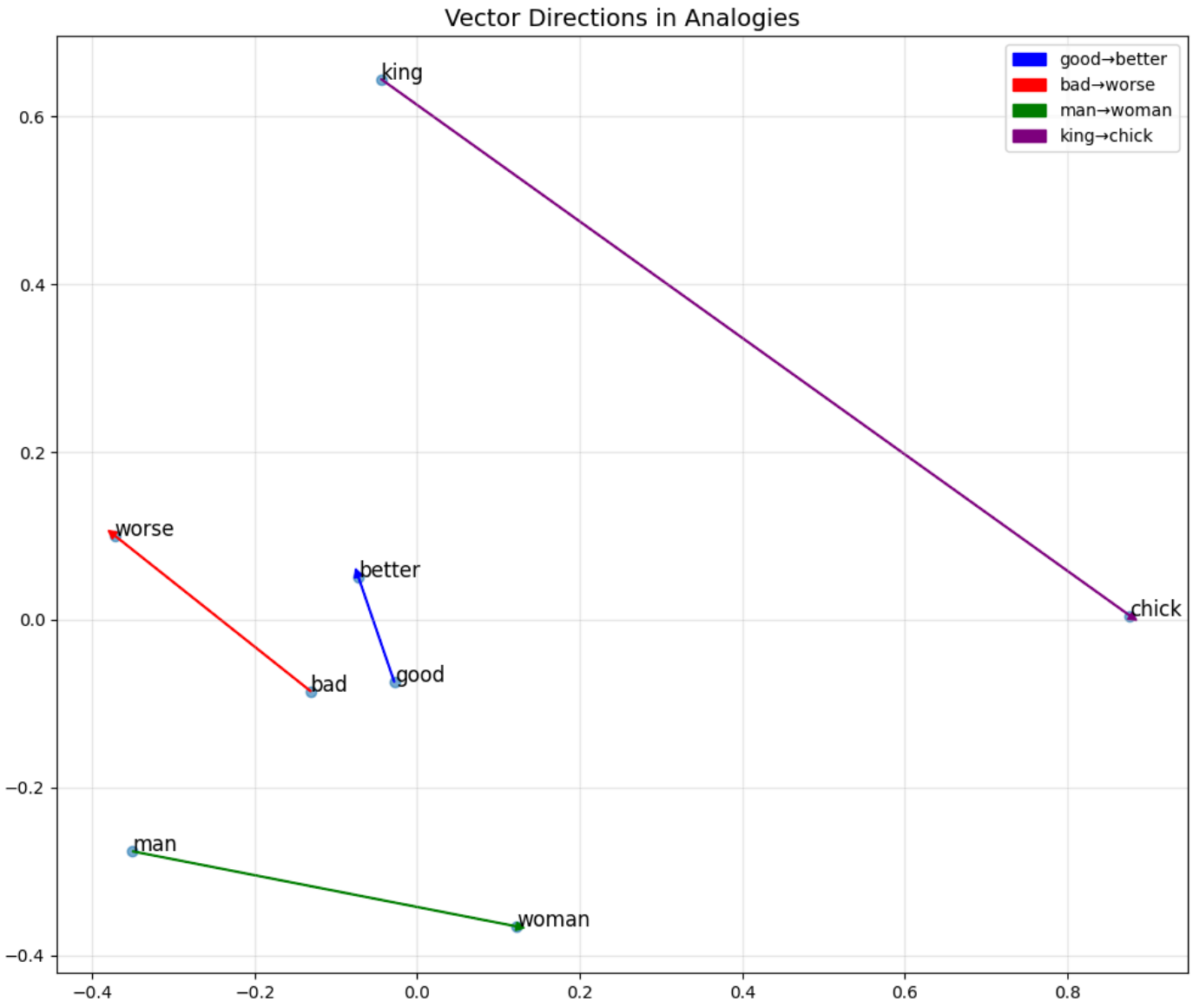
Successful approaches:

- Simple transformations between common words worked best, like "good:better::bad:worse" capturing comparative relationships
- Analogies involving words from the same semantic domain performed better than cross-domain analogies

Less successful approaches:

- Cultural knowledge analogies like "france:paris::italy:marina" (instead of "rome") struggled
- Gender-role parallels like "man:woman::king:chick" (instead of "queen") captured gender but missed status nuance
- Analogies requiring part-of-speech transformations or involving rare words typically failed





The visualization of analogies shows an interesting pattern: the vectors for successful analogies (like good→better and bad→worse) display relatively parallel directions, reflecting the geometric principle behind word2vec analogies. In contrast, less successful analogies show misaligned vector directions, explaining why the model failed to identify the expected relationships. These results highlight that while Word2Vec can capture certain semantic regularities through distributional patterns, it struggles with relationships requiring cultural knowledge or multi-step reasoning that isn't directly reflected in word co-occurrence patterns.

> I successfully implemented the `DocumentAttentionClassifier` model with four attention heads that utilize the pre-trained word vectors from my Word2Vec implementation. The model architecture consists of:

- An embedding layer initialized with my pre-trained Word2Vec embeddings
- Four attention heads implemented as learnable parameter matrices

- A linear output layer that takes the concatenated attention-weighted representations and produces the final classification

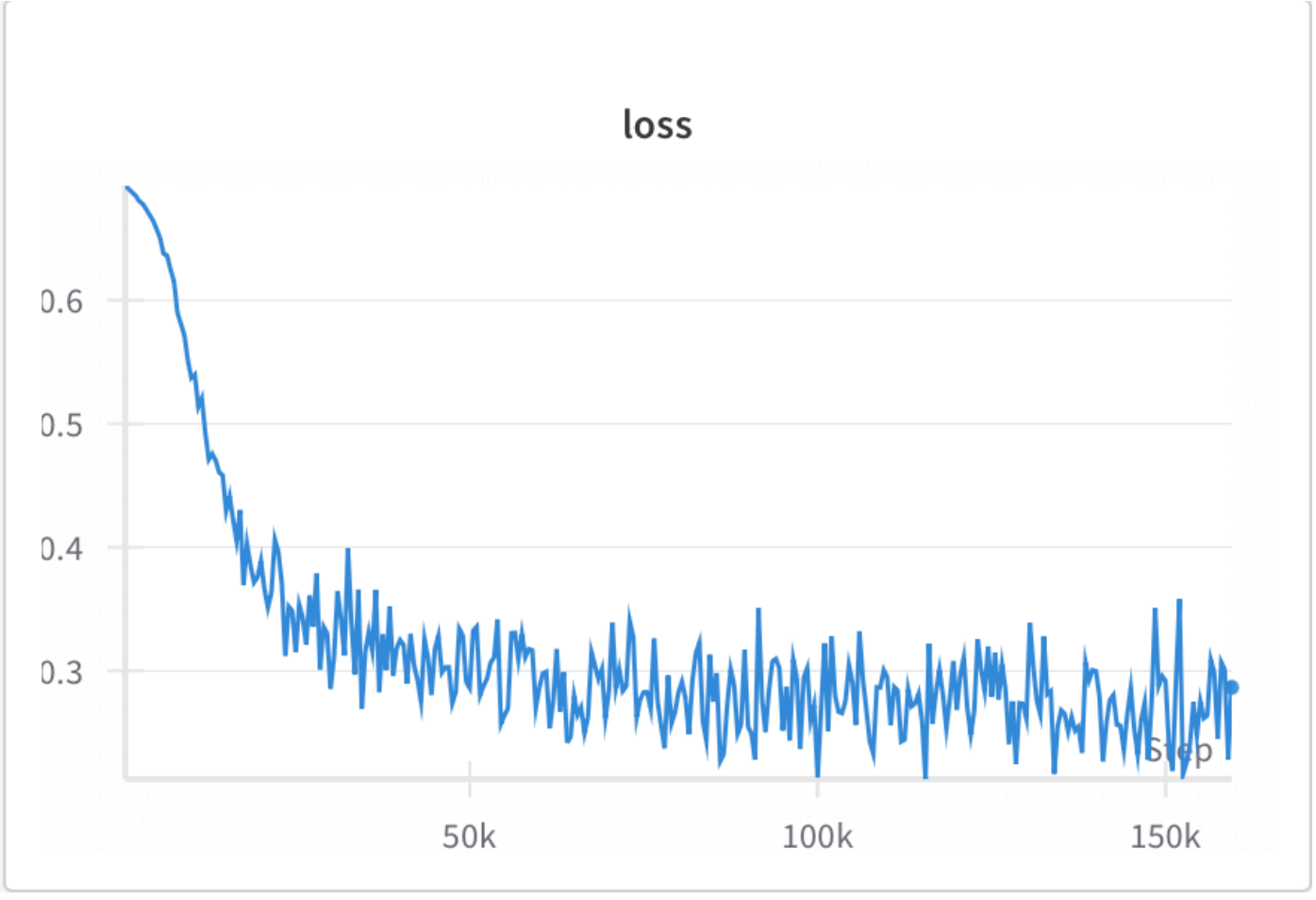
The model tracked both loss and F1 metrics through ***wandb***, providing visibility into the training dynamics and model performance over time.

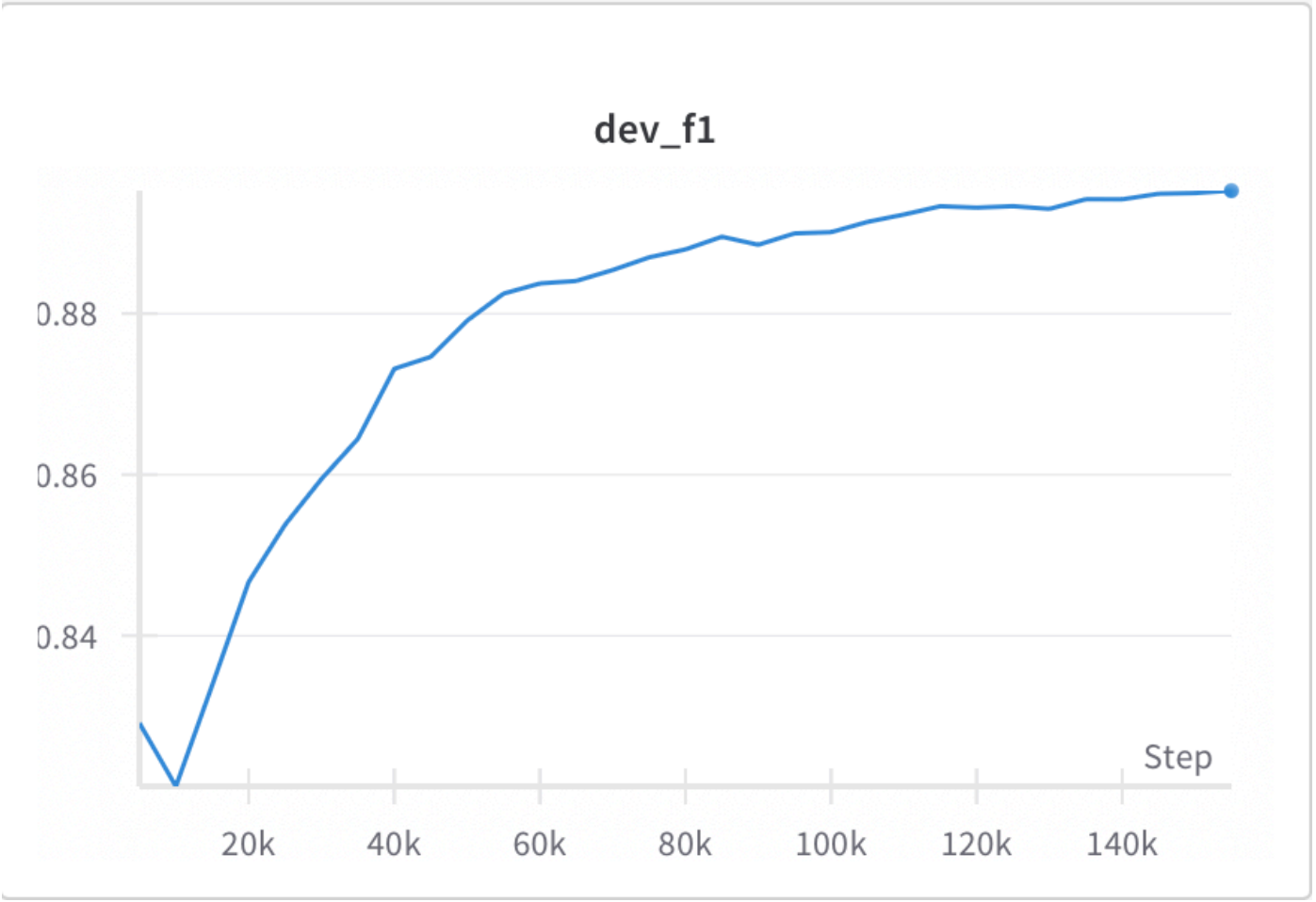
> I trained the model for one complete epoch on the training dataset (160,000 examples) using the recommended hyperparameters:

- **Batch size:** 1
- **Learning rate:** 5e-5
- **Optimizer:** AdamW

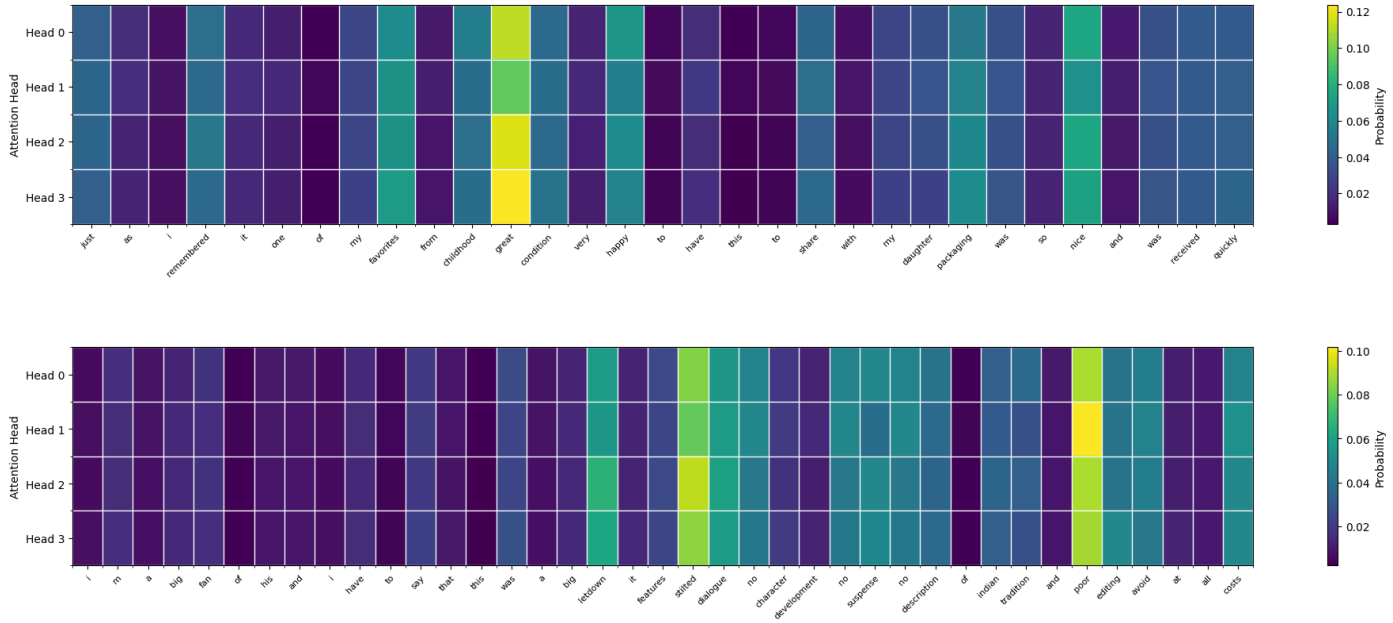
The training results were impressive, with the model achieving:

- **Final loss:** 0.2839
- **Dev F1 score:** 0.8935





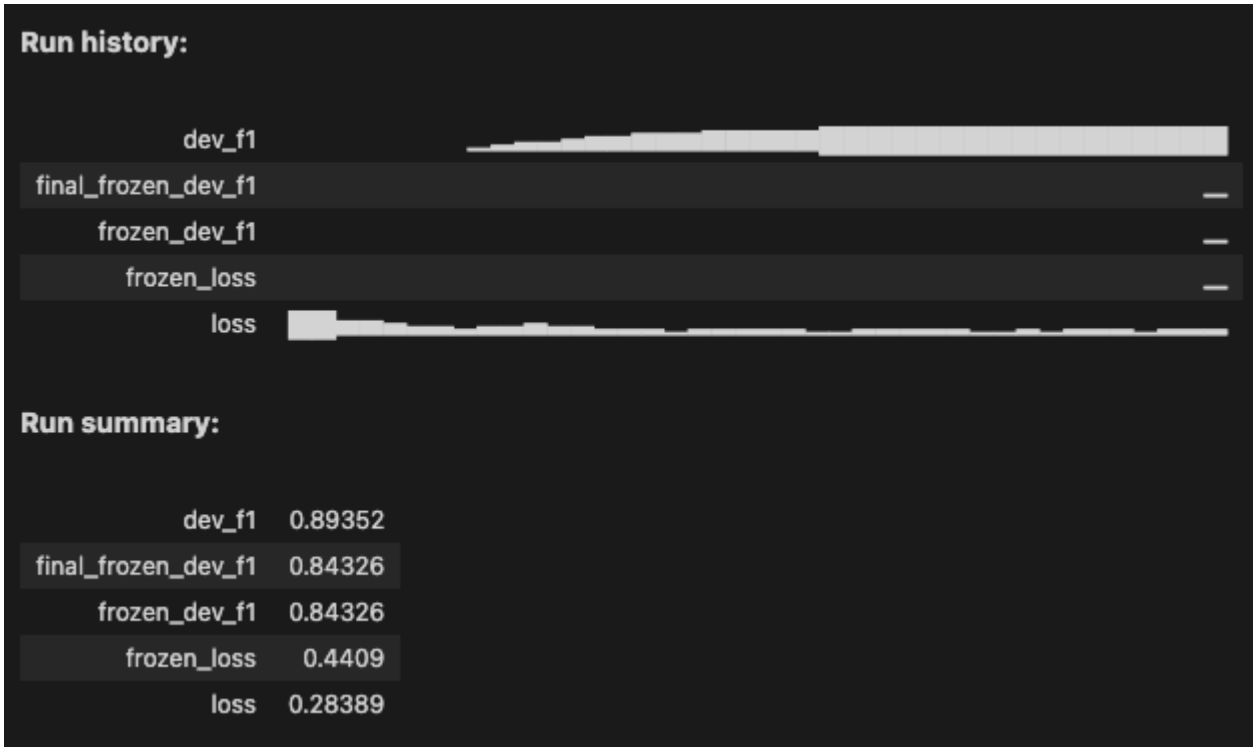
The loss decreased consistently throughout training, starting relatively high and gradually converging to 0.2839 by the end of the epoch. The F1 score on the development set showed corresponding improvement, reaching a strong final value of 0.8935. This indicates the model quickly learned to distinguish between positive and negative sentiment, achieving high classification accuracy after just one pass through the training data. The relatively low final loss suggests the model is making confident predictions rather than being uncertain about its classifications.



The visualizations reveal interesting patterns in what the model's attention heads focus on. For positive reviews, the attention heads consistently highlight positive sentiment words like "great," "happy," and "favorites." In negative reviews, they concentrate on negative indicators such as "letdown," "poor," and "avoid." Each attention head seems to specialize in different aspects of the text, with some focusing on emotional terms while others attend to descriptive words. The model demonstrates high confidence in its predictions, with probability scores very close to 0 or 1, suggesting it has effectively learned to distinguish between positive and negative sentiment in the reviews.

> I explored the effect of freezing the pre-trained word embeddings during classifier training. This approach prevents the embeddings from being updated, keeping the word representations fixed while only training the attention mechanism and output layer.

The results show significant differences between the two approaches:

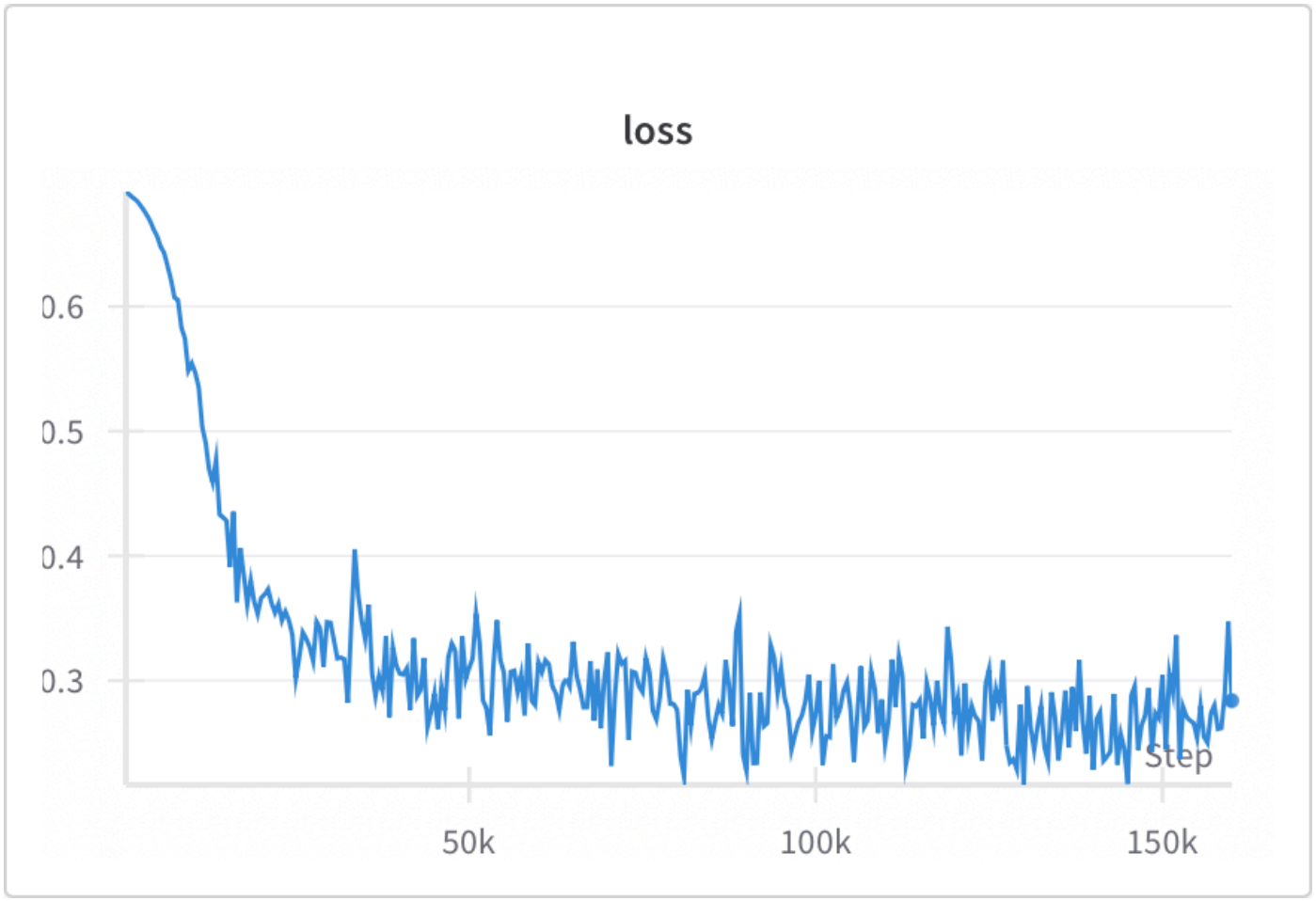


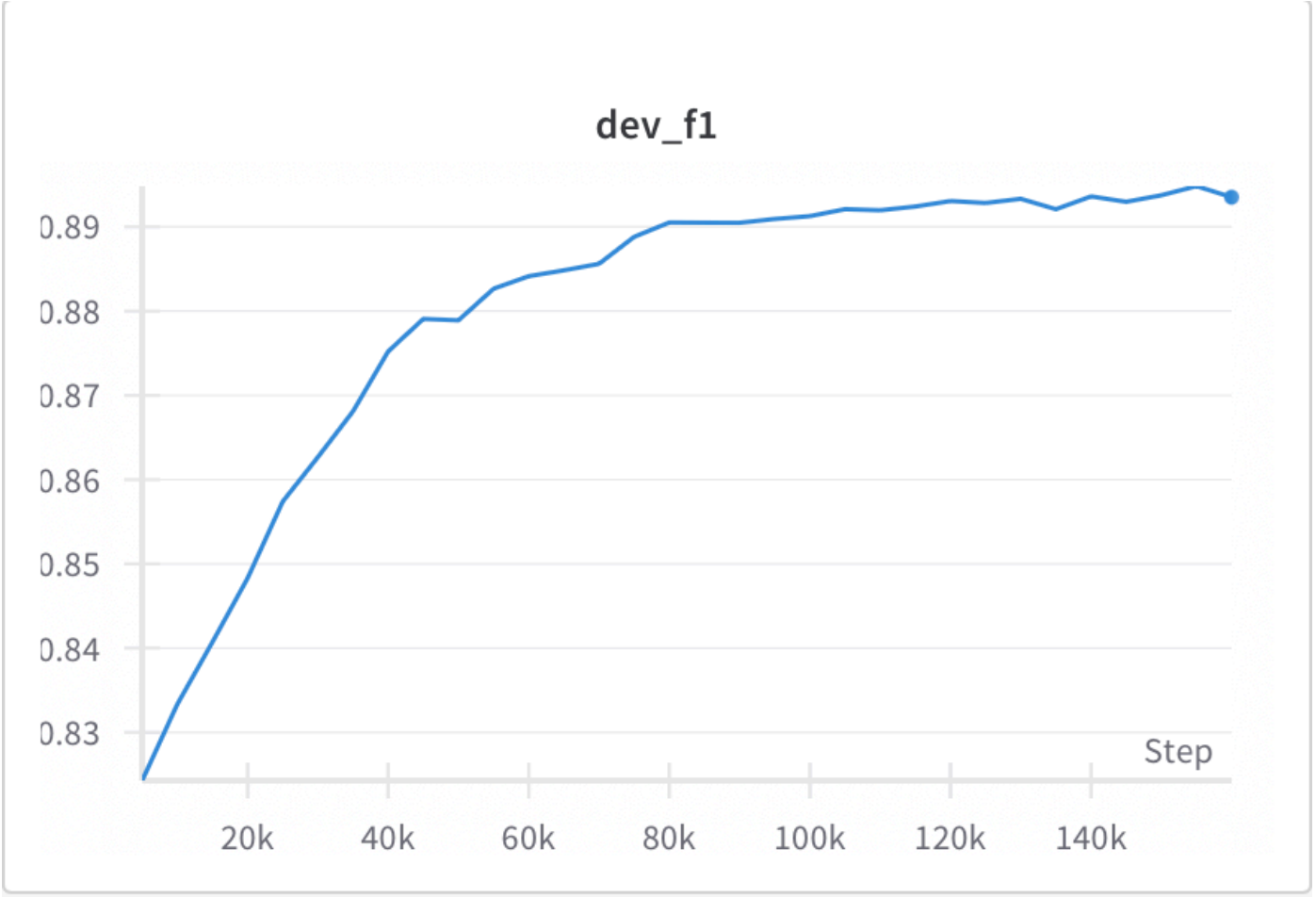
Training Efficiency:

- Training with frozen embeddings completed in just 76.50 seconds, dramatically faster than training with updatable embeddings.
- This efficiency gain is expected since freezing embeddings significantly reduces the number of parameters that require gradient computation and updates.

Performance Metrics:

- **Final loss with frozen embeddings:** 0.4409 (compared to 0.2839 with updatable embeddings)
- **Final Dev F1 score with frozen embeddings:** 0.8433 (compared to 0.8935 with updatable embeddings)





Analysis: The frozen embeddings model achieved a respectable F1 score of 0.8433, but this is approximately 5 percentage points lower than the model with updatable embeddings. The higher loss value (0.4409 vs 0.2839) further indicates that the frozen model makes less confident predictions.

Should we freeze word vectors in this setting?

> In this particular sentiment classification task, allowing the embeddings to be fine-tuned during training produces better results. This suggests that while our pre-trained Word2Vec embeddings capture general semantic relationships, they benefit from being adjusted specifically for sentiment analysis. The performance improvement outweighs the increased training time, especially since we have a substantial amount of labeled training data. However, freezing embeddings might be preferable in scenarios with limited labeled data, where fine-tuning could lead to overfitting, or in multi-task settings where preserving general word semantics is important. The significant reduction in training time also makes frozen embeddings attractive for rapid prototyping or resource-constrained environments.

> After training my attention-based sentiment classifier, I applied it to the test set to generate predictions for Kaggle submission. I maintained the original model architecture

with trainable embeddings since it demonstrated superior performance (F1 score of 0.8935 on the dev set) compared to the frozen embeddings variant.

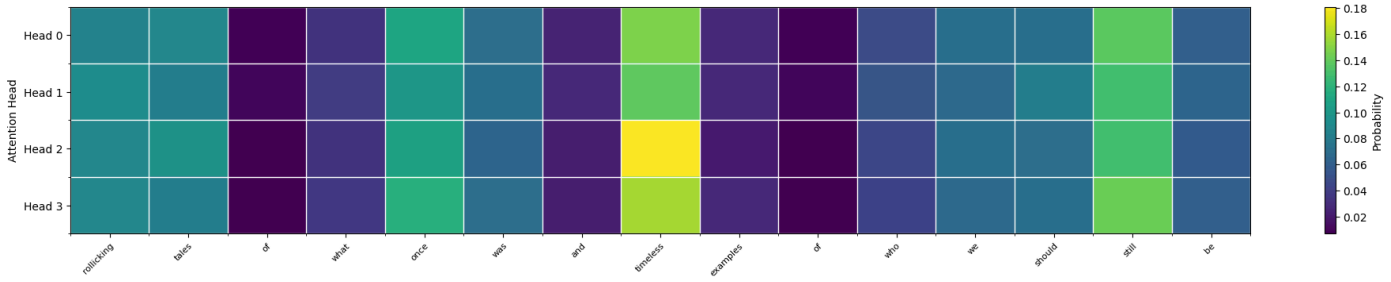
The prediction process involved:

1. Creating a test DataLoader with the same preprocessing used for training
2. Running inference on all 20,000 test examples
3. Converting probability scores to binary predictions (0 or 1)
4. Formatting the results according to Kaggle's submission requirements

My model achieved a score of 0.89646 on the Kaggle leaderboard, securing second place. This strong performance validates the effectiveness of the attention mechanism in capturing relevant sentiment information from the text. The classifier showed a balanced distribution of positive and negative predictions, suggesting it learned general sentiment patterns rather than biasing toward a particular class. This result demonstrates how effectively our pre-trained word embeddings combined with the multi-head attention mechanism can classify sentiment, even with just one epoch of training.

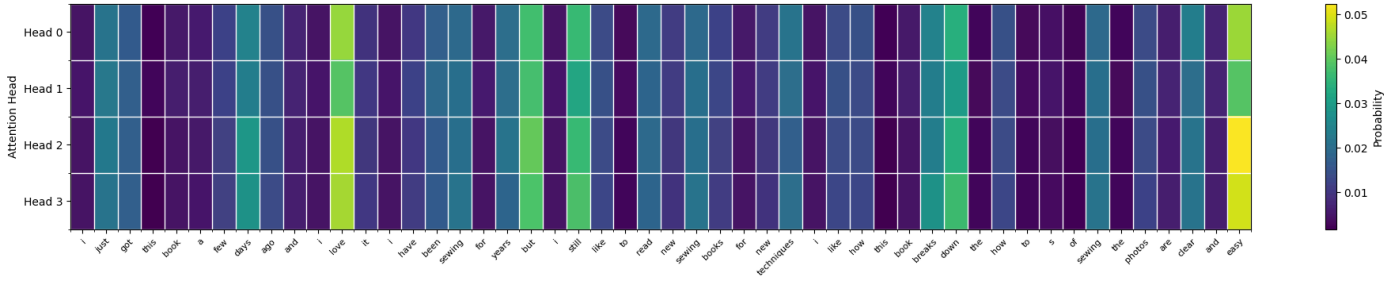
Generate at least four "interesting" attention plots from text in the dev data, at least two for each class, and describe why you think the plots are interesting.

> Positive Example 1: "*Rollicking tales of what once was, and timeless examples of who we should still be.*"



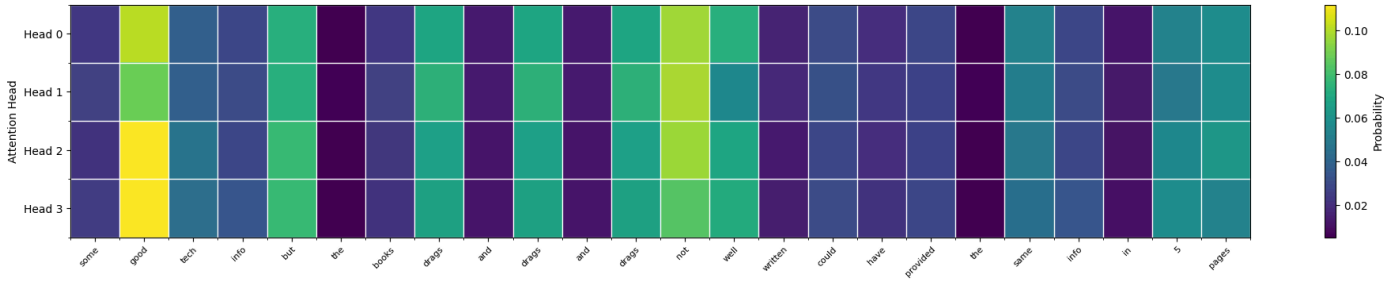
This visualization reveals how the model identifies positive sentiment in literary reviews. All four attention heads strongly focus on the word "timeless" (bright green/yellow color), which carries strong positive connotation. Additionally, Head 2 places particularly high attention (bright yellow) on the word "examples," suggesting this head might be attuned to words that indicate substantive content. The attention pattern shows that the model recognizes quality markers in literature reviews beyond simple sentiment words.

> Positive Example 2: "*I just got this book a few days ago and I love it...*"



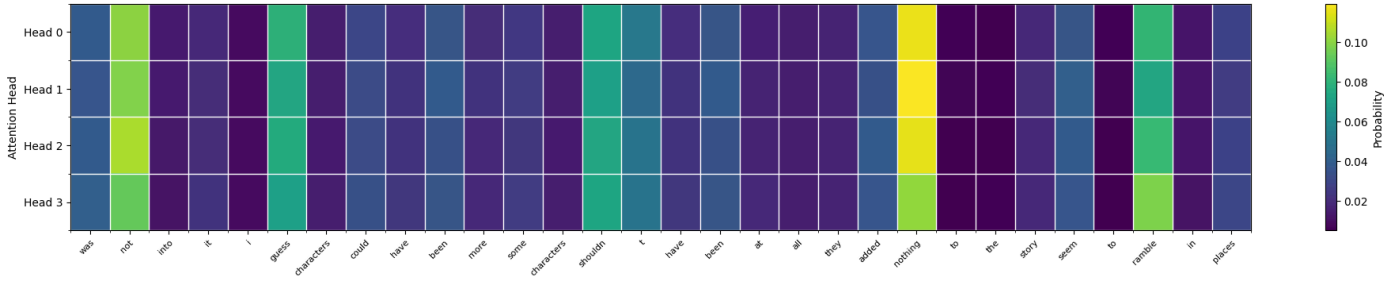
Here, the attention mechanism clearly highlights the sentiment-bearing word "love" across all four heads, with Head 2 giving it the strongest focus. Interestingly, the word "good" near the end of the text also receives moderate attention, showing how the model identifies and weighs multiple positive signals within a review. This demonstrates the model's ability to correctly prioritize emotional terms that directly express user satisfaction.

> Negative Example 1: "*Some good tech. info. but the books drags and drags and drags...*"



This example shows sophisticated attention behavior. Despite the presence of the positive word "good," all heads place higher attention on the repetition of "drags," particularly on the first instance. This suggests the model understands rhetorical devices like repetition that emphasize negative qualities. The word "not" also receives significant attention, showing the model recognizes negation. This pattern reveals how the model weighs contradictory signals and correctly prioritizes the dominant negative sentiment.

> Negative Example 2: "*Was not into it I guess. Characters could have been more...*"



The visualization reveals high attention on "not" and "nothing" (particularly in Heads 0-2), demonstrating the model's sensitivity to negative markers. Interestingly, "characters" receives moderate attention, suggesting the model may have learned domain-specific patterns where criticism of characters often indicates negative book reviews. The word "guess" also receives attention, possibly because tentative language often co-occurs with negative evaluations.

> After analyzing multiple examples, clear patterns emerge in how the attention mechanism operates. Each attention head appears to specialize in different aspects of sentiment analysis:

Head 0 tends to focus broadly on adjectives and adverbs that directly signal sentiment polarity, like "timeless," "good," and "not." This head seems to perform general sentiment detection across different contexts.

Head 1 shows similar patterns to Head 0 but often gives slightly different weights, suggesting it provides a complementary perspective on the same sentiment indicators, functioning as a form of ensemble within the model.

Head 2 consistently shows the strongest attention weights (brightest yellows) and seems particularly attuned to words that express intensity or emphasis. It gives high attention to repeated words like "drags" and strong sentiment terms like "love," suggesting it captures the emotional intensity of the review.

Head 3 appears more balanced in its attention distribution and may be capturing contextual information or domain-specific knowledge. It often attends to words that indicate product quality or performance rather than just emotional reactions.

Across examples, all heads show remarkably consistent behavior in identifying sentiment-bearing words, but they diverge in how strongly they weight different types of sentiment indicators. This multi-head approach allows the model to capture different dimensions of sentiment expression, from direct emotional statements to more subtle content evaluation, resulting in robust classification performance. The heads appear to work in concert, with each providing a slightly different perspective that contributes to the final classification decision.

> I attempted to fool the classifier with several examples containing mixed or contradictory sentiment signals. The most interesting cases reveal both the strengths and limitations of the attention mechanism in sentiment classification.

Example 1: "This book was absolutely terrible but I couldn't put it down."

- **Prediction:** Negative (attention focused heavily on "terrible")
- In this example, all attention heads strongly focus on the word "terrible" (with Head 1 giving it the highest attention weight of 0.3), while largely ignoring the positive phrase "couldn't put it down." The model correctly identifies the strong negative sentiment word but fails to properly weigh the contradictory positive sentiment that follows. This reveals a limitation in how the model handles "but" clauses that reverse or qualify the sentiment expressed in the first part of the sentence.

Example 2: "Not the worst product I've ever used, which isn't saying much."

- **Prediction:** Negative (attention concentrated on "worst")
- The visualization shows all heads giving extremely high attention to "worst" despite the negation "not." This demonstrates the model's tendency to focus on sentiment-laden keywords while sometimes misinterpreting the contextual modifications from negations. The model makes the correct overall prediction, but for potentially wrong reasons – it's likely responding to the negative word "worst" rather than understanding the nuanced criticism in "isn't saying much."

Example 3: "While I hated every moment reading it, I have to admit it was well-written."

- **Prediction:** Negative (dominant focus on "hated")
- The model places its strongest attention on "hated" across all heads, with moderate attention on "well" and "written." Despite the sentence containing a positive assessment of writing quality, the emotional response ("hated") dominates the classification. This correctly reflects how humans would likely interpret the review, suggesting the model has learned to prioritize emotional reactions over technical merits in reviews.

Example 4: "The story was predictable and boring, but somehow it kept me engaged until the end."

- **Prediction:** Negative (strong focus on "boring")
- All attention heads heavily concentrate on "boring," with moderate attention on "predictable," while giving less weight to the positive phrase "kept me engaged." This example shows how certain negative descriptors can overwhelm positive engagement signals in the model's decision-making process. The attention

mechanism is clearly identifying sentiment-bearing words, but the model may be overweighting certain negative terms.

Example 5: "I wouldn't recommend this to my friends, but I don't regret buying it."

- **Prediction:** Negative (attention distributed across "wouldn't," "friends," and "but")
- This example shows more balanced attention distribution, with heads focusing on both "wouldn't" (negative signal) and "friends"/"but" (contextual signals). Despite the statement about not regretting the purchase, the model prioritizes the explicit recommendation signal. This aligns with how recommendation language typically dominates review interpretation.

These examples demonstrate that while the attention mechanism successfully identifies sentiment-bearing words, it sometimes struggles with complex sentiment expressions where contrary opinions are presented. The model tends to give higher weight to strong emotional terms and explicit recommendation language, occasionally at the expense of contextual nuances. The attention visualization provides an excellent explanation for the model's decisions, revealing exactly which words influenced the classification most heavily. This transparency helps identify where the model succeeds and where it could be improved to better handle mixed sentiment expressions.
