



L-Università ta' Malta
Faculty of Information &
Communication Technology

Department of
Computer Information
Systems

Word Embeddings

Coursework 2024/2025

Steve Galea (0015902H)

Study-unit: **Topics in Applied Data Science**

Code: **CIS5231**

Lecturer: Professor John Abela

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines” (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the assignment submitted is my work, except where acknowledged and referenced.

I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

STEVE GALEA

Student Name



Signature

CIS5231

Course Code

Word Embeddings Coursework 2024/2025

Title of work submitted

18/07/2025

Date

Table of Contents

Contents

Table of Contents	3
Table of Figures	4
Introduction.....	5
Literature Review Of Word Embedding Techniques	6
Design Decisions:	9
1. Acquisition and Pre-Processing (1_data_pre_processing.py)	9
2. Models in models/cbow.py and models/skipgram.py	11
3. Training Process (2_and_3_pytorch_model_training_and_optimisations.py).....	13
3.1 Training Loop Implementation	13
3.2 Hyper-Parameter Tuning	14
3.3 Performance Monitoring	15
3.4 Adaptation	15
4. Visualisation Techniques Used (4_tSNE_visualisation_and_analysis.py)	15
Discussion of results.....	18
Challenges Encountered	29
Possible Enhancements/Further work.....	29
Convergence Plots & tSNE embeddings	30
Source Code Appendix.....	59
References	69
ChatGPT usage	70

Table of Figures

Figure 1.1: The above figure was replicated from Mikolov et al. [4], showing the two distinct architectures.	7
Figure 1.1: Initial raw data set, with business and sports news articles	9
Figure 4.1: Example plot showing similar/opposite words being plotted in a space	17
Figure 4.2: The figure shows the full list of experiments conducted available on Github SteveGalea/CIS5231	17
Figure 4.1: Skipgram 0.01 learning rate (left) vs Skipgram 0.001 learning rate	19
Figure 4.2: CBOW 0.01 learning rate (left) vs CBOW 0.001 learning rate (right)	19
Figure 4.3: CBOW 32 batch size(left) vs CBOW 256 batch size (right)	20
Figure 4.4: CBOW 16 batch size (left) vs CBOW 256 batch size (right)	20
Figure 4.5: SkipGram 64 batch size (left) vs SkipGram 256 batch size (right)	21
Figure 4.6: SkipGram 64 batch size (left) vs SkipGram 256 batch size (right)	21
Figure 4.7: Skip-gram (batch 256, learning rate 0.01) window size 2 (left) vs Skip-gram (batch 256, learning rate 0.01) window size 5 (right)	22
Figure 4.8: Skip-gram (batch 32, learning rate 0.0001) 10 window size (left) vs skip-gram (batch 256, learning rate 0.05) window size 10 (right)	22
Figure 4.9: CBOW (batch 256, learning rate 0.01) window 2 (left) vs window 5 (right)	23
Figure 4.10: The business category, for Skip-gram (batch 256, learning rate 0.01), window size 5	24
Figure 4.11: The colour category for Skip-gram (batch 256, learning rate 0.01), window size 5	24
Figure 4.12: The country category for Skip-gram (batch 256, learning rate 0.01), window size 5	25
Figure 4.13: The gender category for Skip-gram (batch 256, learning rate 0.01), window size 5	26
Figure 4.14: The position category for Skip-gram (batch 256, learning rate 0.01), window size 5.	26
Figure 4.15: The royalty category for Skip-gram (batch 256, learning rate 0.01), window size 5.	27
Figure 4.16: The sport category for Skip-gram (batch 256, learning rate 0.01), window size 5.	28

Introduction

The Code for this assignment can be found on GitHub with all the binaries and trained models: [SteveGalea/CIS5231](#)

Applications of word embedding techniques are wide-ranging — from text classification, text generation (such as summarisation), and question answering, to speech recognition and machine translation. To achieve the above, word embedding techniques map words, along with their surrounding context at times, into continuous vector spaces [1].

Word Embeddings are a type of Vector Space Model used in Natural Language Processing (NLP) that are learnt from large data corpora in an unsupervised manner. These models identify patterns from language data sources and represent them as dense and continuous vectors. Unlike one-hot encoded data, which is a sparse representation, using such vectors for representations offers various benefits [1]. Within the domain of NLP, word embeddings can be learned using prediction-based models that rely on local context or through count-based models that leverage global word occurrences [11].

Dense vectors are computationally less expensive and do not require human curation or extensive manual feature engineering to represent a word numerically. The word vector is determined effectively through its context, and such vector embeddings enable models to handle noise or missing data better, make generalisations and inferences from unseen data. This is because word embeddings capture the semantic similarity and cluster words with similar meaning together in the embedding space. This is very unlike rule-based symbolic methods, relying on direct human labelling and offering limited scalability and vocabulary. In dense vector word embeddings[1-3]. Dense vectors produced by earlier models like Word2Vec are static representations, meaning that the vector does not account for revision of the vector representation, even if the context is different. This limitation of polysemy and homonymy is addressed by later models like BERT and ELMo [3]. These vector embeddings have become foundational components of modern neural architectures and deep sequence models [1,3, 8-11]. Notably, Vaswani et al. [10] incorporated them into the multi-layered Transformer architecture, which ultimately paved the way for models like ChatGPT.

Literature Review Of Word Embedding Techniques

Mikolov et al. [4] and Pennington et al. [5] produced Word2Vec and GloVe models, respectively, to come up with highly accurate semantic word encodings. Other models, such as FastText, created by Joulin et al. [6] and Bojanowski et al. [7], BERT by Devlin et al. [8], or ELMo by Peters et al. [9], aimed to address limitations in the vectors encountered by the previous two models.

Mikolov et al [4] created the Continuous Bag Of Words (CBOW) and Skip-gram architectures to compress words from very large datasets into a dense set of vectors via a log-linear classifier. CBOW and Skip-gram both make use of feed-forward functions, aiming to learn word representations. CBOW employs a feed-forward network function which accepts as input a context of many words, with a missing middle target word. CBOW hence tries to predict the central target word from the surrounding words. The order of the words does not matter. Skip-gram is the opposite, where with a target word as an input, it tries to predict the surrounding context. It is more complex computationally than CBOW when the context window range is increased to consider distant word vectors; however, the quality of the overall word vectors is increased. These two architectures were tested with the Google News 6B token corpus, and the vocabulary tested was limited to the most frequent 1 million, and later 30 thousand words, due to time constraints. The experiments carried out involved experimenting with 1 or 3 epochs; varying (e.g. 300 or 600) embedding dimensionality; 783M or 1.6B words, and of course CBOW or Skip-gram. The model was trained with mini-batches, backpropagation and Adagrad stochastic gradient descent (learning rates 0.025 to 0) on a single CPU. Large-scale tests were carried out as well with multiple CPU instances using the full 6B dataset, and 1000-sized embedding dimensions for both architectures. For word similarity and analogy evaluation tasks, various semantic and syntactic questions were carried out to evaluate the accuracy, with the authors coming up with various relationships (e.g. France – Paris, USA - pizza, cold - colder). n. With larger datasets and larger embedding dimensions, the authors indicate that the performance increases, and high-quality embeddings can be achieved with a simple architecture.

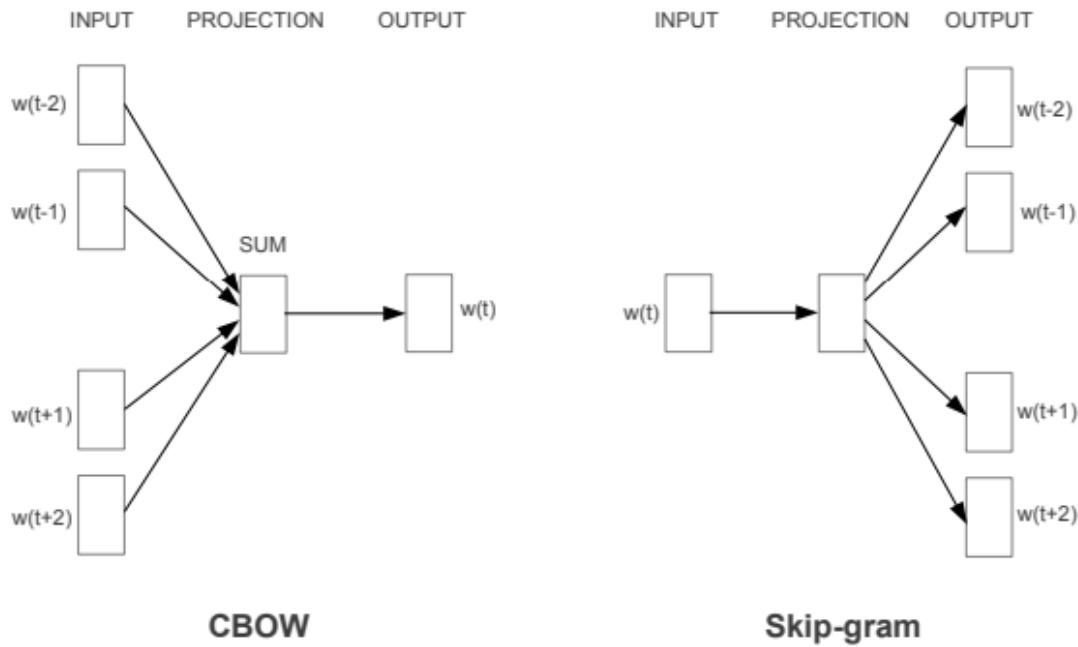


Figure 1.1: The above figure was replicated from Mikolov et al. [4], showing the two distinct architectures.

Pennington et al. [5] combine the benefits of the global matrix factorisation methods (e.g. latent semantic analysis (LSA)) and the local context window models achieved by Mikolov et al [4] (Skip-gram). GloVe counts the word occurrences in the provided corpus as a word-context co-occurrence matrix, extracts probabilities and factorises them. This effectively captures the global probabilities and relationships within the corpus, unlike Word2Vec, which focuses on a local window context. The same Semantic and Syntactical tests as Mikolov et al. [4] were carried out over different models and hyperparameters. The datasets used were Wikipedia dumps from 2010 and 2014 and Gigaword5, to obtain 6 billion tokens. The top 400,000 most frequent words were used, 300+ vector embeddings (trained with 50 epochs for <300 , but 100 epochs for >300), and Adagrad with an initial learning rate of 0.05 was used. An investigation into the left and right sentence contexts was made. The GloVe model outperformed the other baselines, providing superior and faster word vectors and robustly supporting larger datasets. Word similarity, word analogy, and named entity recognition are the NLP tasks for which the GloVe model appears to outperform.

Word2Vec and GloVe, however, both suffer from out-of-vocabulary words as a top X filter is done to limit the vocabulary [3-5]. With this in mind, authors [6-7] created FastText, an extension of the skip-gram model, and in it, it represents each word as an n-gram of characters. The authors made use of Czech, French, Spanish, German, English and Russian Wikipedia dumps for training. The model used was an RNN with 650 LSTM units and trained by Adagrad. It was found that morphologically richer languages like German tend to separate compound nouns into morphemes. Besides that, the FastText model can generate word vectors for out-of-vocabulary words by averaging the n-grams [6-7]. Unfortunately, the n-grams model may not scale, and it can grow rapidly in size, possibly posing a notable limitation.

The limitation of static embedded vectors is a pertinent issue across the simpler models created by [4-7]. BERT by Devlin et al. [8] is a multi-stacked transformer-based bidirectional encoder-decoder architecture that directly generates context-dependent embeddings in the feedforward loop. Two token embeddings are created within BERT to create a dynamic vector representation for the same word, but different contexts through attention mechanisms; the first is WordPiece tokenisation to represent the syntax and semantics, and the second is tokenisation for the position. These two embedded encodings properly handle the limitation of polysemy and homonymy, and the use of bi-directional context, unlike the word embeddings of [4-7] can consider the entire context (instead of a window in Word2Vec Skipgram[4] and FastText [6-7]) from both directions. ELMo by Peters et al. [9] similarly makes use of a bidirectional LSTM architecture to generate dynamic context-dependent embeddings.

Design Decisions:

1. Acquisition and Pre-Processing (1_data_pre_processing.py)

The News Articles¹ dataset was acquired from Kaggle. It contains 2,692 scraped news articles and headlines related to business and sports from 2015 to 2017. For this assignment, since computations will be carried out on a CPU and a domain-specific dataset is needed, the **sports** articles (1408 records) will be focused on.

1	Article	Date	Heading	NewsType
2	KARACHI: The Sindh government has decided to bring down public	1/1/2015	sindh govt decides to cut public transport fares by 7pc kti rej	business
3	HONG KONG: Asian markets started 2015 on an upswing in limited	1/2/2015	asia stocks up in new year trad	business
4	HONG KONG: Hong Kong shares opened 0.66 percent lower	1/5/2015	hong kong stocks open 0.66 percent lower	business
5	HONG KONG: Asian markets tumbled Tuesday following painful	1/6/2015	asian stocks sink euro near nine year	business
6	NEW YORK: US oil prices Monday slipped below \$50 a barrel for the	1/6/2015	us oil prices slip below 50 a barr	business
7	New York: Oil prices tumbled Tuesday to fresh 5.5-year lows as	1/7/2015	oil hits new 5.5 year lows as saudis defend	business
8	KARACHI: Strong bulls on Friday pulled the benchmark KSE-100	1/9/2015	bullish kse jumps over 33000 psychological barrier	business
9	Singapore: Oil fell further in Asia Monday, with weak demand and a	1/12/2015	oil falls further in asian trad	business
10	KARACHI: Wholesale market rates for sugar dropped to less than Re	1/13/2015	sugar prices drop to rs 49.80 in sind	business
11	SYDNEY: Oil prices fell 1 percent on Wednesday after the World	1/14/2015	oil extends losses as world bank cuts growth for	business
12	TOKYO: Tokyo stocks opened 0.74 percent lower on Wednesday, hit	1/14/2015	tokyo stocks open 0.74 percent lower	business
13	HONG KONG: Hong Kong stocks edged up 0.24 percent in early	1/15/2015	hong kong stocks open 0.24 percent higher	business
14	London: World oil prices rebounded Friday after the International	1/16/2015	oil market rebounds on ieo repor	business
15	ISLAMABAD: Long queues of vehicles on fuel stations were visible in	1/16/2015	govt may take up to ten days to overcome petrol shortag	business
16	SINGAPORE: Brent crude oil prices traded around \$50 a barrel on	1/19/2015	brent crude steady around 50	business
17	ISLAMABAD: A two member committee tasked to probe the	1/20/2015	report presented to pm holds ogra responsible for fuel crisi	business
18	ISLAMABAD: The National Electric Power Regulatory Authority	1/21/2015	notification issued for reduction in electricity charges for karaci	business
19	Hong Kong: Asian markets extended their rally this week, while the	1/22/2015	asia stocks mostly higher ahead of ecb meeting	business
20	ISLAMABAD: The Economic Coordination Committee of the Cabinet	1/23/2015	ecc approves export of 1.2 million tons of	business
21	SINGAPORE: Saudi Arabia's new king is expected to continue a	1/23/2015	new saudi king seen holding line on opec policy to keep oil outp	business

Figure 1.1: Initial raw data set, with business and sports news articles

Pre-Processing Steps:

1. Text Normalisation: Convert all text to lowercase to maintain consistency.
2. Punctuation Removal: Remove punctuation using regular expressions to clean the text, as per the following code snippet.

```
▲ Steve Galea *
def clean_text(text):
    # https://www.geeksforgeeks.org/machine-learning/python-efficient-text-data-cleaning/
    # text normalisation (lower-case, etc)
    text = text.lower()
    # ignore non-ascii characters
    text = text.encode('ascii', 'ignore').decode()
    # keep only words
    text = re.sub(r'[^a-z\s]', '', text)
    # trim spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

df['clean_text'] = df['Article'].dropna().apply(clean_text)
```

3. Tokenisation: Split text into individual tokens (words) & Stop-Word Removal: Filter out common stop words to reduce noise and focus on semantically rich words..

```
# tokenisation & stop-word removal
df['tokens'] = df['clean_text'].apply(lambda x: [t for t in word_tokenize(x) if t not in STOPWORDS])
```

5. Vocabulary Limiting/Filtering: Construct a vocabulary of the top 10,000 most frequent words, replacing infrequent words with an <UNK> token. Lemmatisation was carried out using **spacy**.

¹ <https://www.kaggle.com/datasets/asad1m9a9h6mood/news-articles>

Indexing was also carried out, where I converted tokens into numerical indices based on the constructed vocabulary

```
def lemmatise_tokens(tokens):
    doc = nlp(" ".join(tokens))
    return [token.lemma_ for token in doc]

df['lemmas'] = df['tokens'].progress_apply(lemmatise_tokens)

# vocabulary creation - most common 10k words and mapped to indices
all_lemmas = [lemma for doc in df['lemmas'] for lemma in doc]
lemma_freq = Counter(all_lemmas)

most_common = lemma_freq.most_common(MAX_VOCAB_SIZE - 1) # -1 for UNK
vocab = {word: idx + 1 for idx, (word, _) in enumerate(most_common)} # start from 1
vocab[UNK_TOKEN] = 0 # UNK mapped to 0

# indexing tokens into numerical indices based on the constructed vocabulary
# Steve Galea
def convert_to_indices(lemmas):
    return [vocab.get(token, vocab[UNK_TOKEN]) for token in lemmas]

df['lemmas_indices'] = df['lemmas'].apply(convert_to_indices)
```

7. Dumping Data created -> the data created was dumped into the Data folder.

```
# save
df.to_csv(f"Data/preprocessed_{news_type}_articles.csv", index=False)
df.to_pickle(f"Data/preprocessed_{news_type}_articles.pkl")

with open(f"Data/vocab_{news_type}_dict.pkl", "wb") as f:
    pickle.dump(vocab, f)
```

Sample script run:

```
i_data_pre_processing x
[nltk_data]  C:\Users\steve\AppData\Roaming\nltk_data...
[nltk_data]  Package punkt_tab is already up-to-date!
100%|██████████| 1408/1408 [00:25<00:00, 54.38it/s]
Total business articles processed: 1408
Total tokens: 228129
Vocabulary size: 10000

Top 20 most frequent lemmas:
say          1893
first        1727
test         1496
team         1376
match        1330
cricket      1328
england      1308
pakistan     1301
play          1287
world         1286
two           1228
run            1205
make           1164
ball           1089
take           1082
wicket         1061
one             968
captain        938
second          914
four            902

Process finished with exit code 0
```

2. Models in models/cbow.py and models/skipgram.py

- The models created follow the ChatGPT boilerplate code(prompt in the ChatGPT log attached). This 2 source code links were then used to create the CBOW and Skipgram Code and training loop:
 1. Prompt in the appendix
 2. [pytorch-continuous-bag-of-words/cbow.py at master · FraLotito/pytorch-continuous-bag-of-words · GitHub](#)
 3. [Skip-Gram Word2Vec Algorithm Explained | by Ido Leshem | Medium](#)
- One-hot encoding & Binary vector representation were not used, as dense vector representations are preferred. Instead, training was performed using mini-batches and computing the CrossEntropyLoss & optimisers upon was preferred and put in **2_and_3_pytorch_model_training_and_optimisations.py** handle the training and stochastic descent with Adam.
- Softmax was implemented in both CBOW and Skipgram models to get log-probabilities.

CBOW Model documentation:

1. Predicts a target central word, from the context words
2. Prediction-based model with a simple feed-forward network

CBOW Model attributes:

Attribute	Why needed
self.embeddings	Converts word indices to dense vectors. For this assignment, we are using a 256 inner vector dimension
embeds.mean	Takes all context words, gets an average, forwards to self.linear
self.linear	Layer to receive and forward an average embedding vector from all contexts to be encoded, forwards to log_softmax
Log_softmax	Log-probabilities to predict the target word

Function	Why needed
Forward	Receives word indices and outputs target central word log-probabilities, for updating in the training loop in 2_and_3_pytorch_model_training_and_optimisations.py

Skipgram Model documentation:

1. Predicts the context, from an initial central word.
2. Also a prediction-based model.

Attribute	Why needed
self.embeddings	Converts word indices to dense vectors. For this assignment, we are using a 256 inner vector dimension
self.linear	Layer to receive a center word embedding, forwards to log_softmax
Log_softmax	Log-probabilities to predict the context words

Function	Why needed
Forward	Receives word index and outputs context log-probabilities, for updating in the training loop in 2_and_3_pytorch_model_training_and_optimisations.py

CBOW & Skipgram Dataset:

1. Prepares samples for CBOW/Skipgram model
 - a. On get item, CBOW dataset gets a set of contexts and a single target vector from the vocabulary.
 - b. On get item, Skipgram dataset gets a single index and a set of context vectors to be predicted from the vocabulary.
 - c. As a failsafe, UNK_TOKEN vector is provided in case the index passed is an unseen word

Utility dataset functions:

1. **generate_cbow_data**: iterates over a token list and generates (context, target) vector pairs
2. **generate_skipgram_data**: iterates over a token list, and prepares the center tokens according to the window used.

The embeddings for both models are updated in the training loop after calculating the CrossEntropyLoss by **loss.backward()**. The Adam optimizer applies the gradients at this point using **optimizer.step()**, as in the snippet in **3.1 Training Loop Implementation**.

3. Training Process

(2_and_3_pytorch_model_training_and_optimisations.py)

3.1 Training Loop Implementation

```
# prepare data and Model
if model_name == "cbow":
    model_data = generate_cbow_data(df['lemmas'].tolist(), window_size=window_size)
    dataset = CBOWDataset(model_data, vocab)
    model = CBOWModel(len(vocab), embedding_dim).to(device)
else:
    model_data = generate_skipgram_data(df['lemmas'].tolist(), window_size=window_size)
    dataset = SkipGramDataset(model_data, vocab)
    model = SkipGramModel(len(vocab), embedding_dim).to(device)

# mini batches
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, num_workers=4)

# loss, and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_history = []

# training loop
for epoch in range(epochs):
    total_loss = 0
    model.train()

    # adapted from https://github.com/FraLotito/pytorch-continuous-bag-of-words/blob/
    # adapted training loop from both https://leshem-ido.medium.com/skip-gram-word2ve
    # also optimised by ChatGPT
    for context_idxs, target_idx in dataloader:
        context_idxs = context_idxs.to(device)
        target_idx = target_idx.to(device)
        optimizer.zero_grad()
        # forward
        output = model(context_idxs) # output shape: (batch_size, vocab_size)
        loss = criterion(output, target_idx)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(dataloader)
    loss_history.append(avg_loss) # log the average loss
    print(f"Epoch {epoch + 1}/{epochs} Loss: {avg_loss:.4f}")
```

As seen above, the necessary data loader is called, and the dataset is divided into mini-batches. The Dataset classes and functions convert raw dataset indices to mini-batch sets of training vectors.

Models were trained locally on the CPU and Google Colab. Sample Screen Dump of training process:

The screenshot shows a Google Colab notebook titled "CIS5231-GPU run". The code cell at the top prints a warning message and a series of loss values for 15 epochs. A plot titled "Training Loss Over Epochs" shows the average loss decreasing from approximately 7.6 at epoch 1 to about 7.0 at epoch 15. The final model is saved to a local directory.

```
File Edit View Insert Runtime Tools Help
Share Connect T4
File Commands + Code + Text Run all
99     print(f"Model saved to {save_dir}")
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624: UserWarning: This DataLoader will create 4 workers which may be too many for your system. Consider specifying max_workers=0 or a lower value.
  warnings.warn(
Epoch 1/15, Loss: 7.5968
Epoch 2/15, Loss: 7.2362
Epoch 3/15, Loss: 7.1655
Epoch 4/15, Loss: 7.1259
Epoch 5/15, Loss: 7.0981
Epoch 6/15, Loss: 7.0762
Epoch 7/15, Loss: 7.0587
Epoch 8/15, Loss: 7.0440
Epoch 9/15, Loss: 7.0314
Epoch 10/15, Loss: 7.0198
Epoch 11/15, Loss: 7.0101
Epoch 12/15, Loss: 7.0009
Epoch 13/15, Loss: 6.9922
Epoch 14/15, Loss: 6.9842
Epoch 15/15, Loss: 6.9768
Training Loss Over Epochs
Model saved to models/sports/skipgram/skipgram_model_32b_0.0001l_15e_10w_files
```

3.2 Hyper-Parameter Tuning

All training and experimentation were run on seed 42 to prevent any unwanted variations.

```
Steve Galea
def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)
TSNE(n_components=2, random_state=42,
```

26 hyperparameter tuning experiments were run, 13 for SkipGram and 13 for CBOW. The following hyperparameters were experimented with:

- Batch sizes: 16 (CBOW only), 32, 64, 128, 256
- Epochs (including checkpoints): 10, 20, 30, 40, 50 or 100
- Window sizes: 2, 3, 4, 5, 10
- Learning rates: 0.0001, 0.001, 0.01, 0.05
- Models: cbow, skipgram

3.3 Performance Monitoring

The models' performance was measured through training loss. Models with the lowest training loss and at a converged state were considered to have finished training. All obtained Loss Plots are listed in the appendix. At each checkpoint, the training loss and vectors at that checkpoint are exported and saved locally.

```
avg_loss = total_loss / len(dataloader)
loss_history.append(avg_loss) # log the average loss
print(f"Epoch {epoch + 1}/{epochs}, Loss: {avg_loss:.4f}")
# save the trained model
if (epoch + 1) % 10 == 0:
    os.makedirs(f"{save_dir}/checkpoints/checkpoint_{epoch}e", exist_ok=True),
    torch.save(model.state_dict(), f"{save_dir}/checkpoints/checkpoint_{epoch}e/checkpoint_{epoch}e_{export_file_name}")
    loss_df = pd.DataFrame({'epoch': range(1, epoch + 2), 'loss': loss_history})
    loss_df.to_csv(f"{save_dir}/checkpoints/checkpoint_{epoch}e/training_loss_log_checkpoint_{epoch}e_{export_file_name}.csv", index=False)

torch.save(model.state_dict(), f"{save_dir}/{export_file_name}")
loss_df = pd.DataFrame({'epoch': range(1, epochs + 1), 'loss': loss_history})
loss_df.to_csv(f"{save_dir}/training_loss_log_{export_file_name}.csv", index=False)

plt.plot(range(1, epochs + 1), loss_history, marker='o')
plt.title("Training Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Average Loss")
plt.grid(True)
plt.tight_layout()
plt.savefig(f"{save_dir}/loss_plot_{export_file_name}.png")
plt.show()
```

3.4 Adaptation

To complete within a 2-3 hour training window on a CPU, one should lower the epoch count. Note, these were estimated, but should keep to a 2-3 hour window.

Model	Epochs	Learning Rate	Window size	Batch size
CBOW	100	0.01	2	128
SkipGram	12	0.01	5	256

For CBOW, ~50 epochs were observed to fit in a 2-3 hour window, but for SkipGram, this took longer due to the more complex processing.

4. Visualisation Techniques Used

(4_tSNE_visualisation_and_analysis.py)

A training loss against the Epochs curve was used to determine if the model converged (e.g. **loss_plot_skipgram_model_256b_0.05l_10e_10w.pth.png**).

```
# get embedding weights: shape (vocab_size, embedding_dim)
embeddings = model.embeddings.weight.data.cpu().numpy()

## dimensionality reduction:
## use t-SNE to reduce the high-dimensional embeddings to 2D space. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
tsne = TSNE(n_components=2, random_state=42, perplexity=30, max_iter=1000)
embeddings_2d = tsne.fit_transform(embeddings)
```

Also, a dedicated script for analysis was created **4_tSNE_visualisation_and_analysis.py**. In it one can input hyperparameters and specify which saved embeddings to analyse.

```

27     device = torch.device("cpu")
28
29     # set params
30     model_name = "cbow" # "skipgram"
31     batch_size = 256
32     lr = "0.01"
33     epochs = "50"
34     window = "2"
35     filepath = f"models/sports/{model_name}/{model_name}_model_{batch_size}b_{lr}l_{epochs}e_{window}"
36     filename = f"{model_name}_model_{batch_size}b_{lr}l_{epochs}e_{window}w.pth"

```

After loading the model's embeddings at the last saved state, TSNE is applied to the embeddings.

```

if model_name == "cbow":
    model = CBOWModel(len(vocab), embedding_dim).to(device)
else:
    model = SkipGramModel(len(vocab), embedding_dim).to(device)

# load saved state dict
model.load_state_dict(torch.load(filepath + filename, map_location=device))
print(model.eval())

# get embedding weights: shape (vocab_size, embedding_dim)
embeddings = model.embeddings.weight.data.cpu().numpy()

# dimensionality reduction:
# use t-SNE to reduce the high-dimensional embeddings to 2D space. https://scikit-learn.org/stable/
tsne = TSNE(n_components=2, random_state=42, perplexity=30, max_iter=1000)
embeddings_2d = tsne.fit_transform(embeddings)

```

Also, for each model, all TSNE was used to reduce the high-dimensional embeddings of 10,000 words to a 2D space to visualise and plot any clusters (e.g. file **top_10000_tsne_word_embeddings_skipgram_model_256b_0.05l_10e_10w.pth.png**). To supplement this, a dedicated CSV file with all coordinate and word data was created to describe points and clusters (e.g. **tsne_embeddings_skipgram_model_256b_0.05l_10e_10w.pth.csv**)

In addition, the following categories were manually created to have a more manageable visualisation to interpret. The conditions for the categories and the words were chosen intuitively and were derived from the dataset.

```

# define groups for better analysis
categories = {
    'Gender': ["man", "woman", "boy", "girl", "male", "female", "men", "women", "womens"],
    'Royalty': ["king", "queen", "man", "woman", "prince", "princess"],
    'Business': ["businessman", "man", "business", "businessmenin"],
    'Sports': ["foot", "ball", "football", "soccer", "american", "cricket", "india", "basketball", "america", "states",
               "olympics", "major", "tournament", "handball"],
    'Country': ["america", "american", "british", "english", "england", "united", "states", "ozil", "mueller",
                "germany", "germans", "turkey", "bonucci", "armonucci", "italy"],
    'Colors': ["yellow", "gold", "red", "green", "blue", "angry"],
    'Position': ["silver", "second", "gold", "first", "third", "bronze", "win", "lose"]
}

```

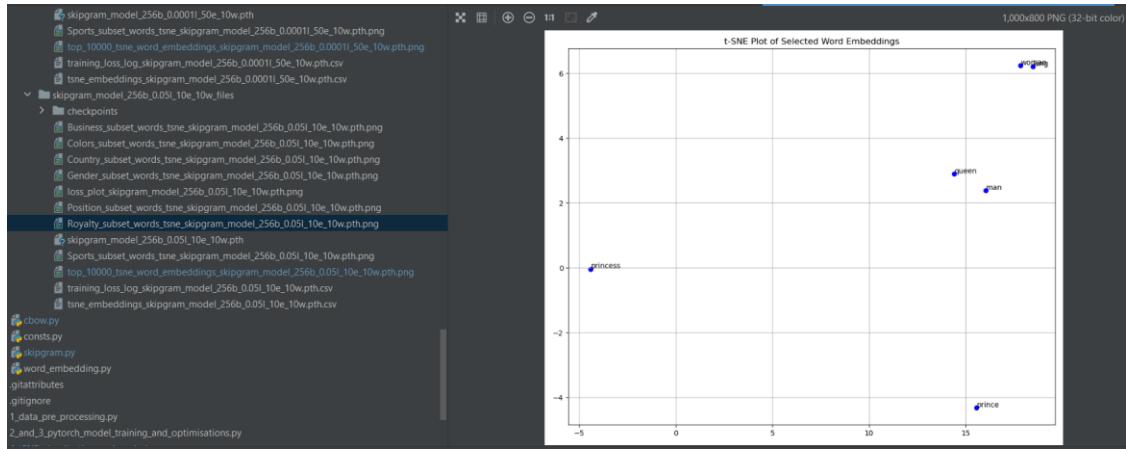


Figure 4.1: Example plot showing similar/opposite words being plotted in a space.

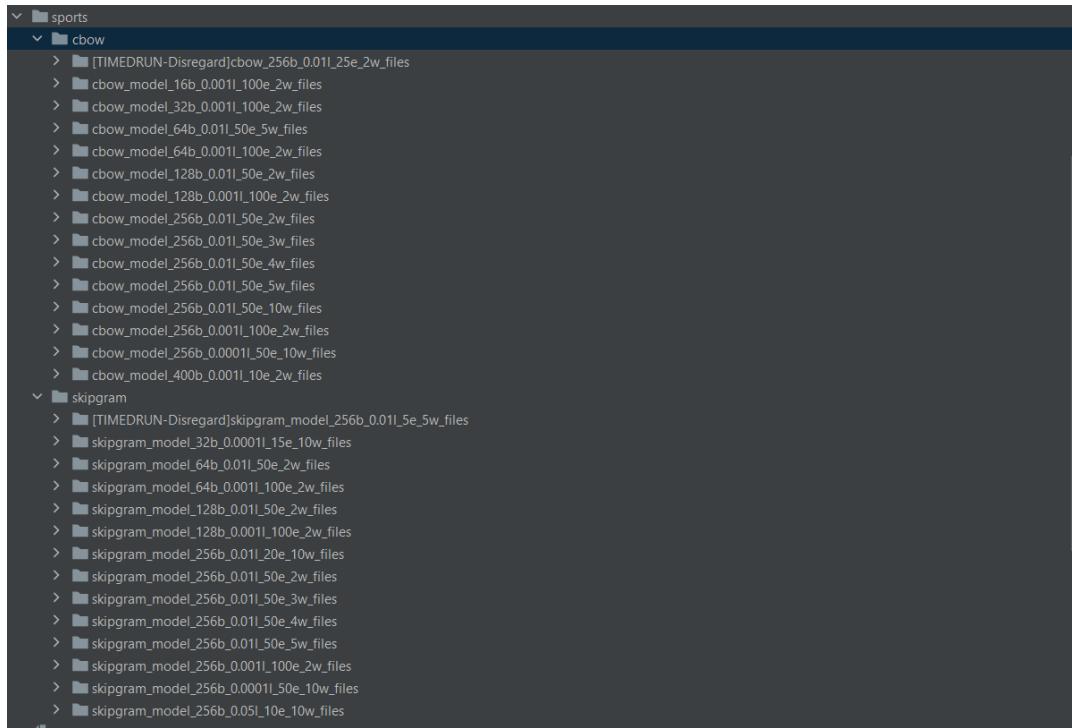


Figure 4.2: The figure shows the full list of experiments conducted available on Github [SteveGalea/CIS5231](https://github.com/SteveGalea/CIS5231)

Discussion of results

- The full plots, visualisations and results can be located on Github [SteveGalea/CIS5231](#).
- All the models converged successfully, and both models got fitted on tSNE without issue and produced clusters of sorts. Most notably, Skip-gram models with 3-5, and 10 windows produced better clustering than CBOW models.
- Consistent with the Literature Reviewed, CBOW was quicker than Skip-gram. Increasing window size made Skip-gram training slower and more complex.
- Refer to the Appendix **Convergence Plots & tSNE embeddings** for specific model plots, or access GitHub to view specific images and results by the model name.

Results will be discussed as follows:

1. Effect of the number of epochs
2. Effect of learning rate
3. Effect of batch size for Skip-gram and CBOW
4. Effect of window size on Skip-gram and CBOW
5. Skip-gram generic results & clusters overall vs CBOW generic results & clusters overall
6. A further analysis per category on a particular model

Effect of the number of epochs

While it is true, more epochs mean convergence should occur, at some point, the models reached a minimum in the training loss, where the average loss did not significantly change as much from one epoch to another.

Effect of learning rate

Smaller learning rates in skip-gram (batch 64, window size 2) seemed to produce more distinctive and spread out clusters (image on the left visually has more clusters), while that of 0.001 has a large central cluster, but many small ones around it.

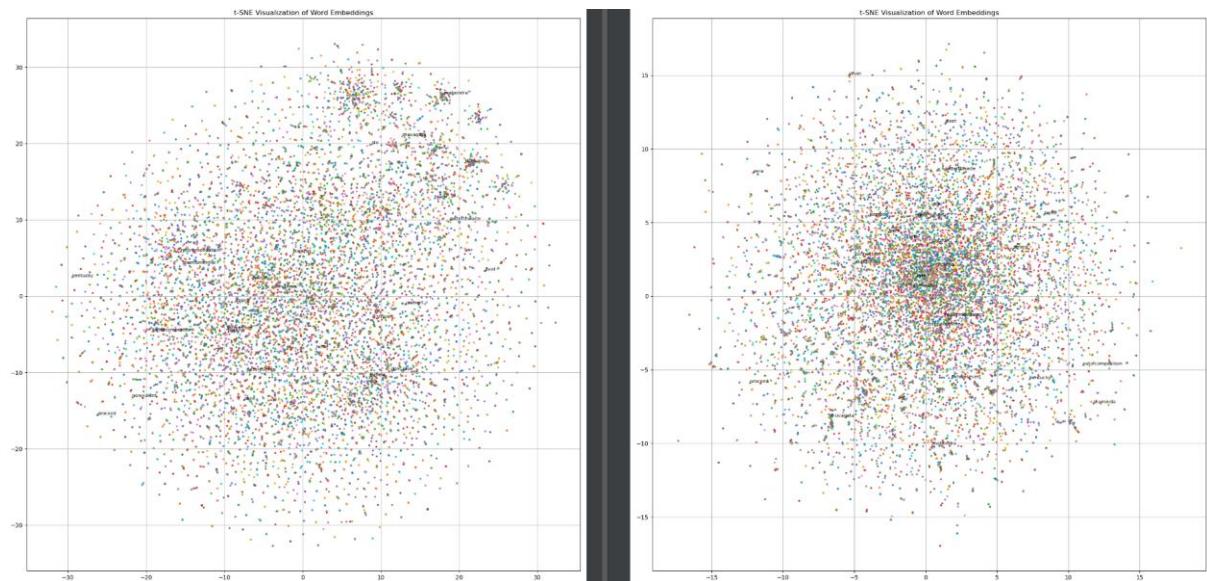


Figure 4.1: Skipgram 0.01 learning rate (left) vs Skipgram 0.001 learning rate

The same thing occurs for CBOW (batch 256, window size 2)

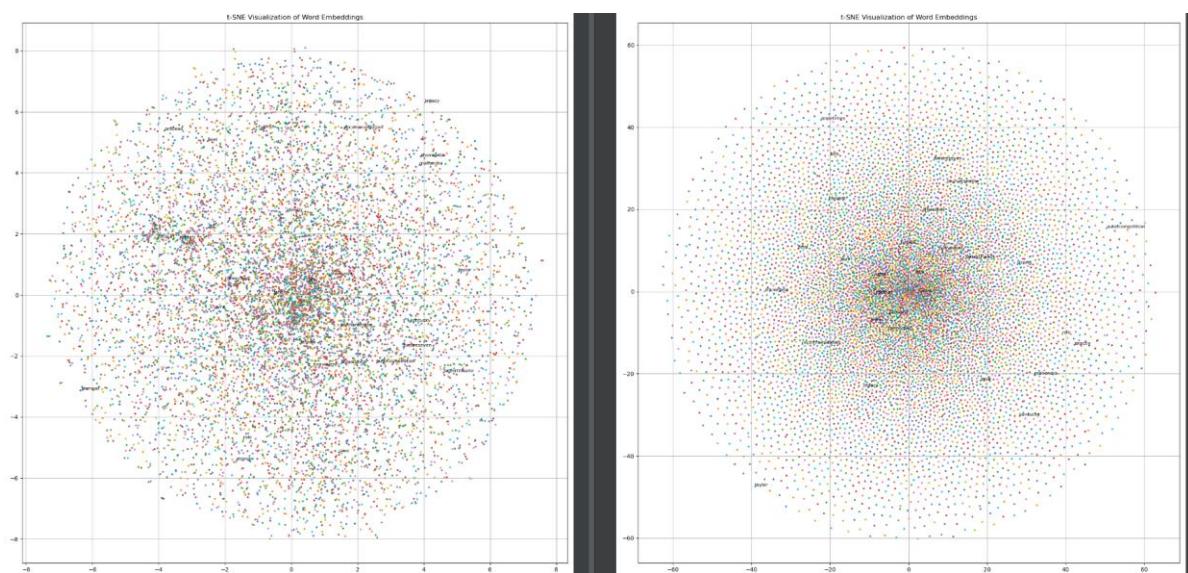


Figure 4.2: CBOW 0.01 learning rate (left) vs CBOW 0.001 learning rate (right)

Effect of batch size

Smaller mini-batches of size 16 for CBOW model (learning rate 0.001) over a window of 2 **cbow_model_16b_0.001l_100e_2w_files** did seem to provide better clusters than 32, 64, 128 and 256 (**cbow_model_256b_0.001l_100e_2w_files**) and **cbow_model_64b_0.001l_100e_2w_files** when it came to tSNE, as below.

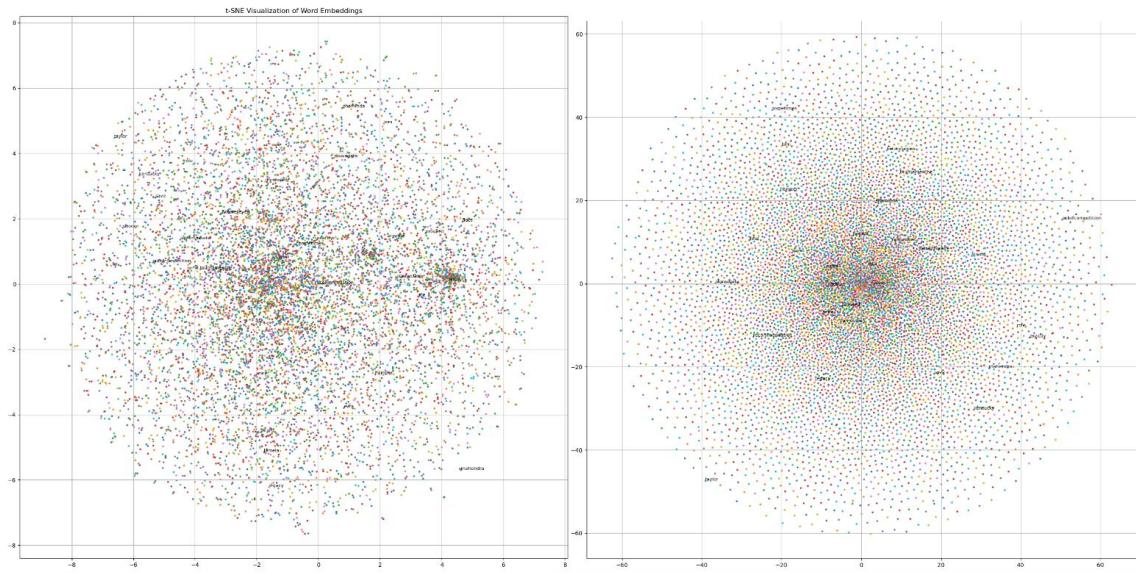


Figure 4.3: CBOW 32 batch size(left) vs CBOW 256 batch size (right)

Also, smaller batch sizes allowed quicker convergence

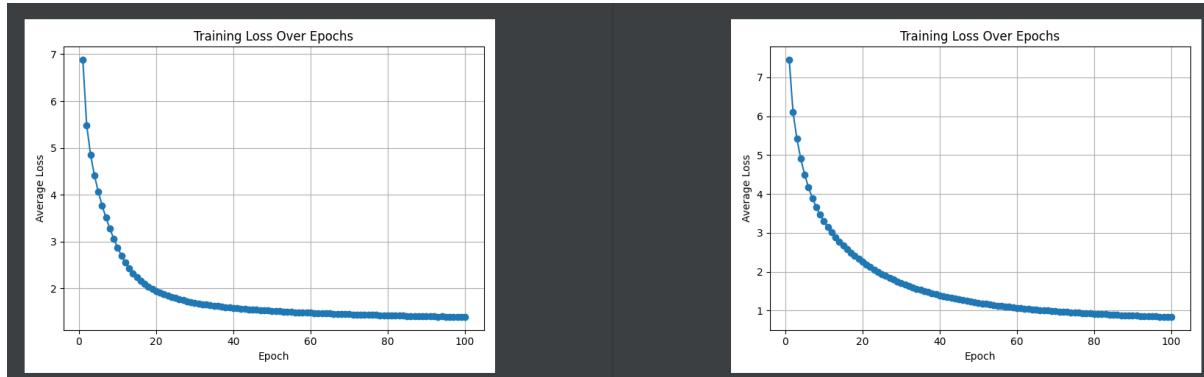


Figure 4.4: CBOW 16 batch size (left) vs CBOW 256 batch size (right)

For skip-gram, smaller batch size made the tSNE embeddings fit within a bigger range on the x- and y-axis.

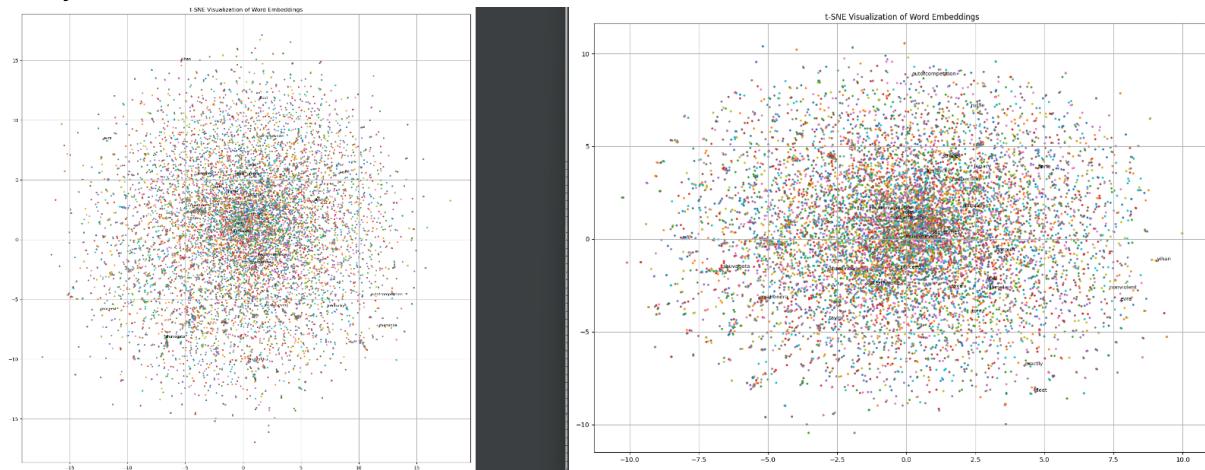


Figure 4.5: SkipGram 64 batch size (left) vs SkipGram 256 batch size (right)

A bigger batch size allowed for a smaller average loss value.

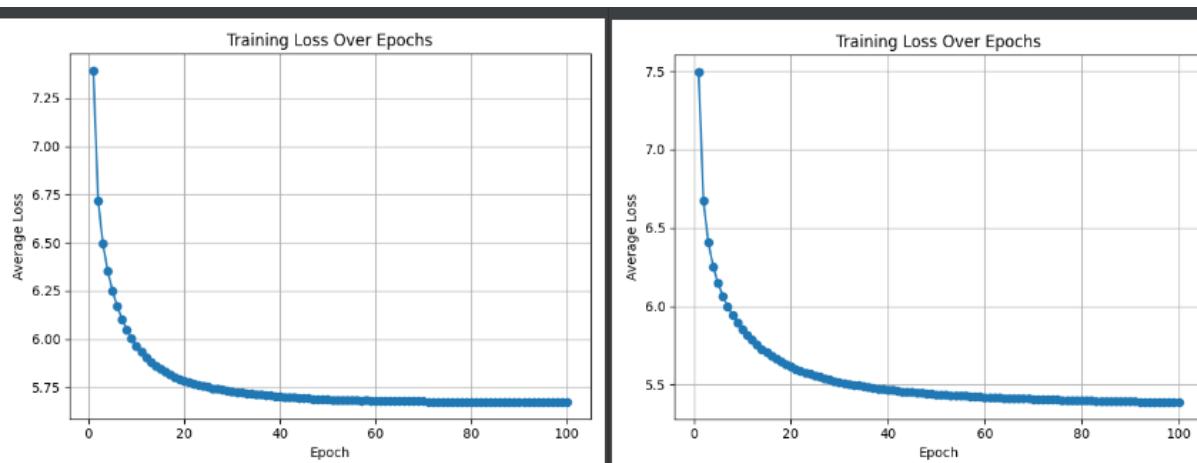


Figure 4.6: SkipGram 64 batch size (left) vs SkipGram 256 batch size (right)

Effect of window size

With a larger window size (> 2), Skip-gram models seemed to produce more distinguished clusters, as below:

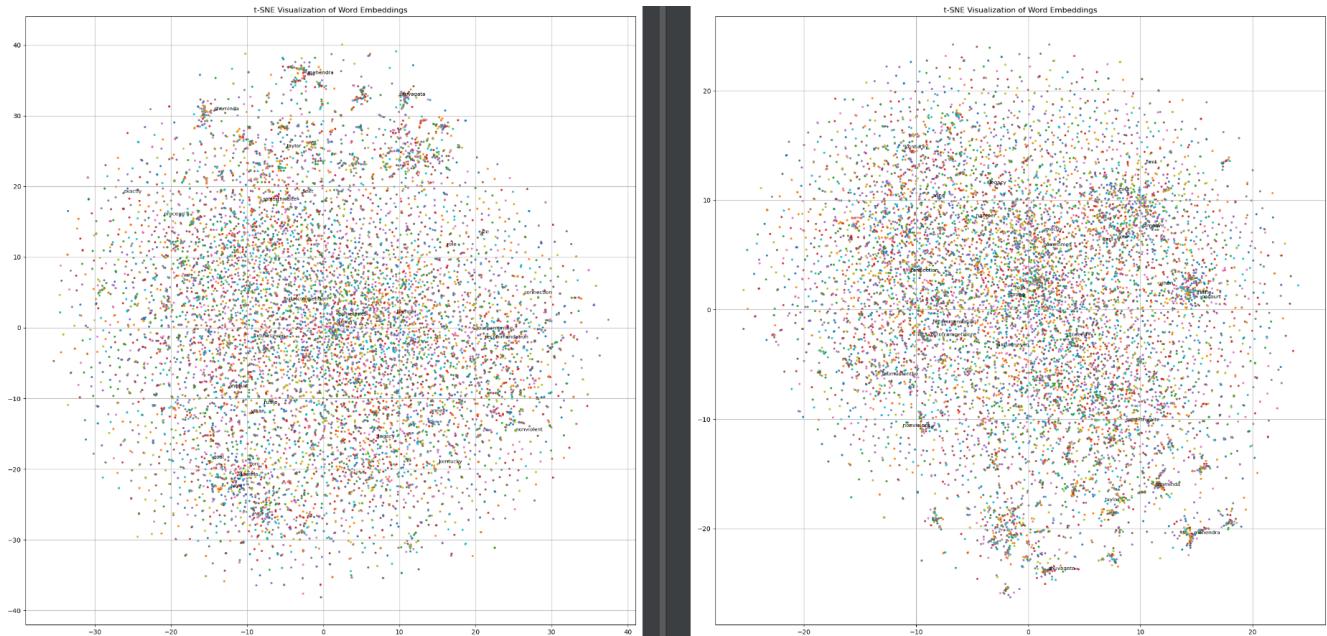


Figure 4.7: Skip-gram (batch 256, learning rate 0.01) window size 2 (left) vs Skip-gram (batch 256, learning rate 0.01) window size 5 (right)

Same window size (10) below, however, this tests extremes, a small minibatch of 32 with a small learning rate of 0.0001 produces a big range of values, with a large central cluster and some distinct clusters at the edges. A large batch of 256 with a larger learning rate of 0.05 produces smaller data ranges, and 2 large clusters at the centre.

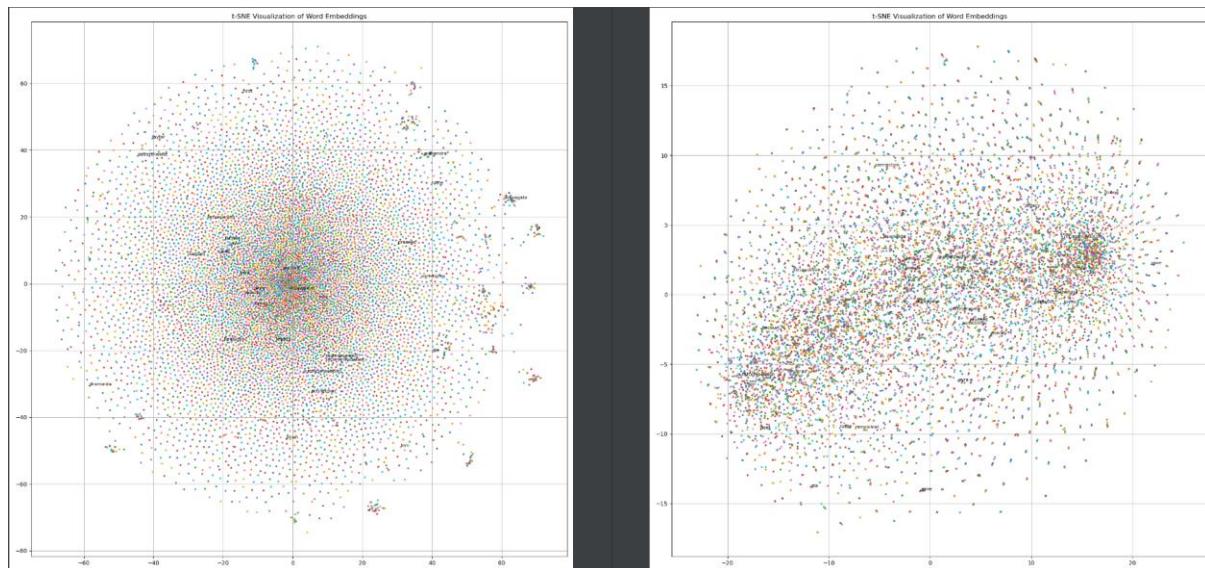


Figure 4.8: Skip-gram (batch 32, learning rate 0.0001) 10 window size (left) vs skip-gram (batch 256, learning rate 0.05) window size 10 (right)



Figure 4.9: CBOW (batch 256, learning rate 0.01) window 2 (left) vs window 5 (right)

The window size doesn't seem to affect CBOW. The same x-, y-axis ranges are there, and no clusters are immediately visible with the window.

Skipgram generic results & clusters overall vs CBOW generic results & clusters overall

- Overall, for Skipgram, increasing the window size immediately distinguishes certain clusters. However, for CBOW, decreasing the batch size directly affects getting distinguished and separated clusters.

A further analysis per category

To analyse further by category, consider **Skip-gram (batch 256, learning rate 0.01), window size 2.**

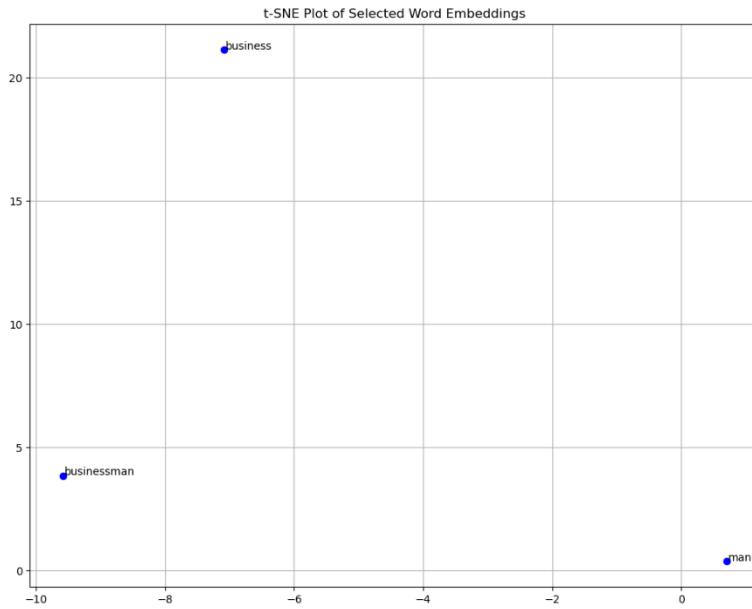


Figure 4.10: The business category, for **Skip-gram (batch 256, learning rate 0.01), window size 5.**

In the figure above, it can be seen that skip-gram makes a clear distinction between the three words. But this alone is not enough to deduce the effectiveness of the word embeddings.

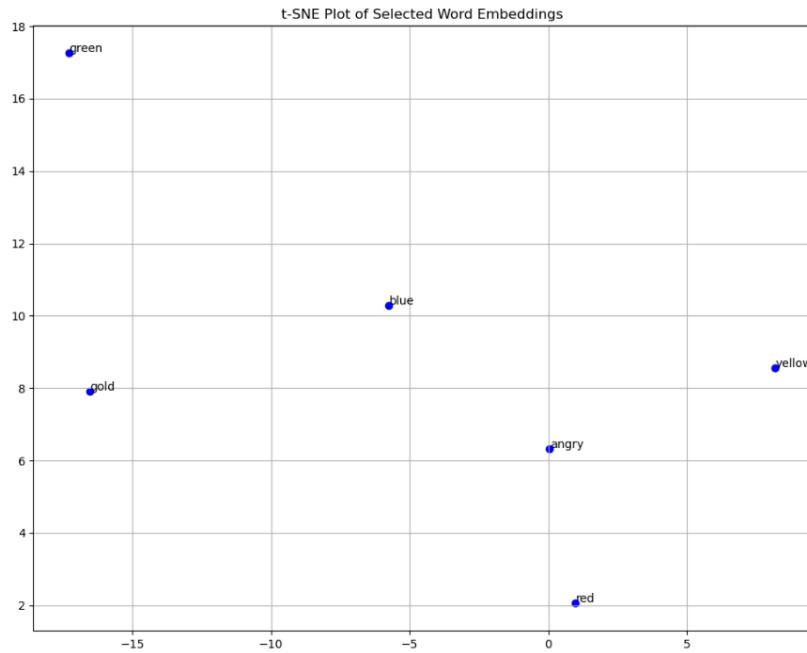


Figure 4.11: The colour category for **Skip-gram (batch 256, learning rate 0.01), window size 5.**

Different colours above were put for this category, and "angry" was inserted in. Anger is often associated with "seeing red". It was interesting that of all the words, "angry" seems closest to the "red" colour.

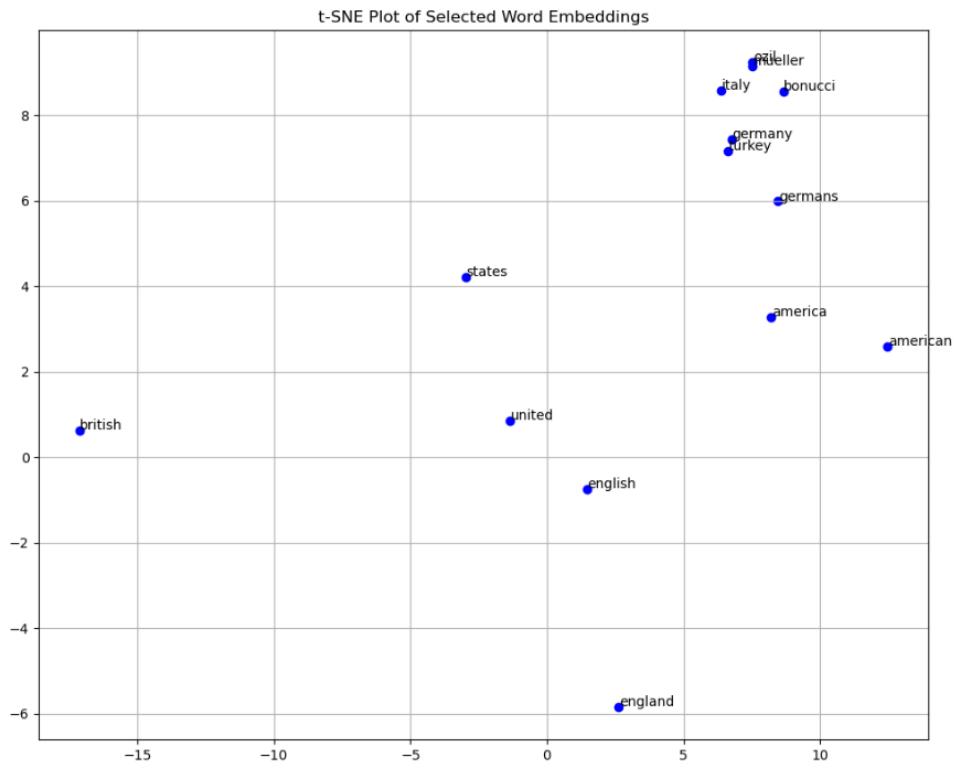


Figure 4.12: The country category for Skip-gram (batch 256, learning rate 0.01), window size 5.

The country category contained keyword pairs related to countries and 3 football players. Ozil is half german, half turk and was a German National football player, Mueller is a German National football player and Bonucci is an Italian national player.

It is interesting to see that “united”-“states”, “America”-“American”, “English”-“England”, “united”-“British”, “united”-“english”, “Germany”-“germans” for the countries are all immediate neighbours. “ozil” and “mueller”, two star german players have very close vector embeddings, and have “bonucci” close to them. “Italy” and “bonucci” are also similarly close to each other. This shows that meaningful clusters were made.

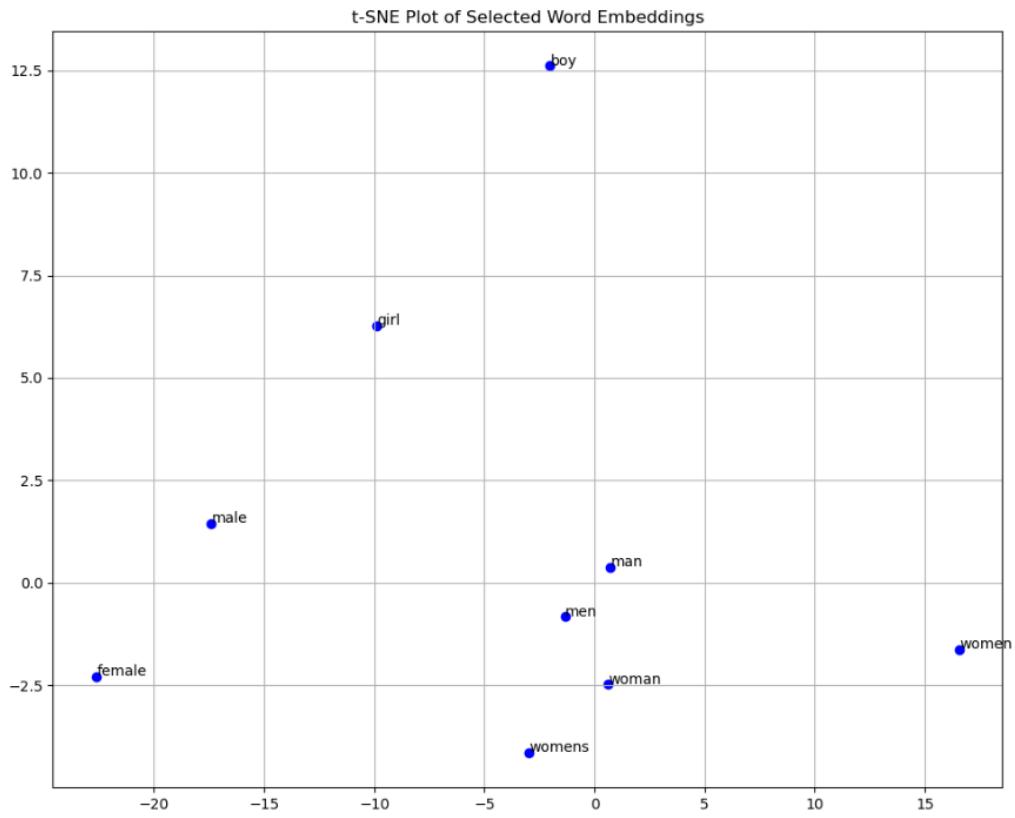


Figure 4.13: The gender category for Skip-gram (batch 256, learning rate 0.01), window size 5.

The gender category contains the same thing. “man” and “men”, “woman” and “womens” are close to each other. Opposites, or words with completely different meaning, seem to have separation like “boy” and “girl” or “male” and “female”.

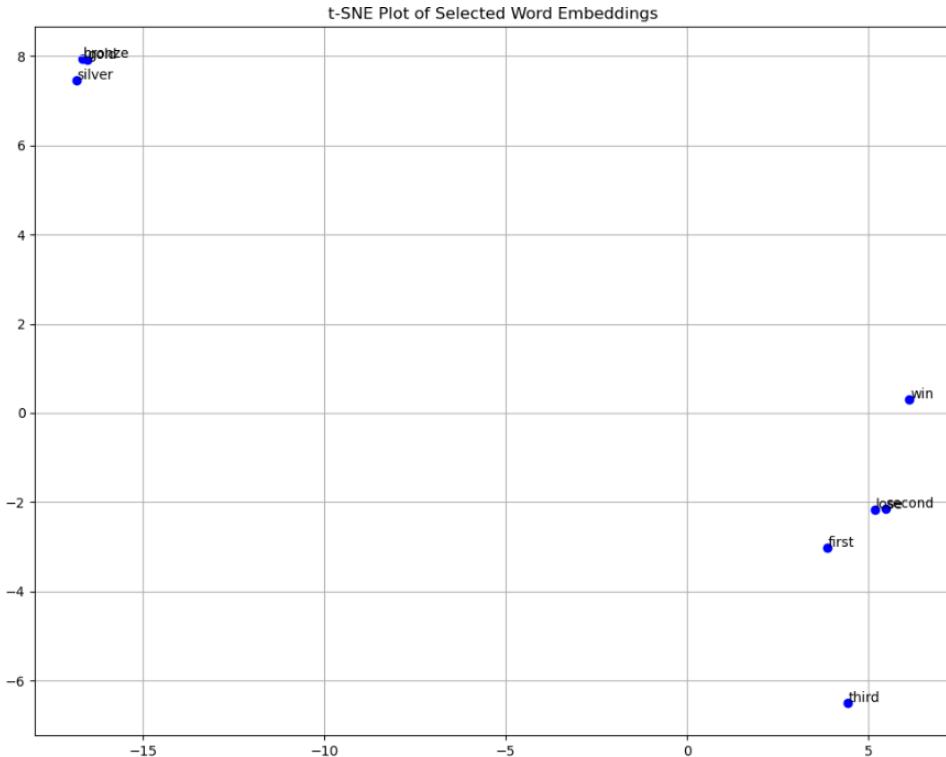


Figure 4.14: The position category for Skip-gram (batch 256, learning rate 0.01), window size 5.

As a reminder, this category contained 3 similar words “gold”, “silver”, “bronze” (which are well grouped up in the upper left corner), 2 opposite words “win”, “lose”, and “first”, “second”, “third”.

Clearly from the image above, skip-gram distinguishes the metals “gold”, “silver”, “bronze” from the first second and third positions. “first” is interestingly not associated with “win”, however, “lose” is closely related to “second”. “third” is in a plot of its own.

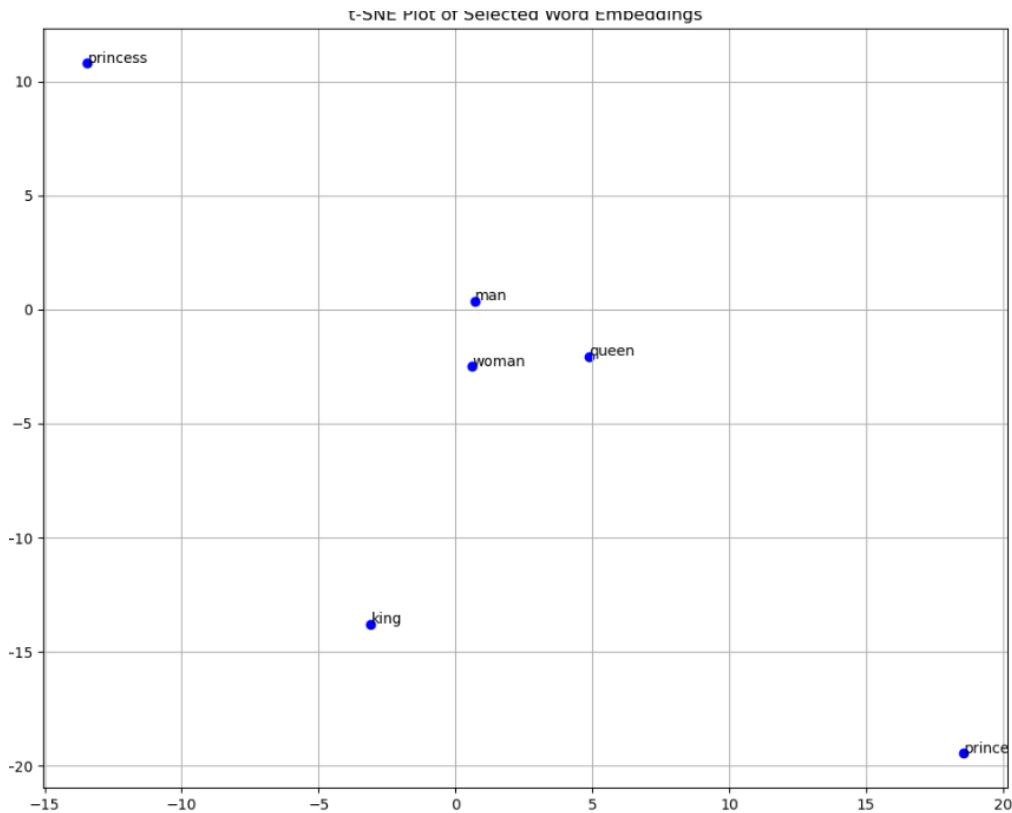


Figure 4.15: The royalty category for Skip-gram (batch 256, learning rate 0.01), window size 5.

The skip-gram model above seems to be focusing on defining opposites for this dataset. “man” seems to be singled out and nearer to “woman” than “king”. This could be because of dimensionality reduction, so the association may be clearer at higher dimensionality, or because of the shared “man” word in “wo”-“man”. “woman” and “queen” are close to each other.

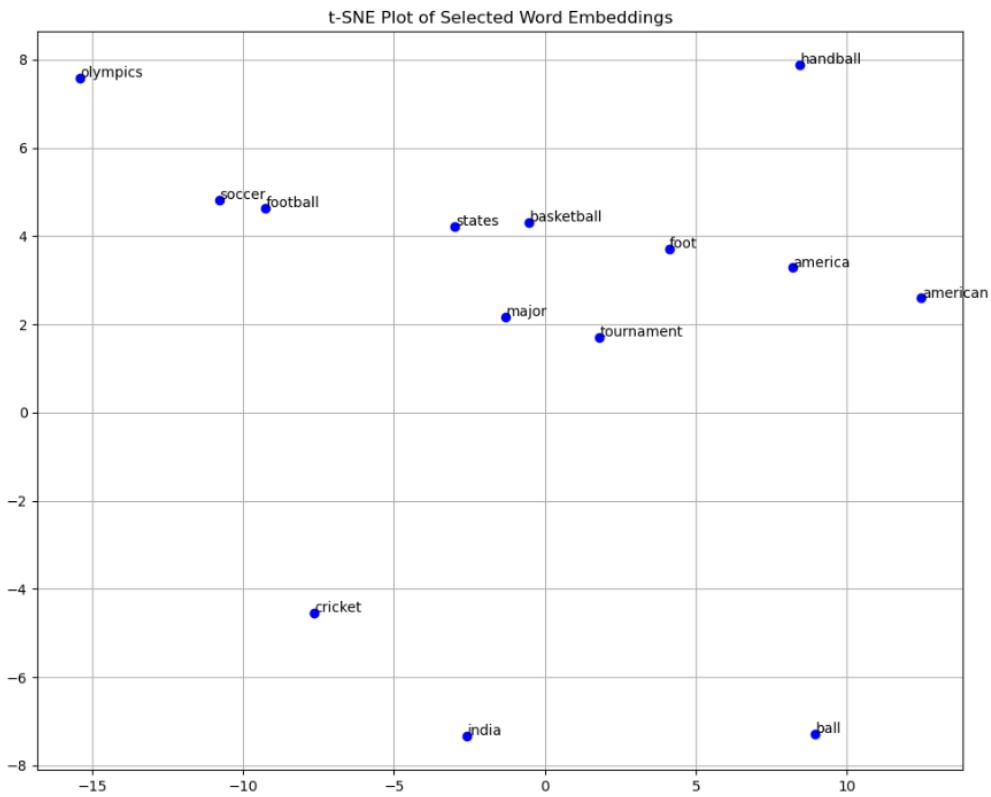


Figure 4.16: The sport category for Skip-gram (batch 256, learning rate 0.01), window size 5.

The final category analysed is sports. “America”-“American”, “soccer”-“football”, “states”-“basketball”, “immediately seem to be clustered together. “cricket”’s closest neighbour in the plot is “india”, attributed possibly to the popularity of sports articles published in the country. “foot” and “ball” seem separated despite being closely related to “football”.

Challenges Encountered

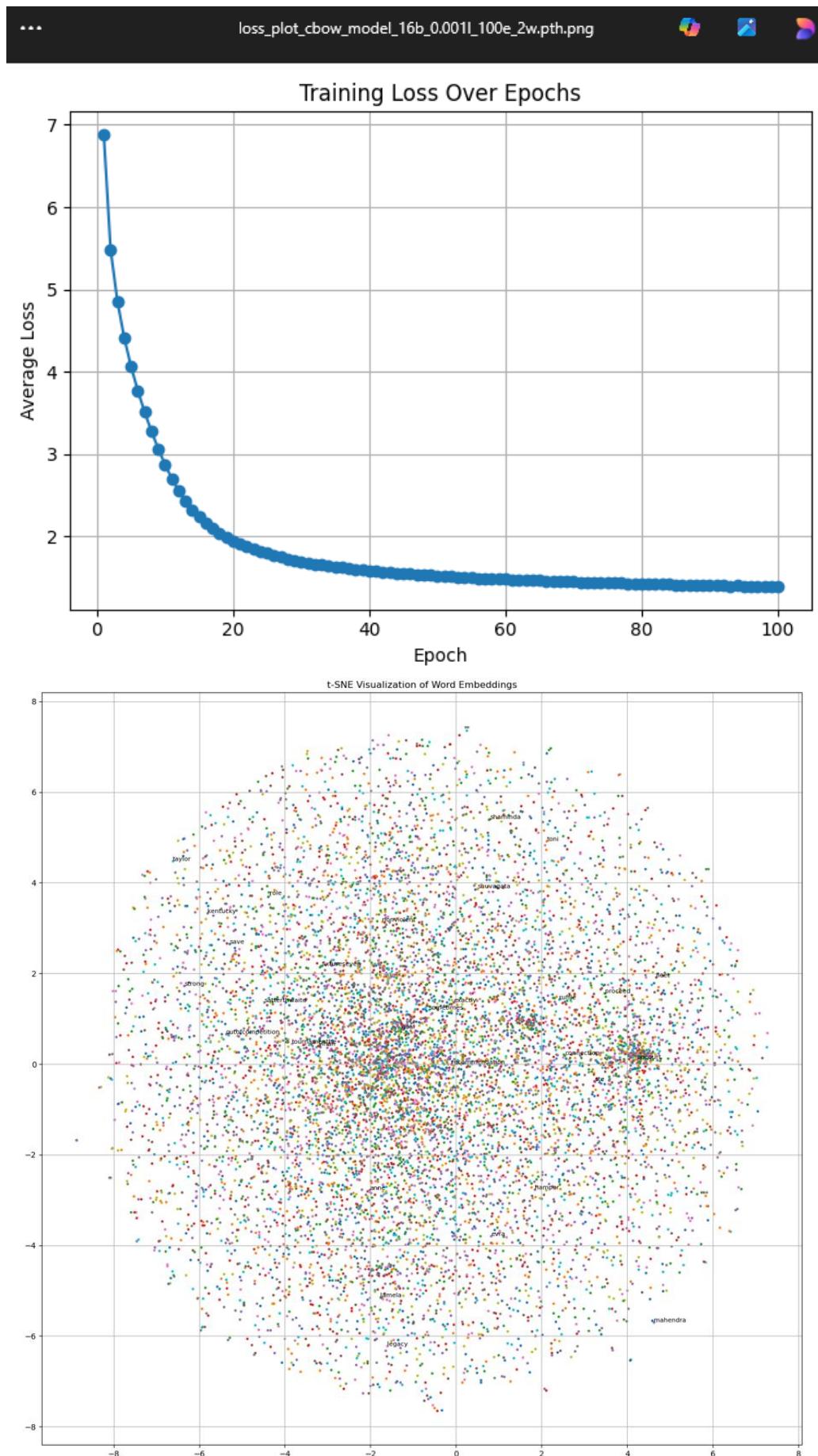
- Hyperparameter optimisation was performed using both CPUs and GPUs offered by Google Colab. However, due to time constraints, extensive tuning/hyperparameter optimisation was not possible.
- Establishing the right epoch size/batch size/context window size proved to be a challenge
- Interactive tSNE plots (using plotly package) and kNN visualisations were planned. Due to the size of the vocabulary and hardware requirements, this was not achieved, and resulted in errors
- Evaluating the best-performing model was also a challenge, as consistent and meaningful comparison metrics were difficult to establish.

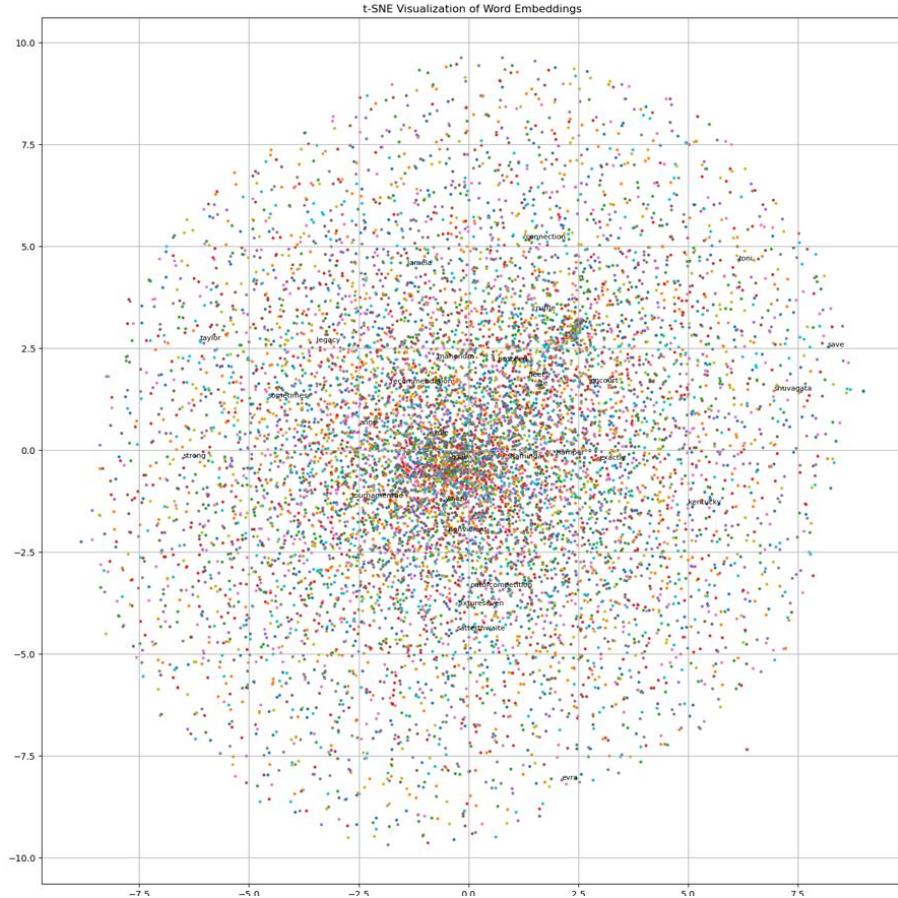
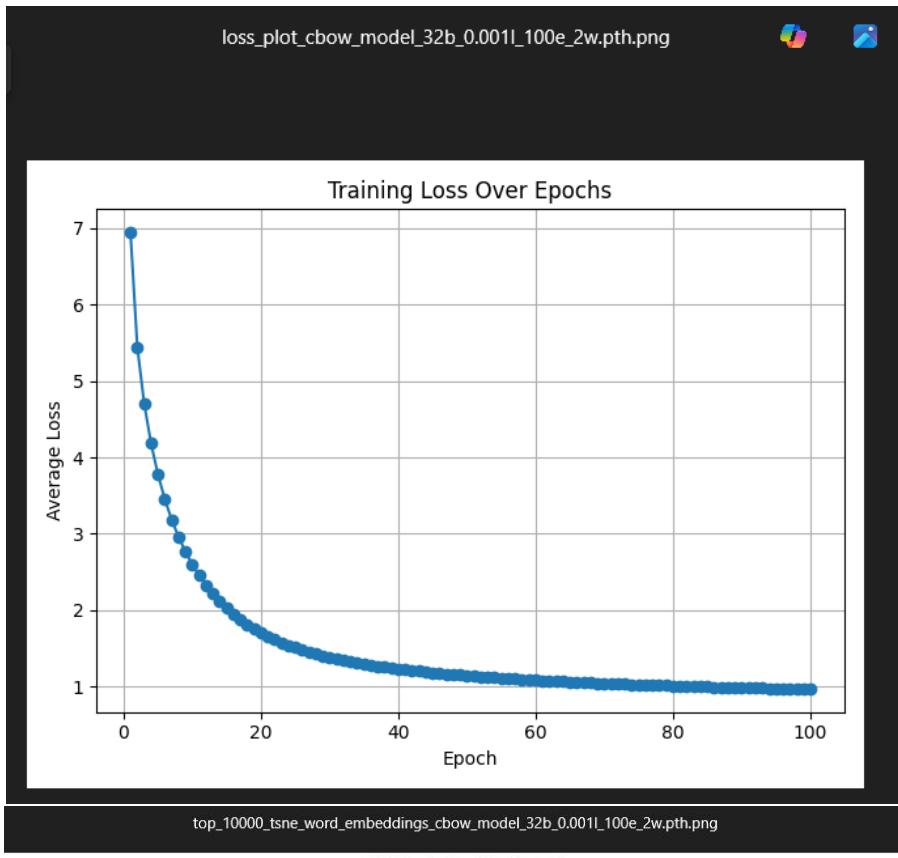
Possible Enhancements/Further work

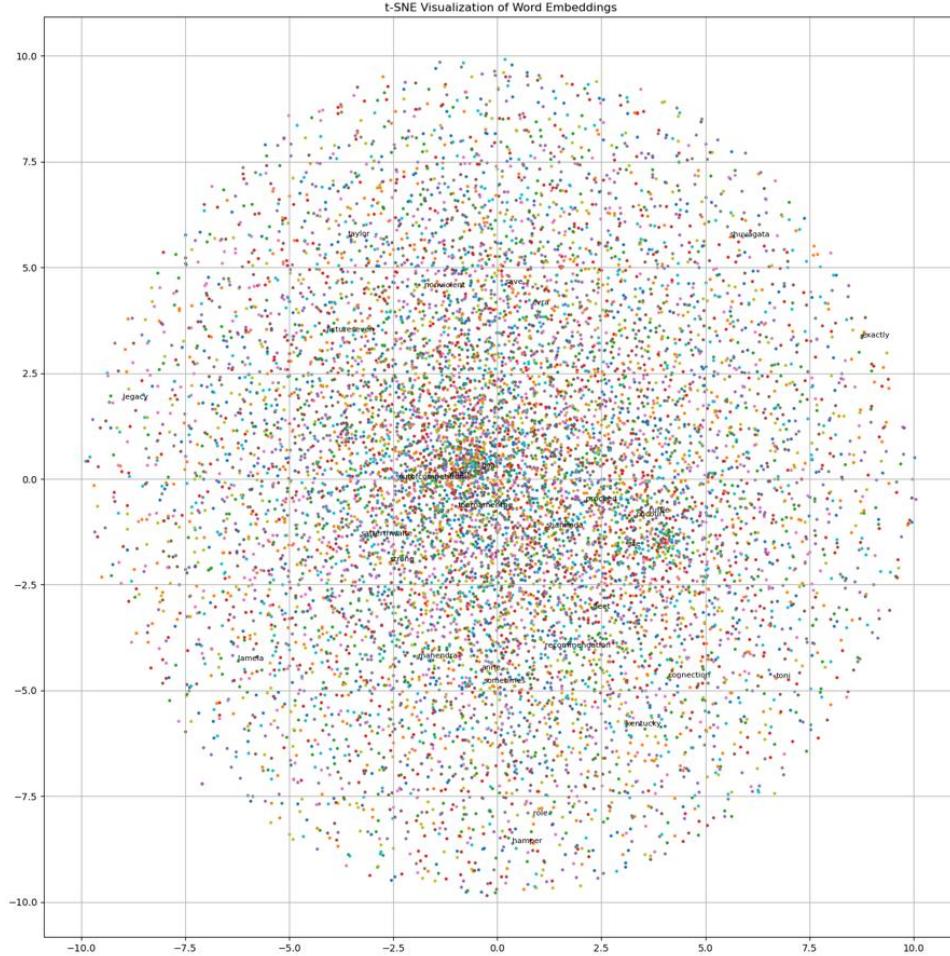
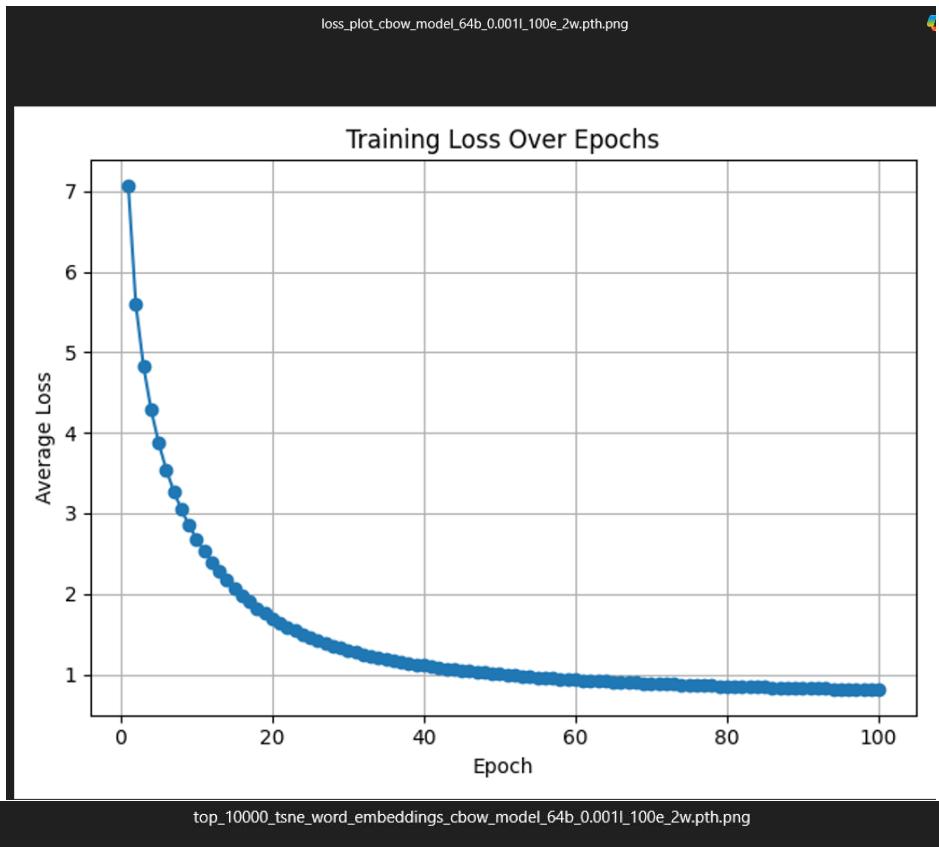
- Feed larger datasets as per the literature review (6 billion) and use benchmark datasets
- Decrease the epoch count
- Better data preprocessing regimen
 - Treat noise better and remove highly common words
- Make use of different models as GloVe or ELMo
- Make use of a different optimiser, like Adagrad, used by Mikolov et al. [4]
- Make use of a different loss function, instead of CrossEntropyLoss
- Make more use of GPUs
- Further hyperparameter experimentation:
 - Mini-batches can be made smaller for CBOW
 - Assess with a larger dataset if this separates clusters better
 - Further window sizes for skip-gram
 - Assess with a larger dataset if this separates clusters better
- Analyse quality of embeddings through word similarity and analogy evaluation, using the same datasets as the literature
- Use cluster quality metrics
- Use kNN to visualise tSNE clusters
- Interactive tSNE plots
- Analyse polysemy and homonymy

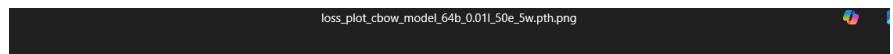
Convergence Plots & tSNE embeddings

[Next page](#)

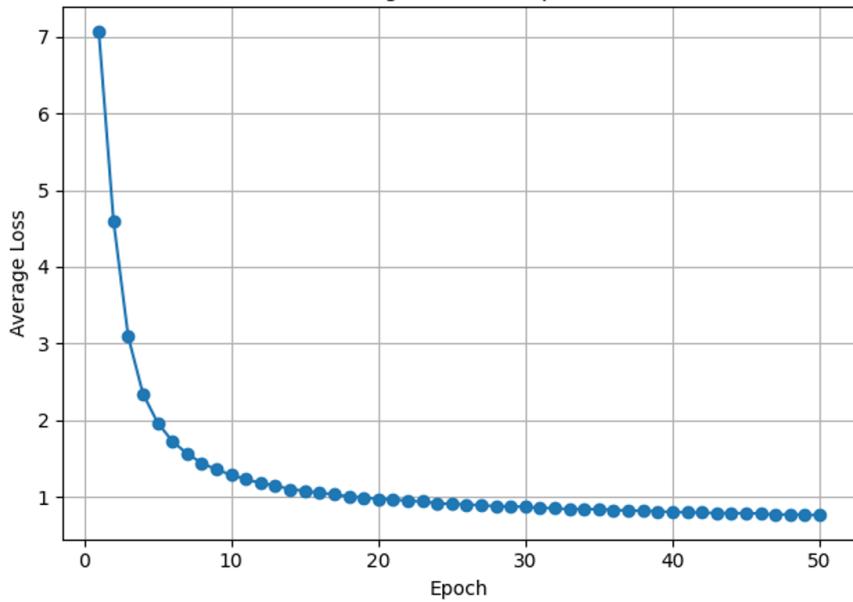




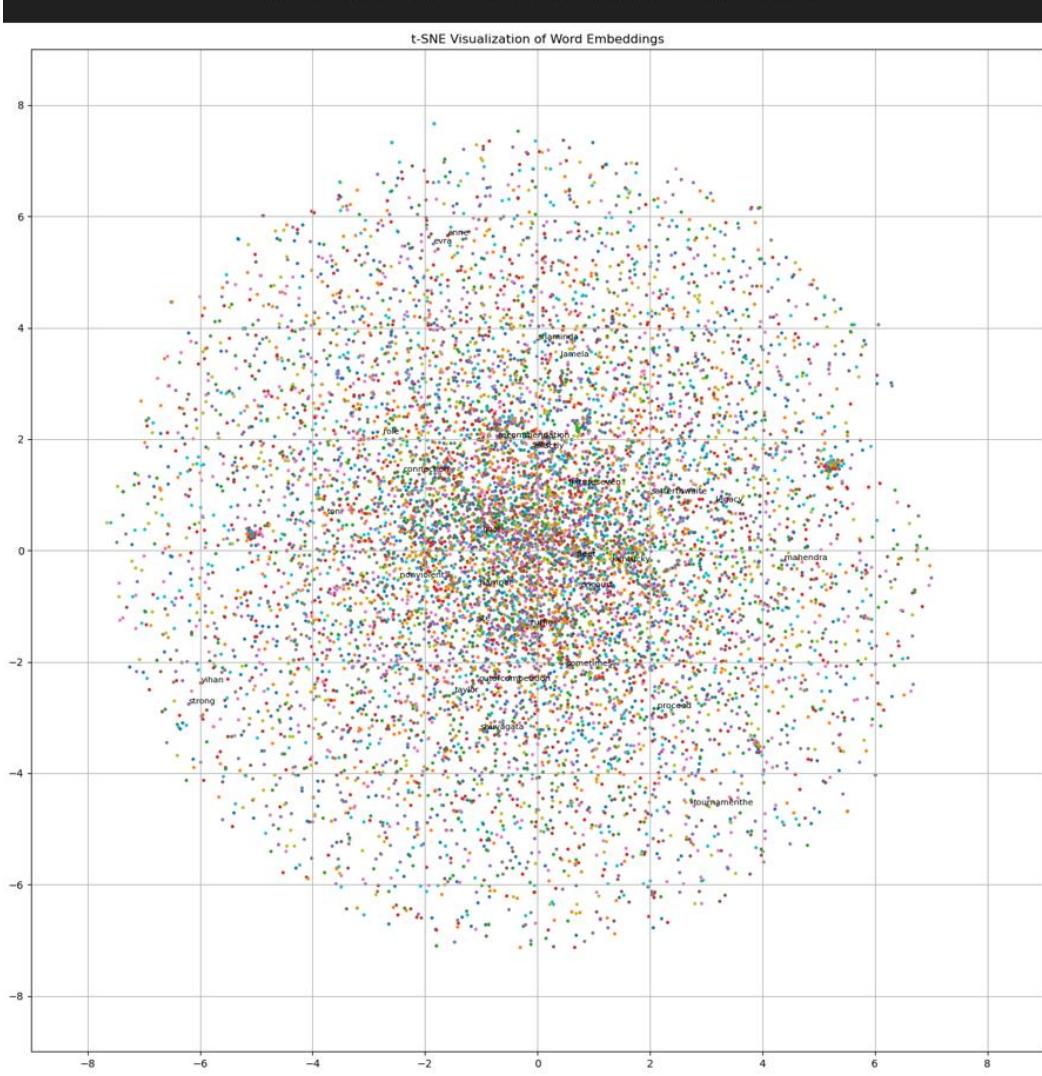


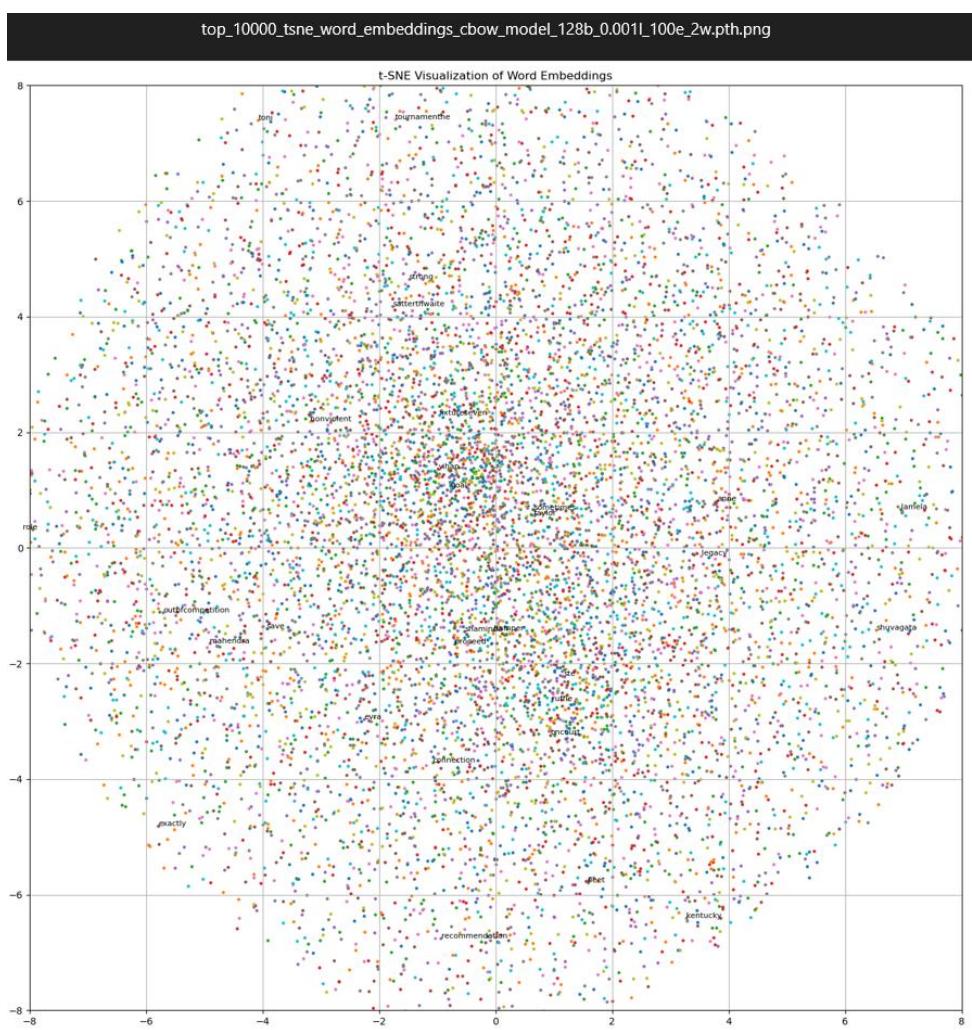
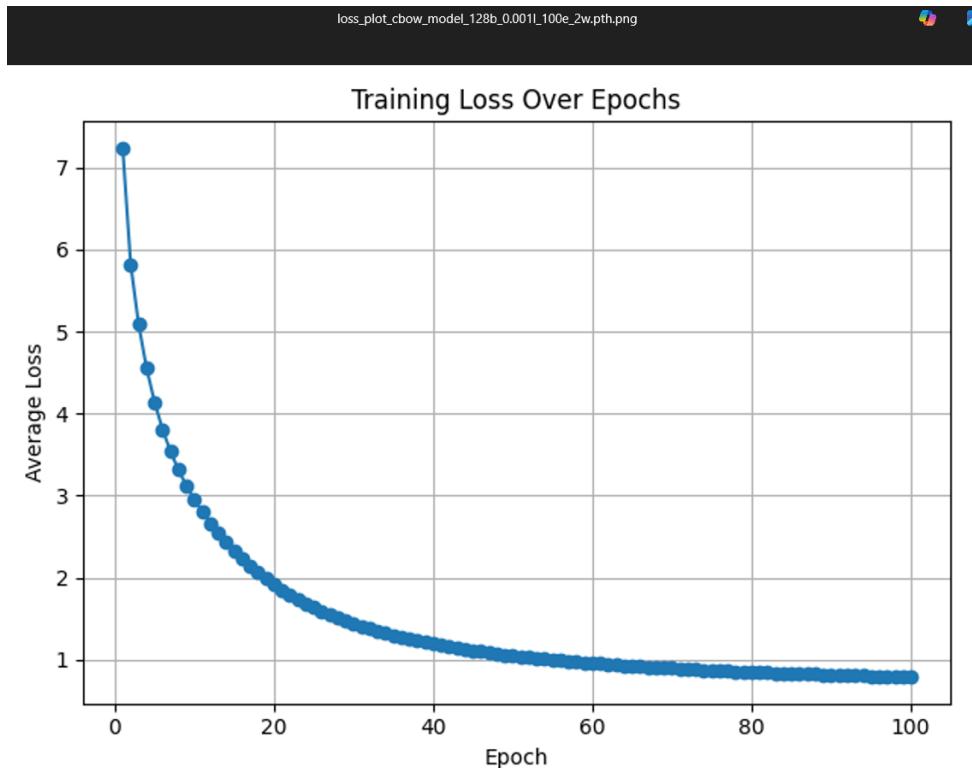


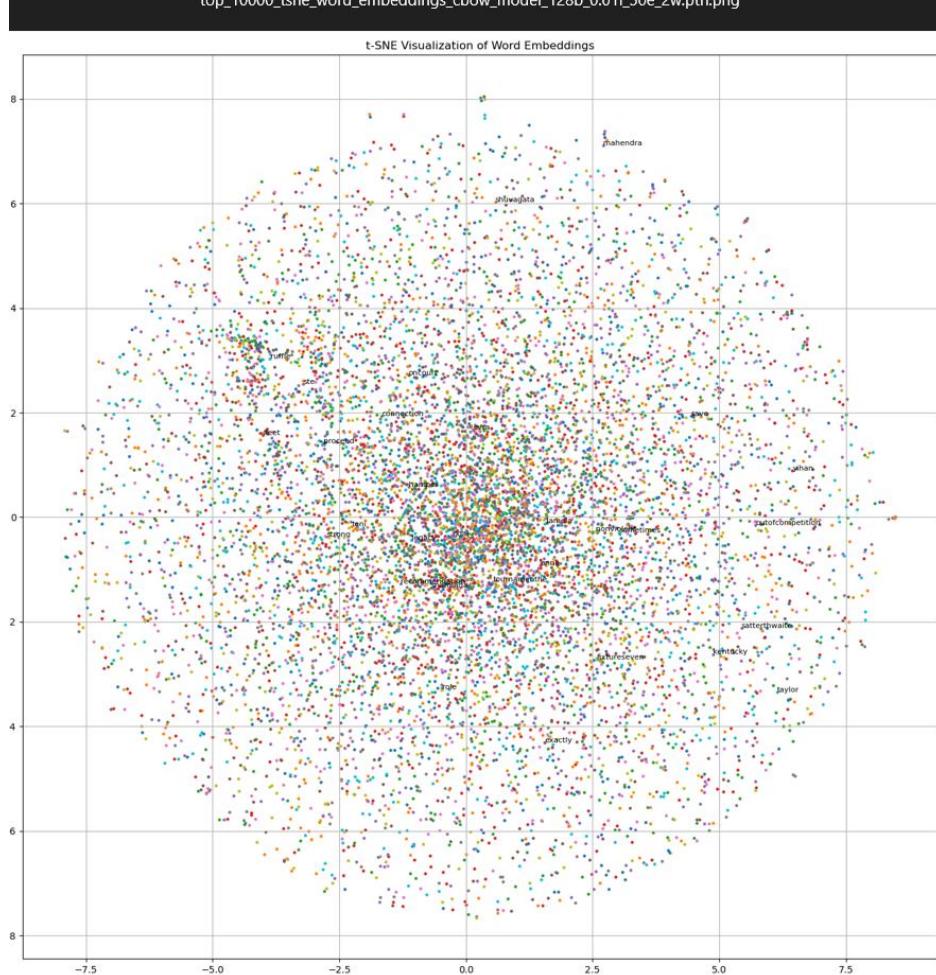
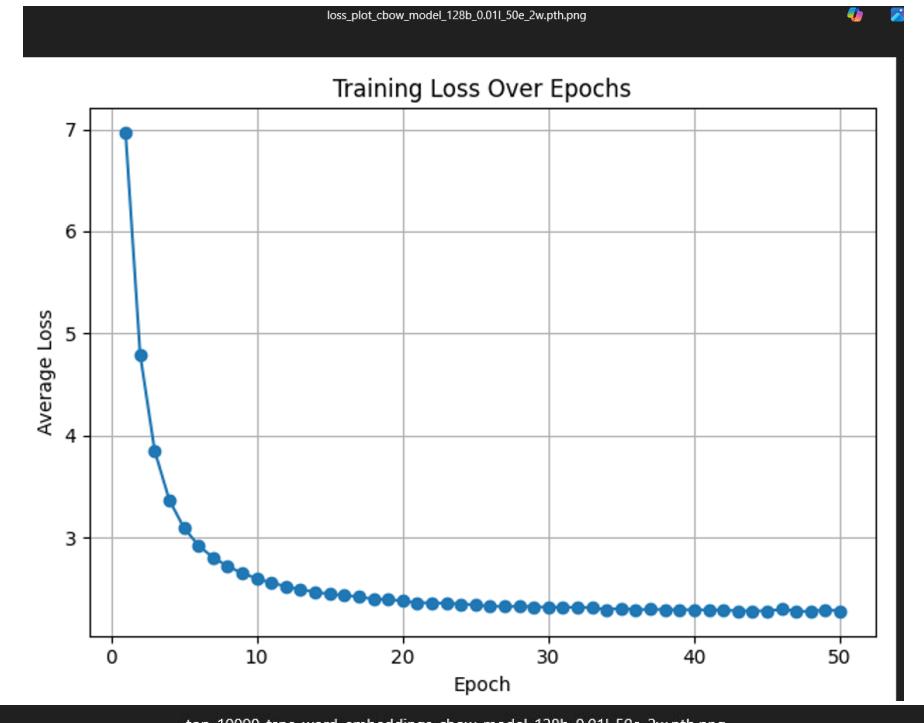
Training Loss Over Epochs

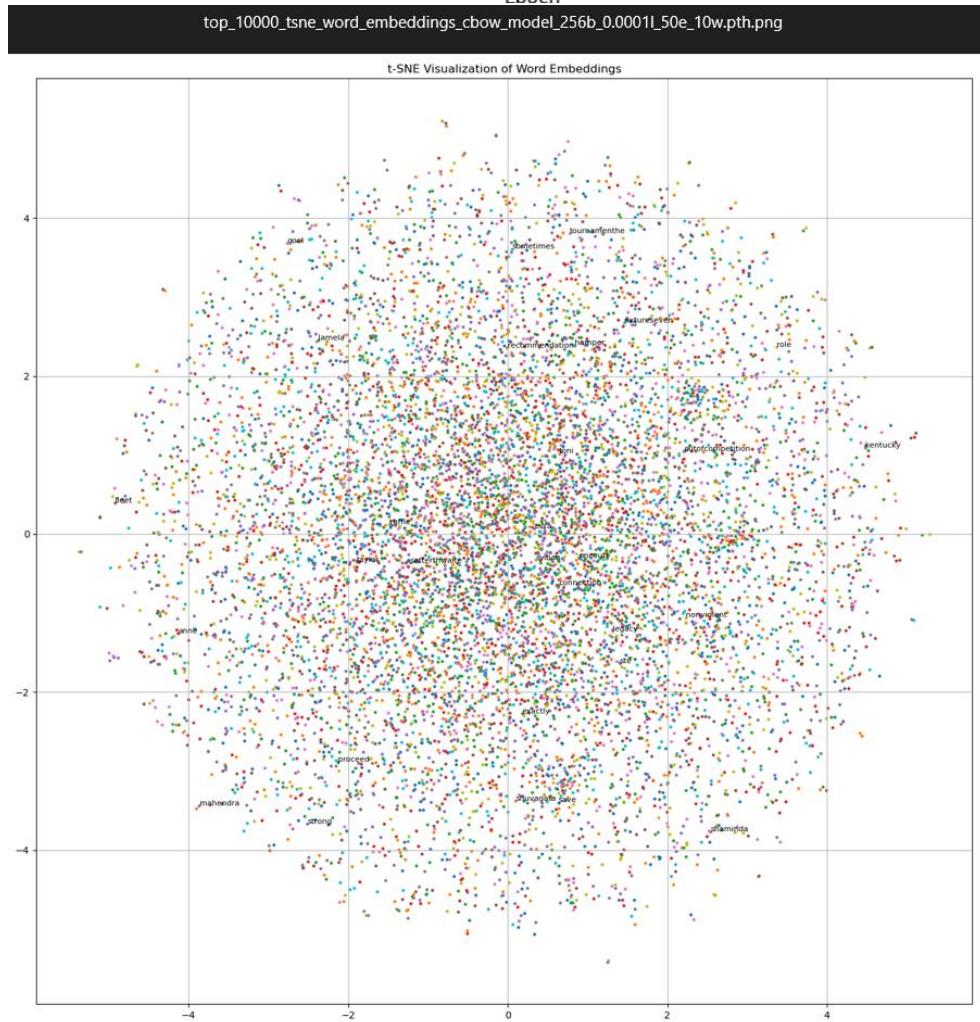
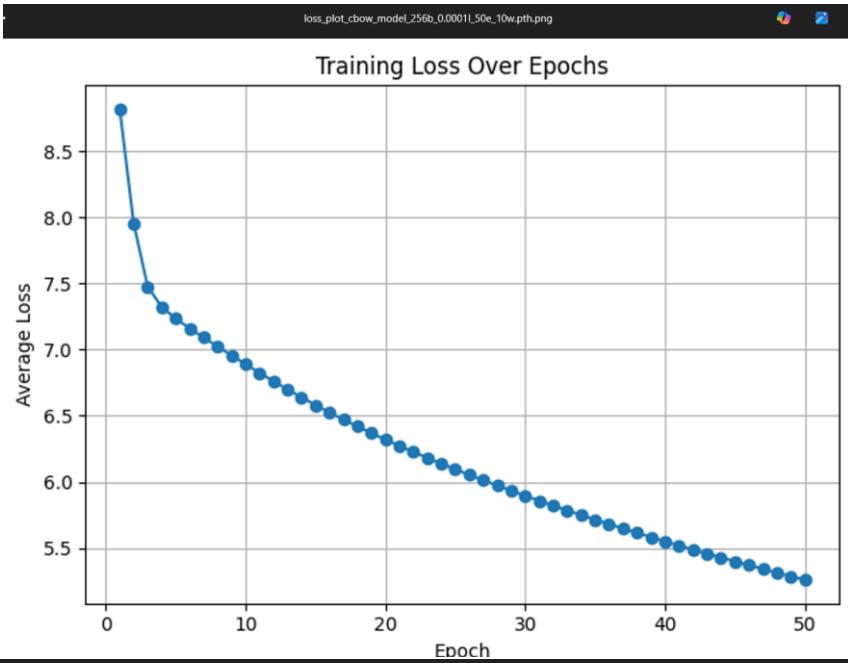


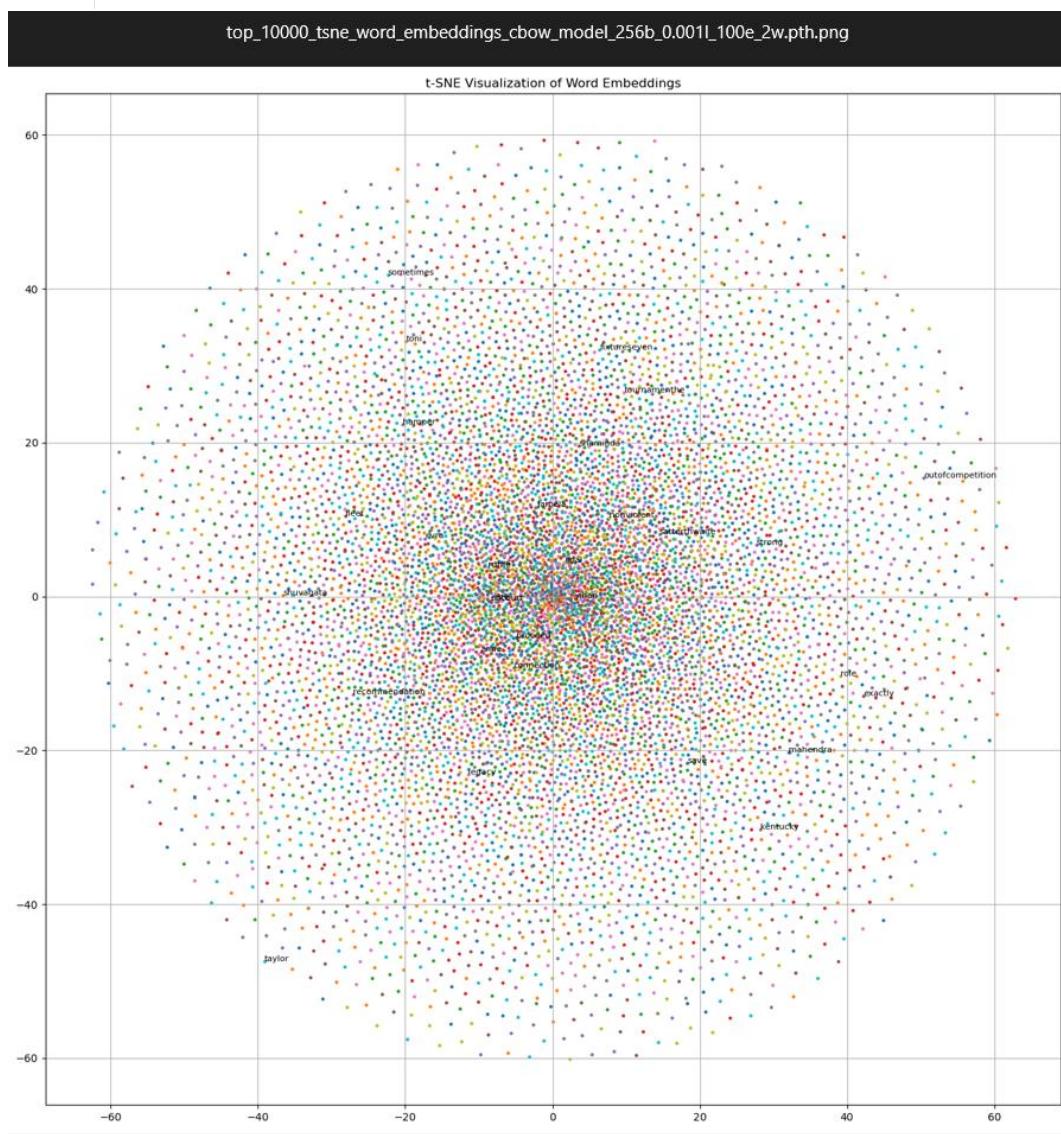
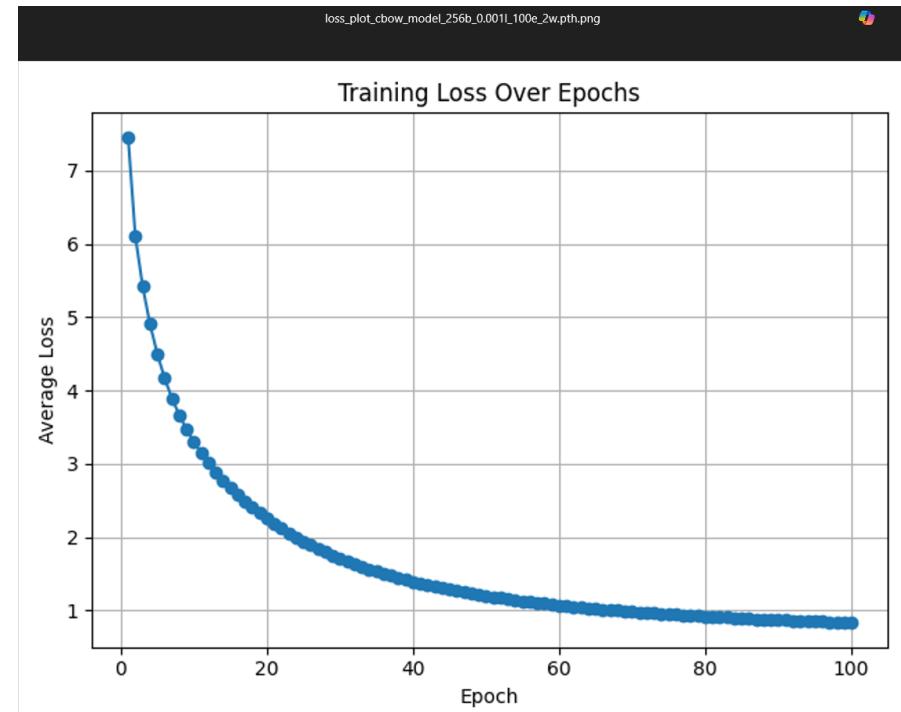
top_10000_tsne_word_embeddings_cbow_model_64b_0.01_50e_5w.pth.png

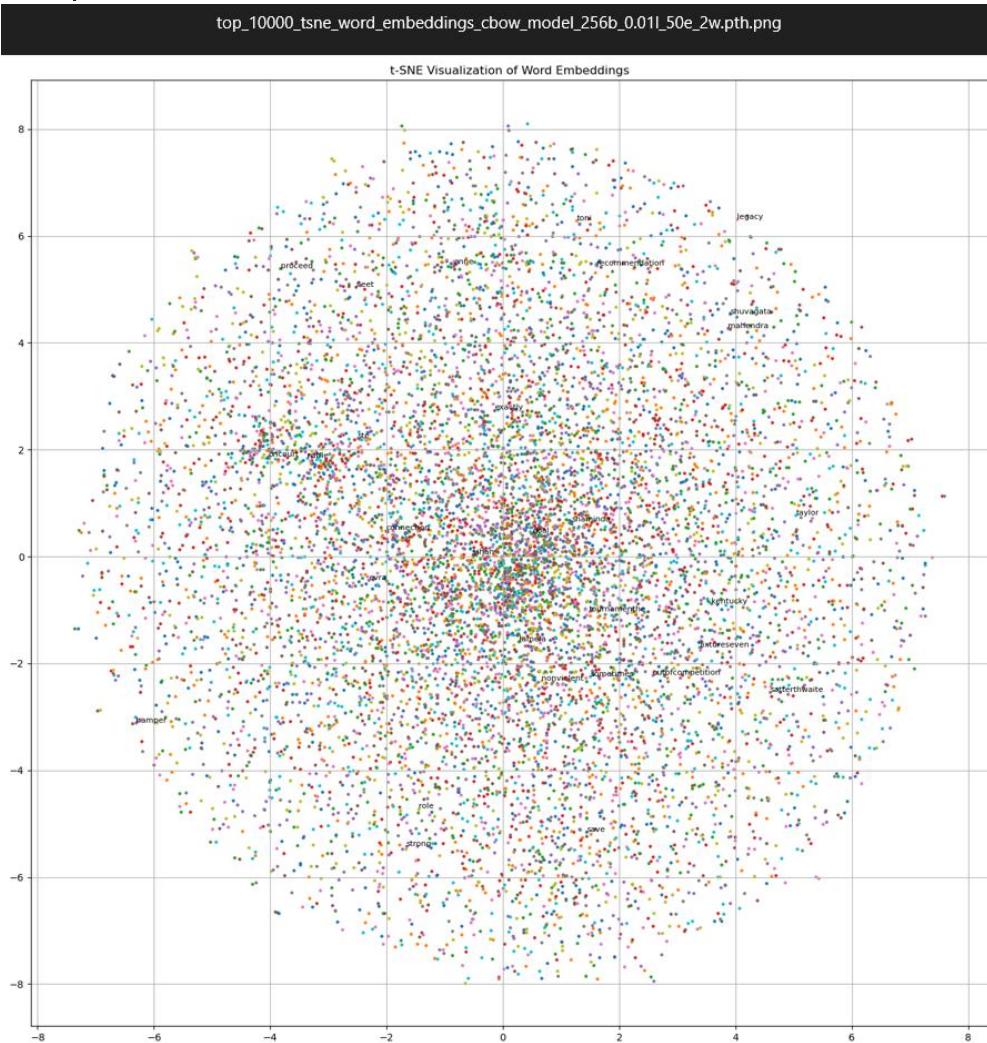
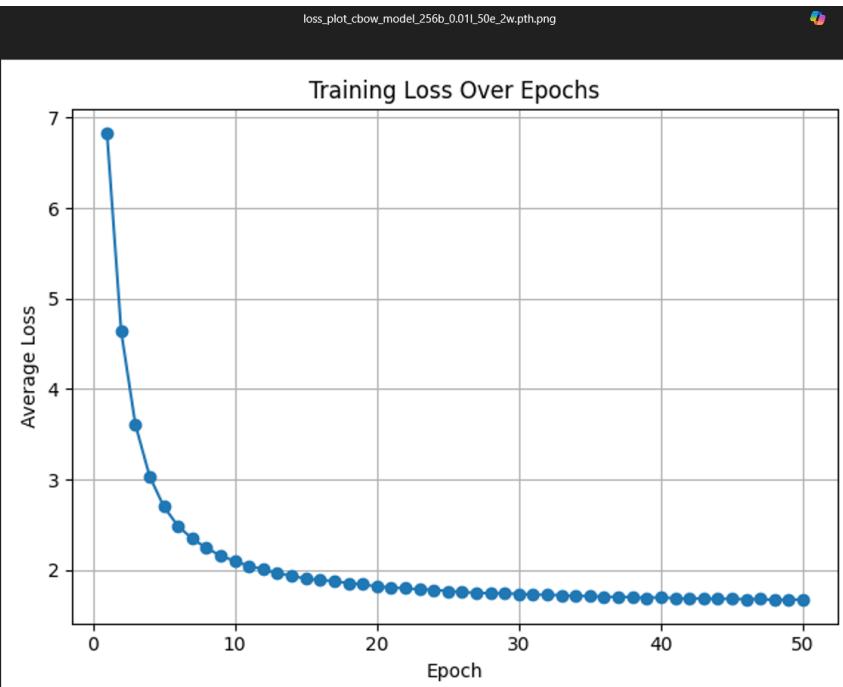


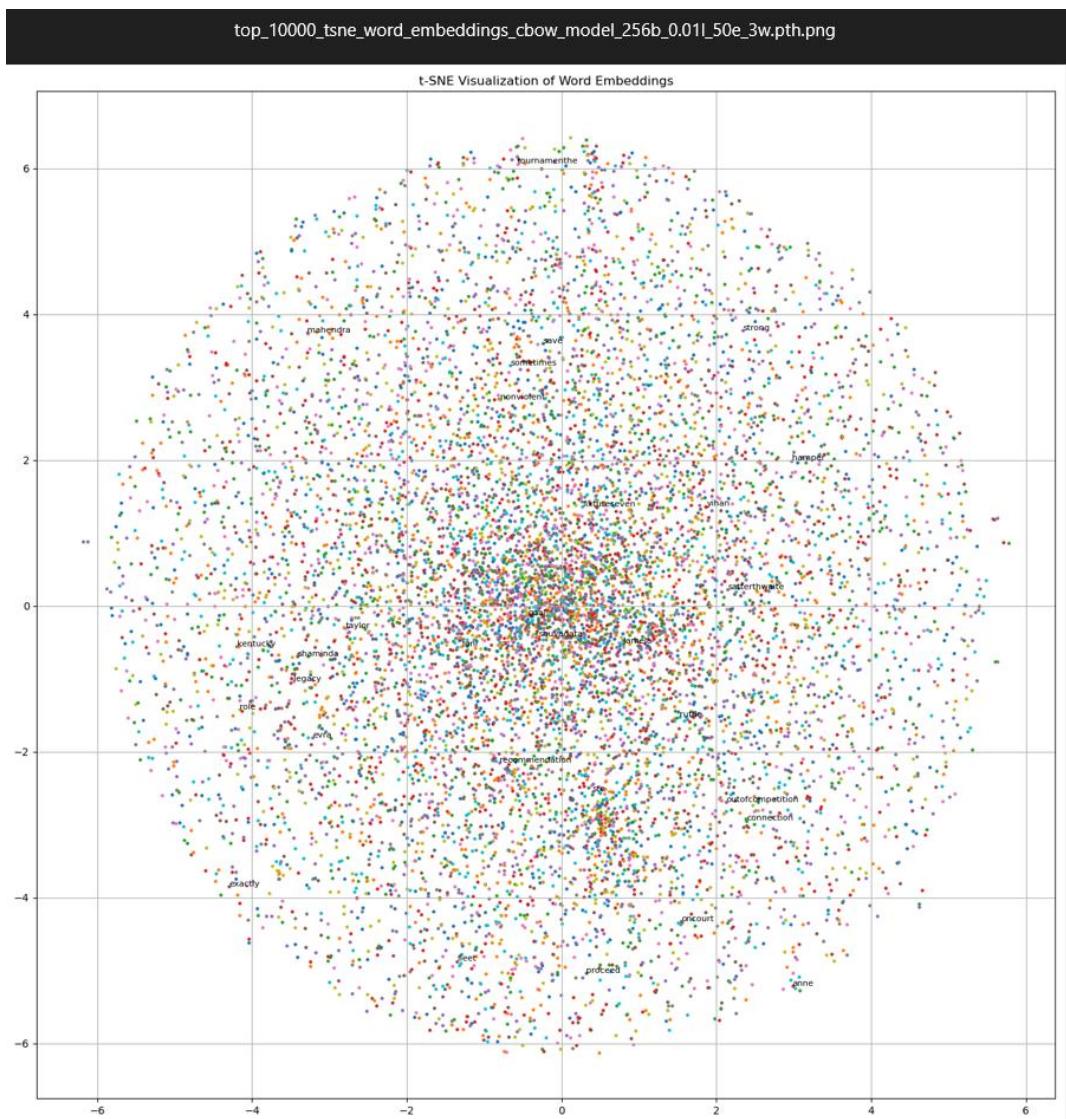
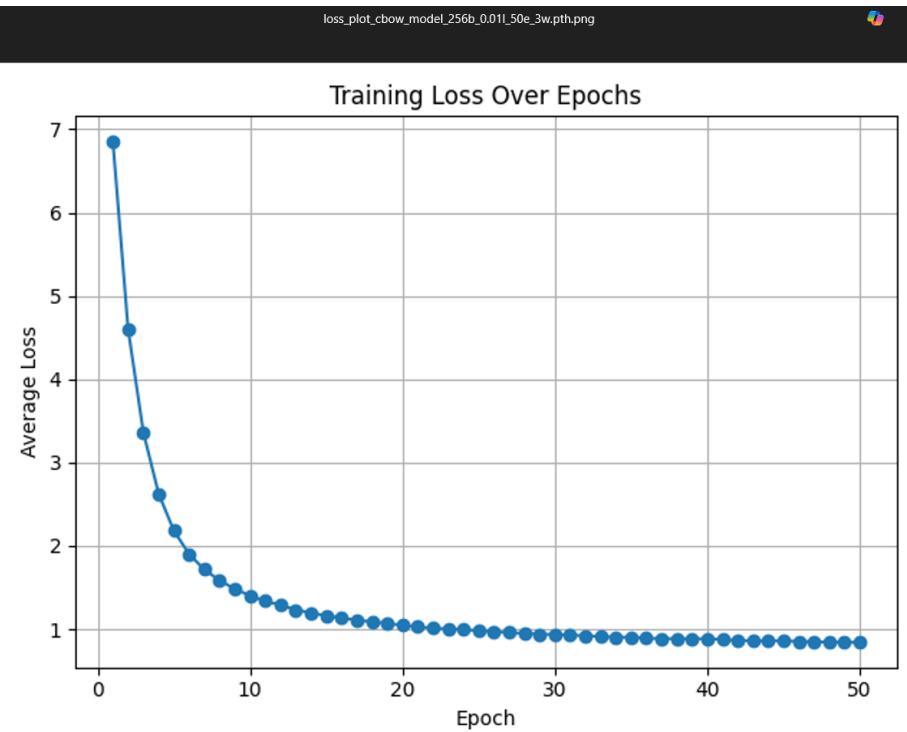


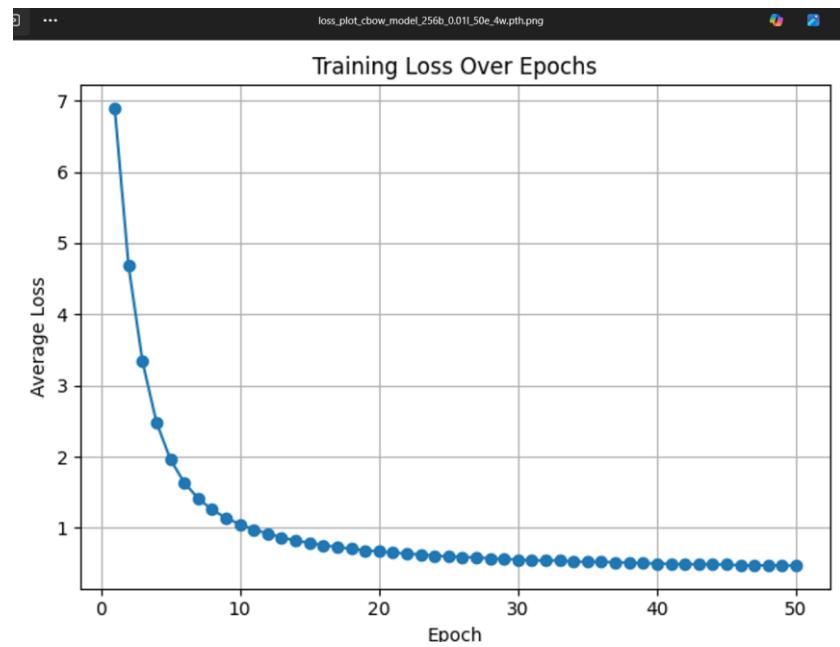


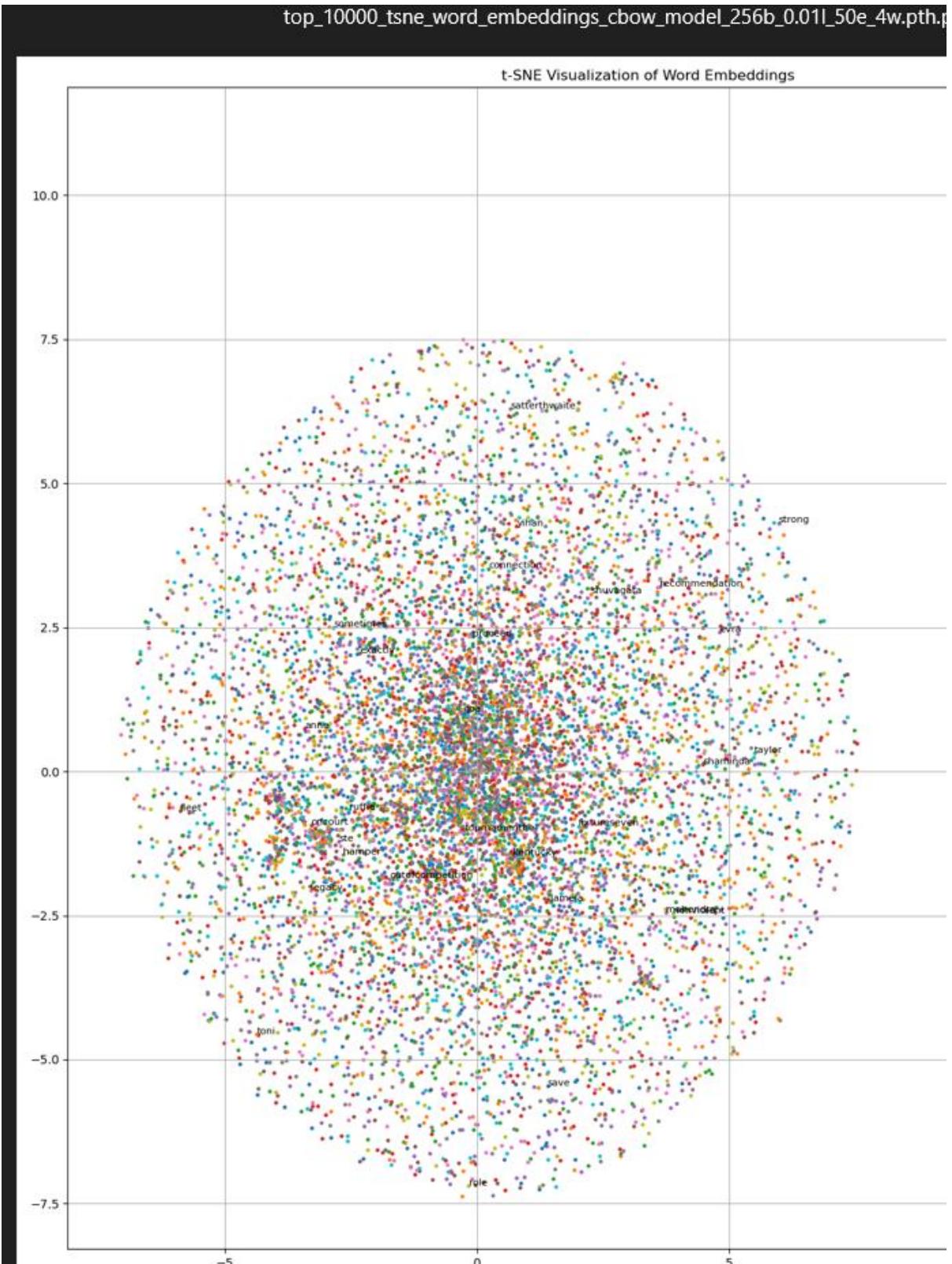


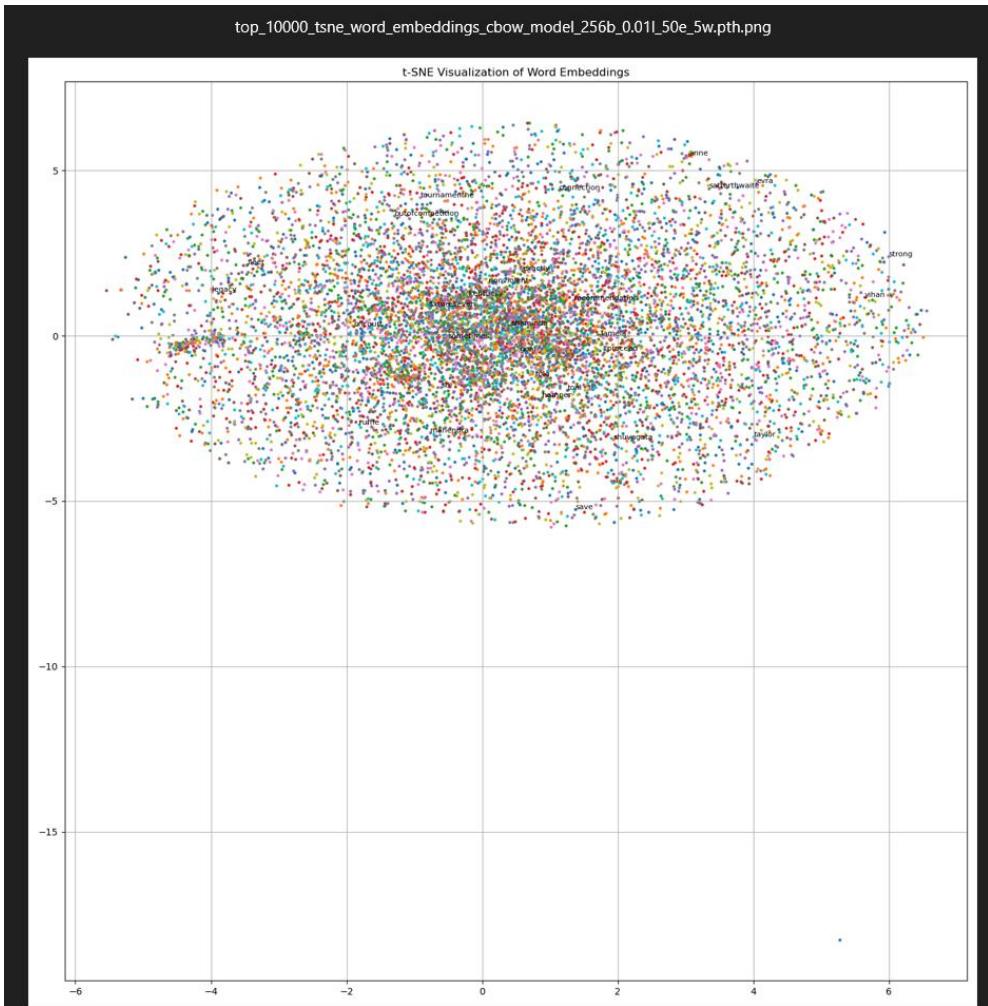
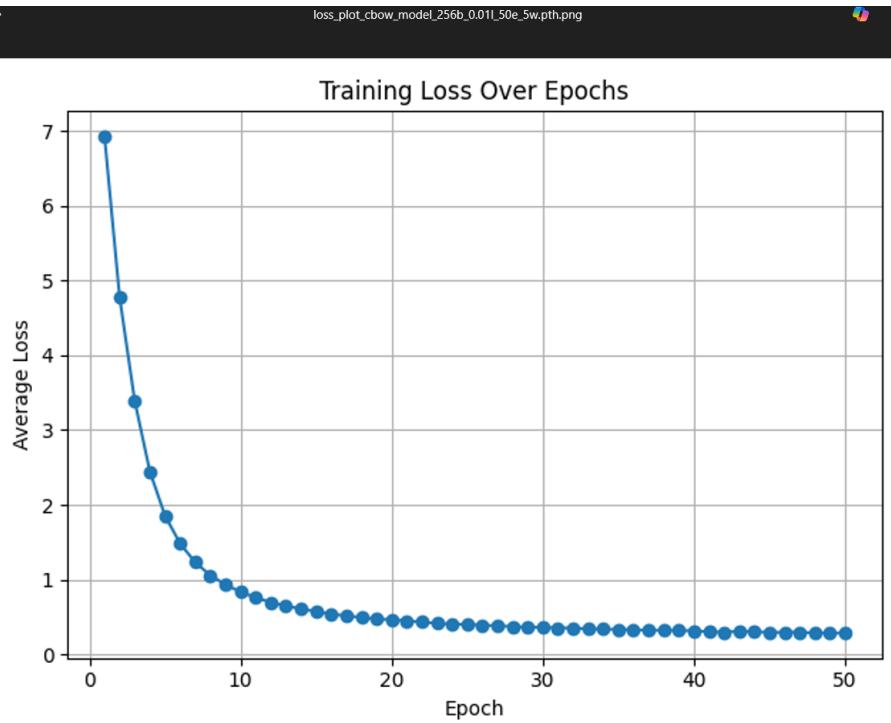


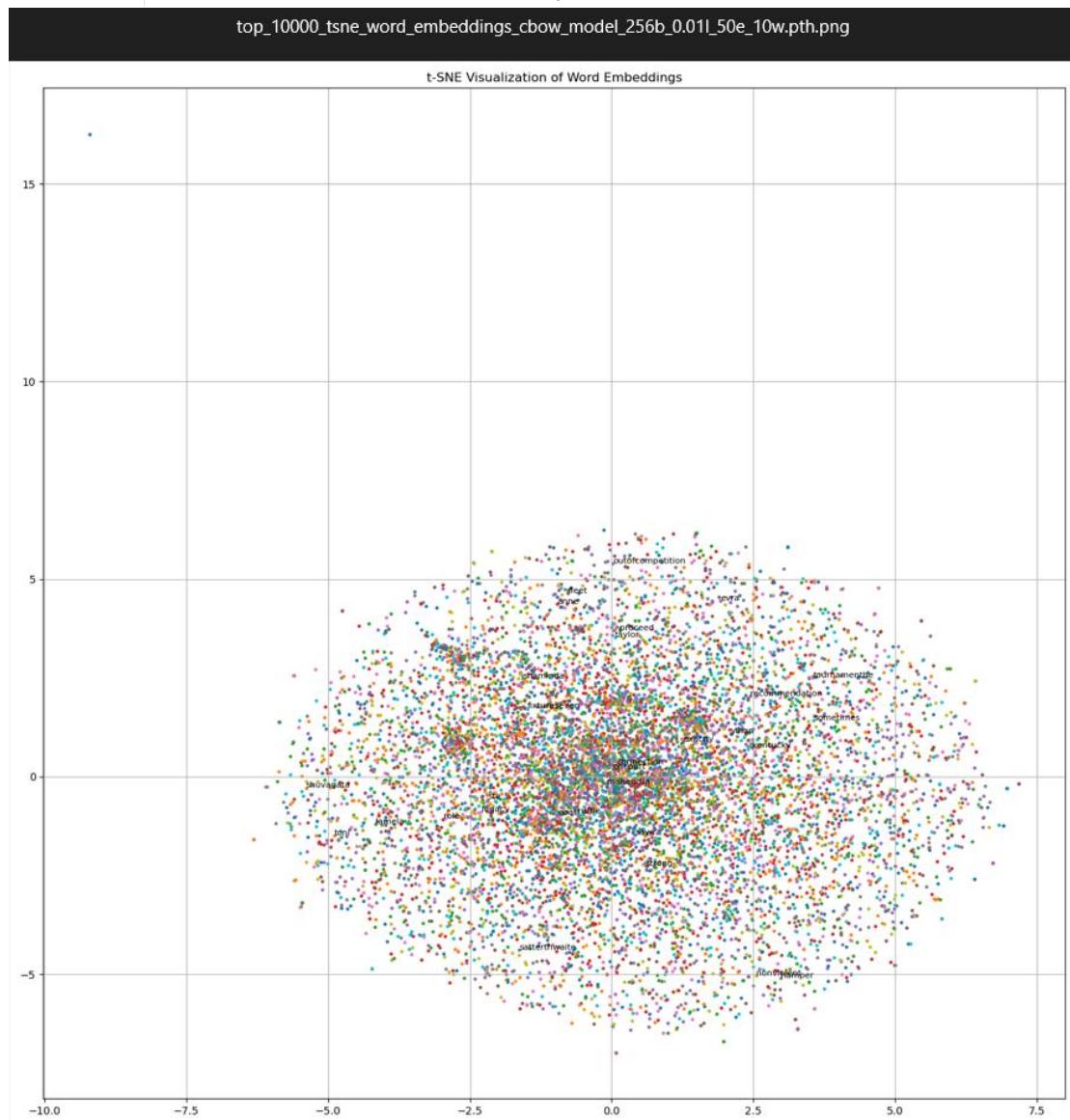
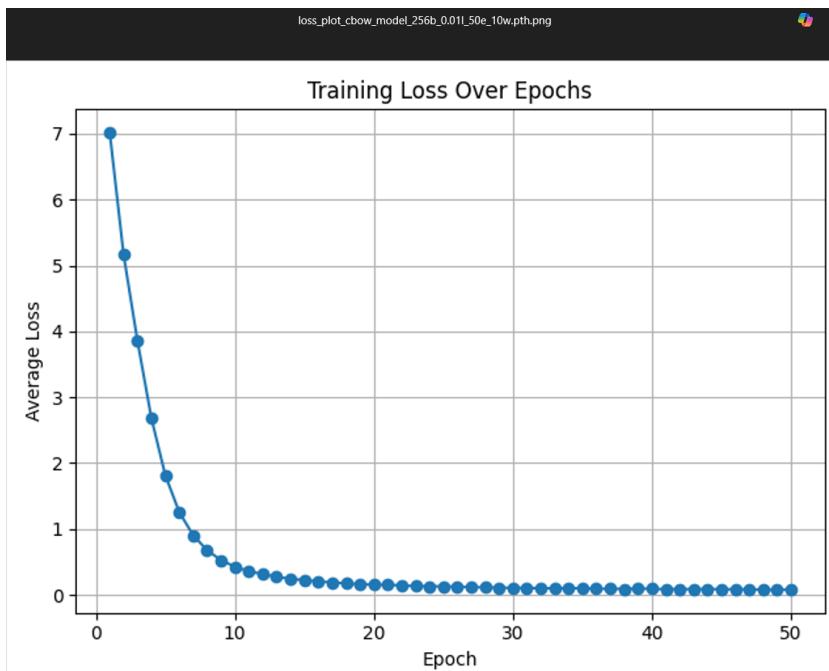


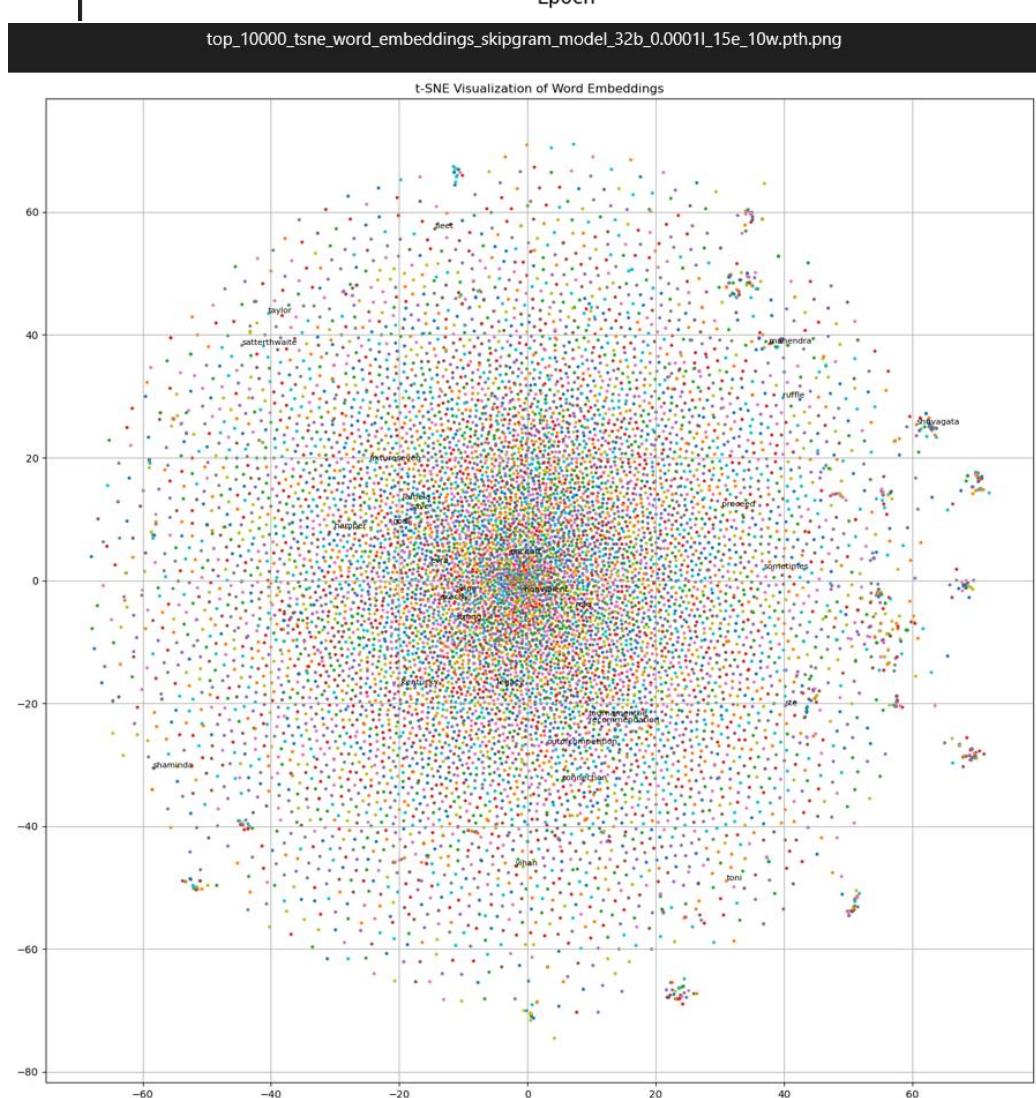
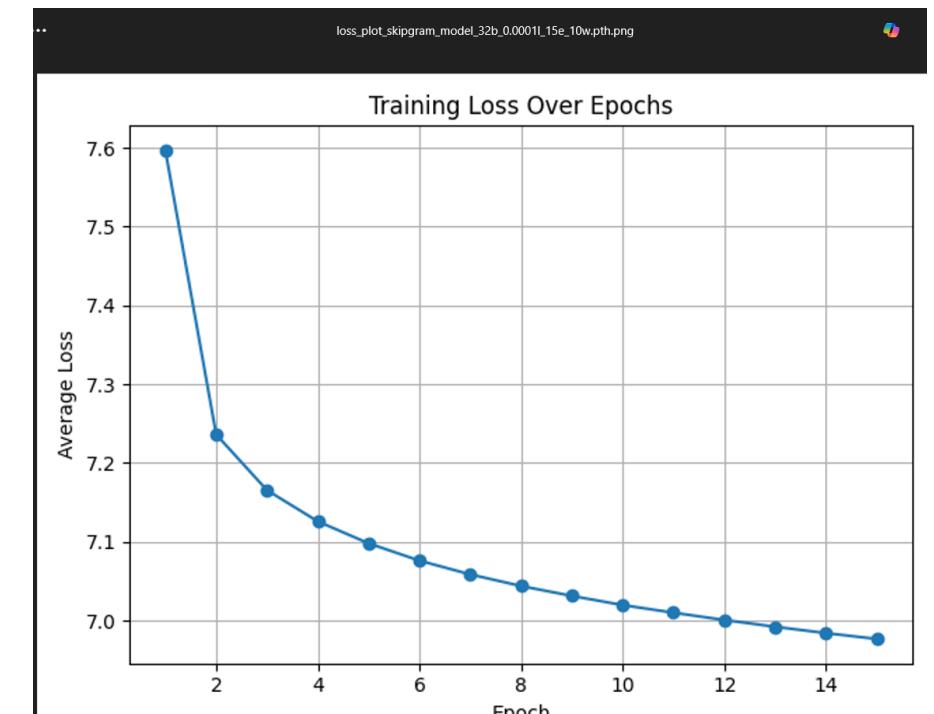


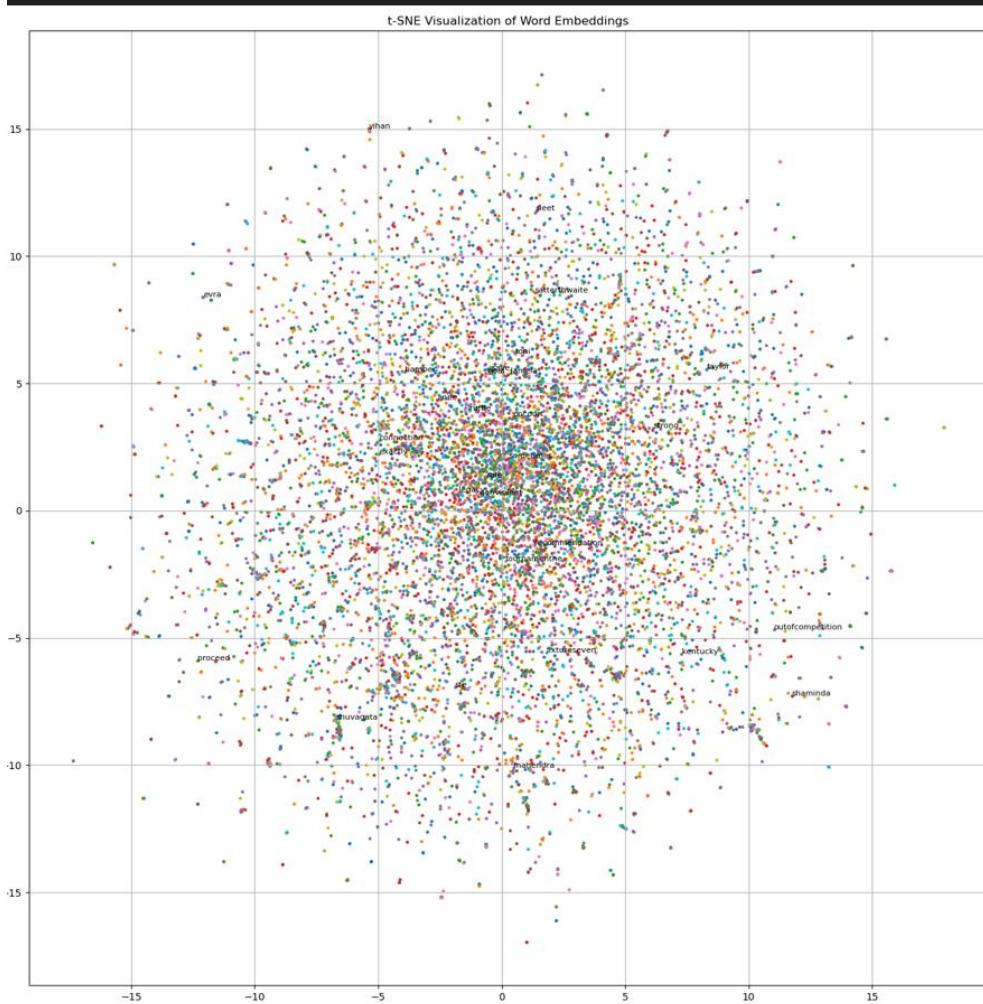
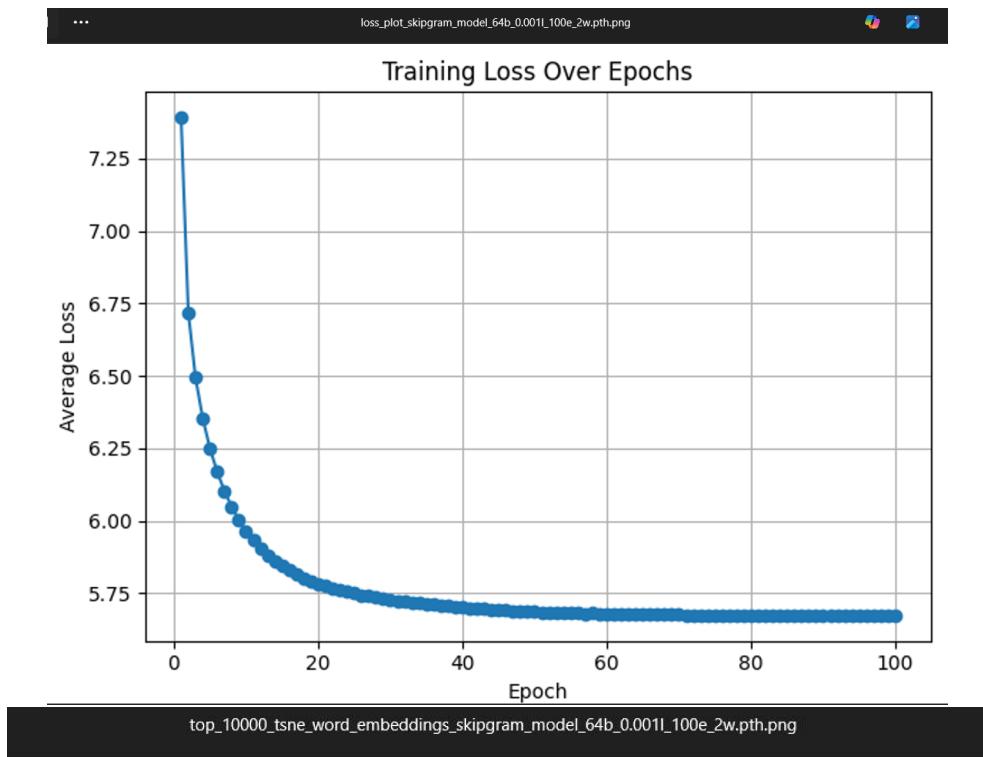


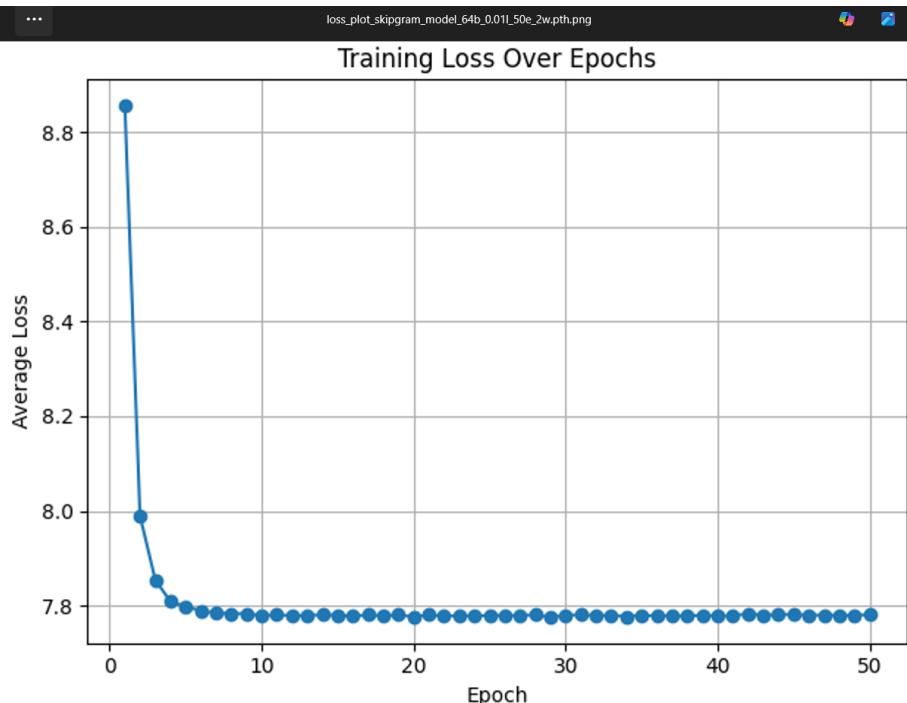




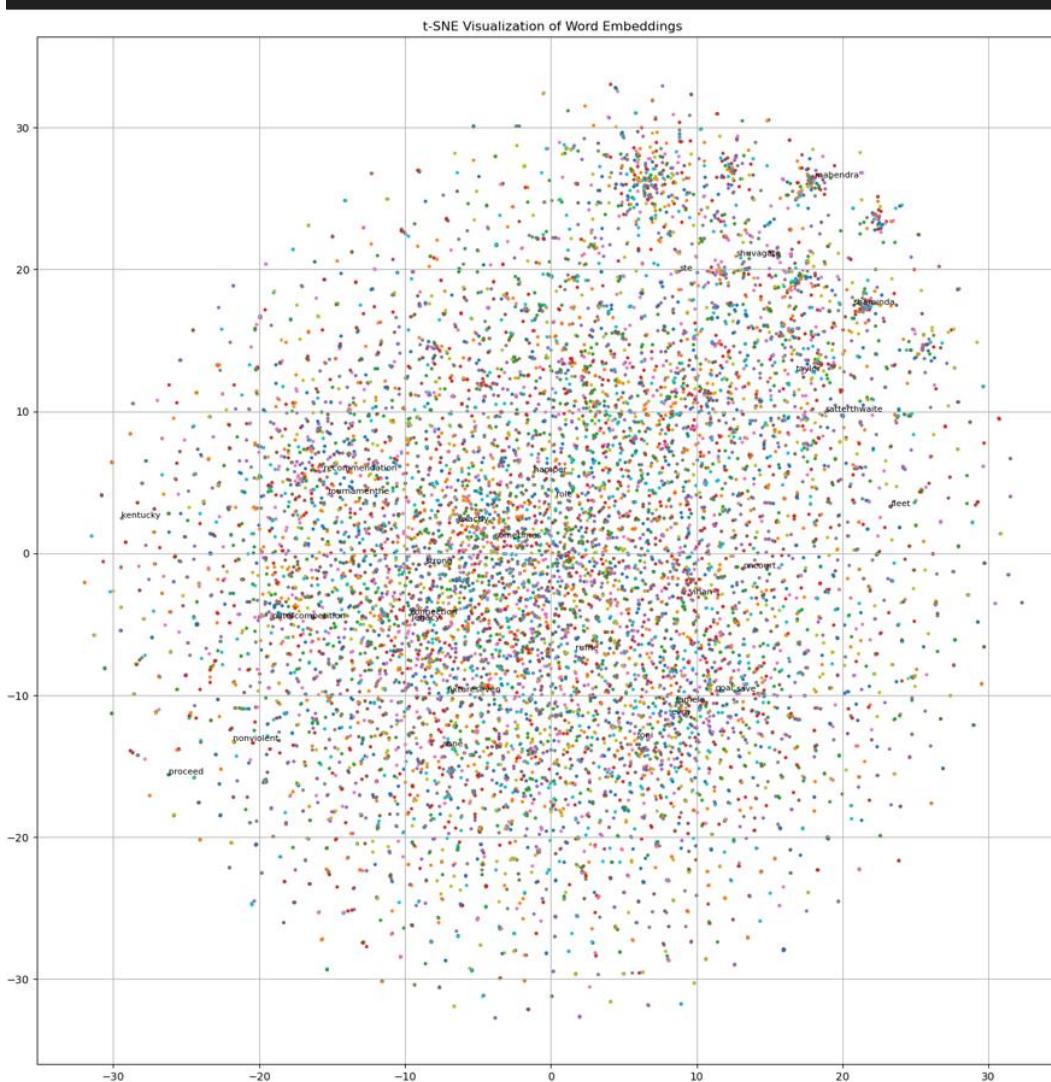


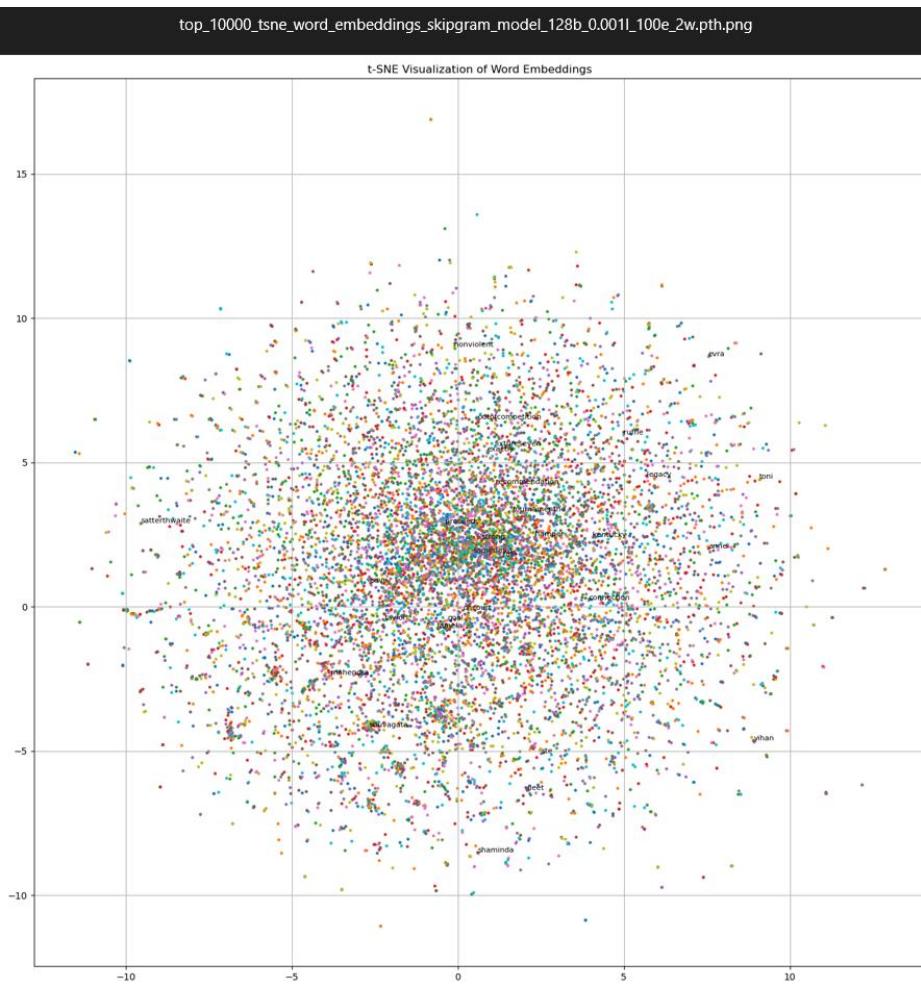
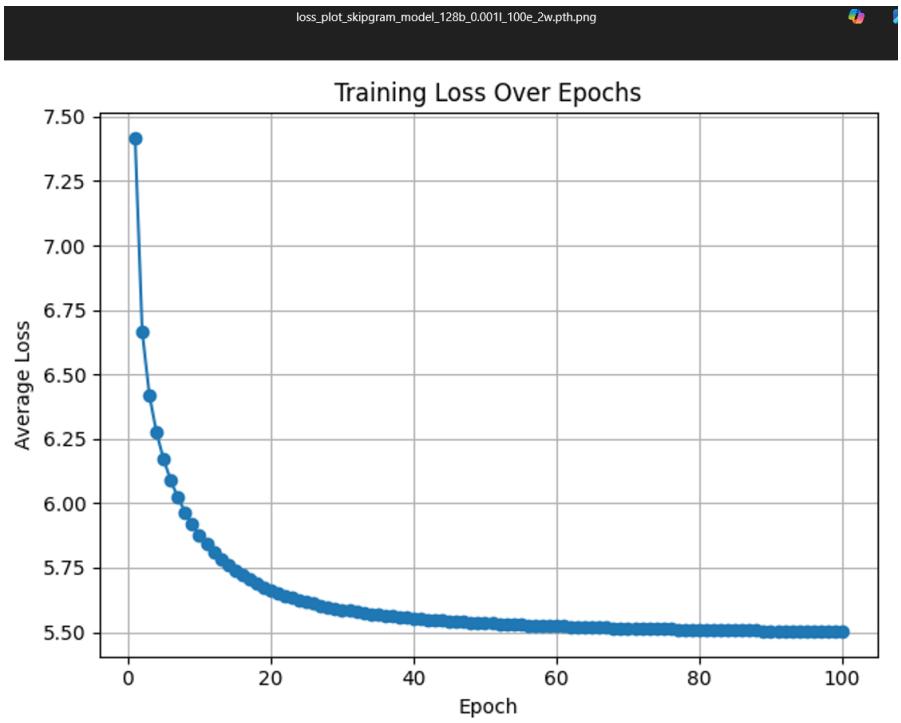


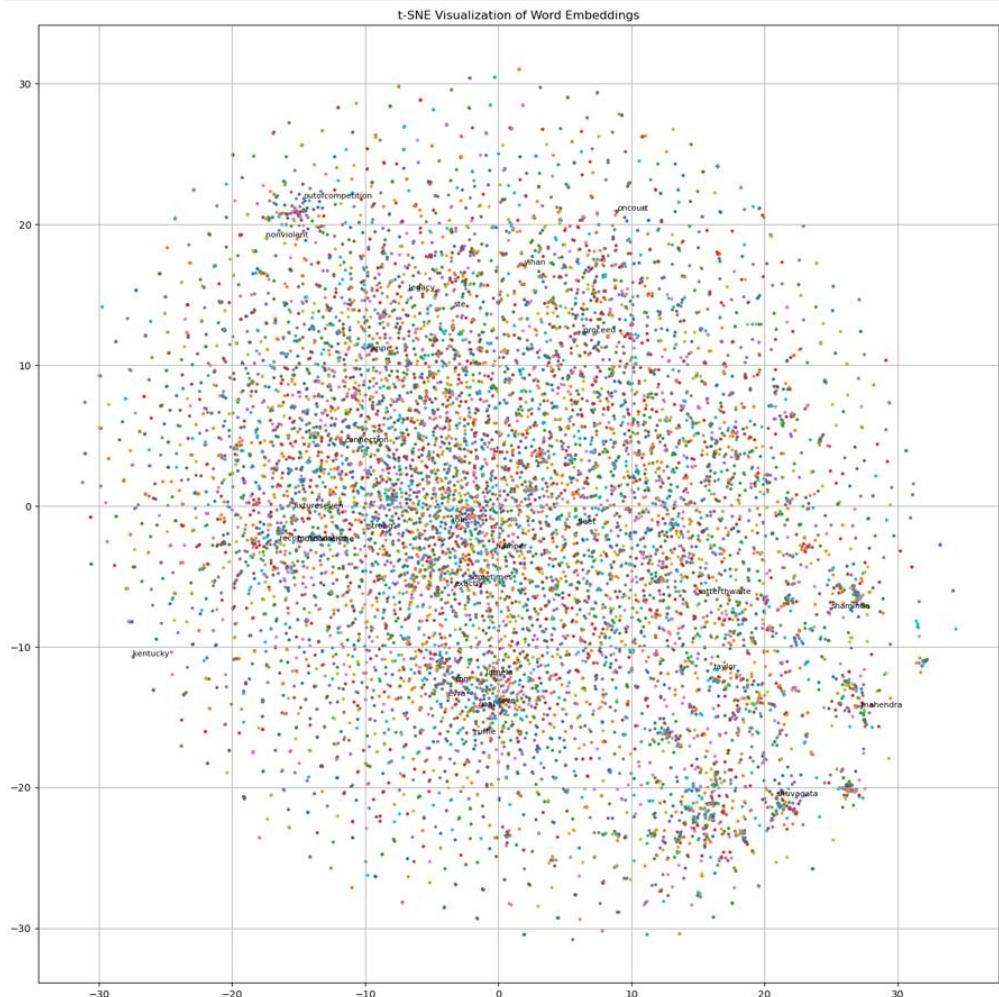
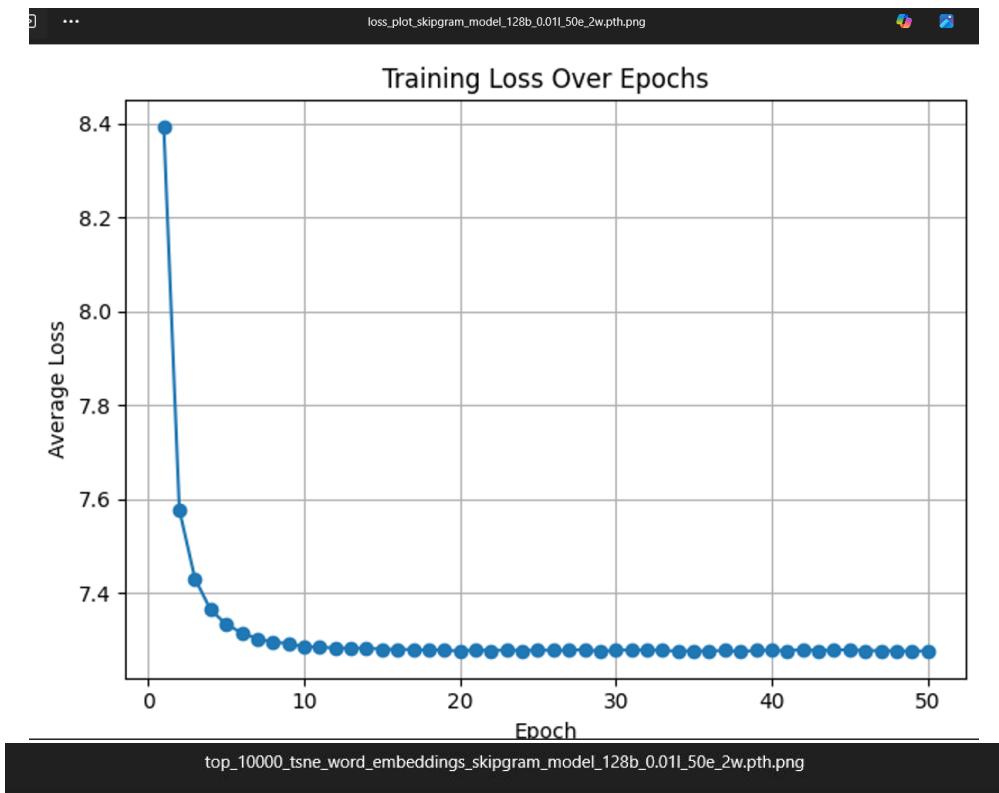


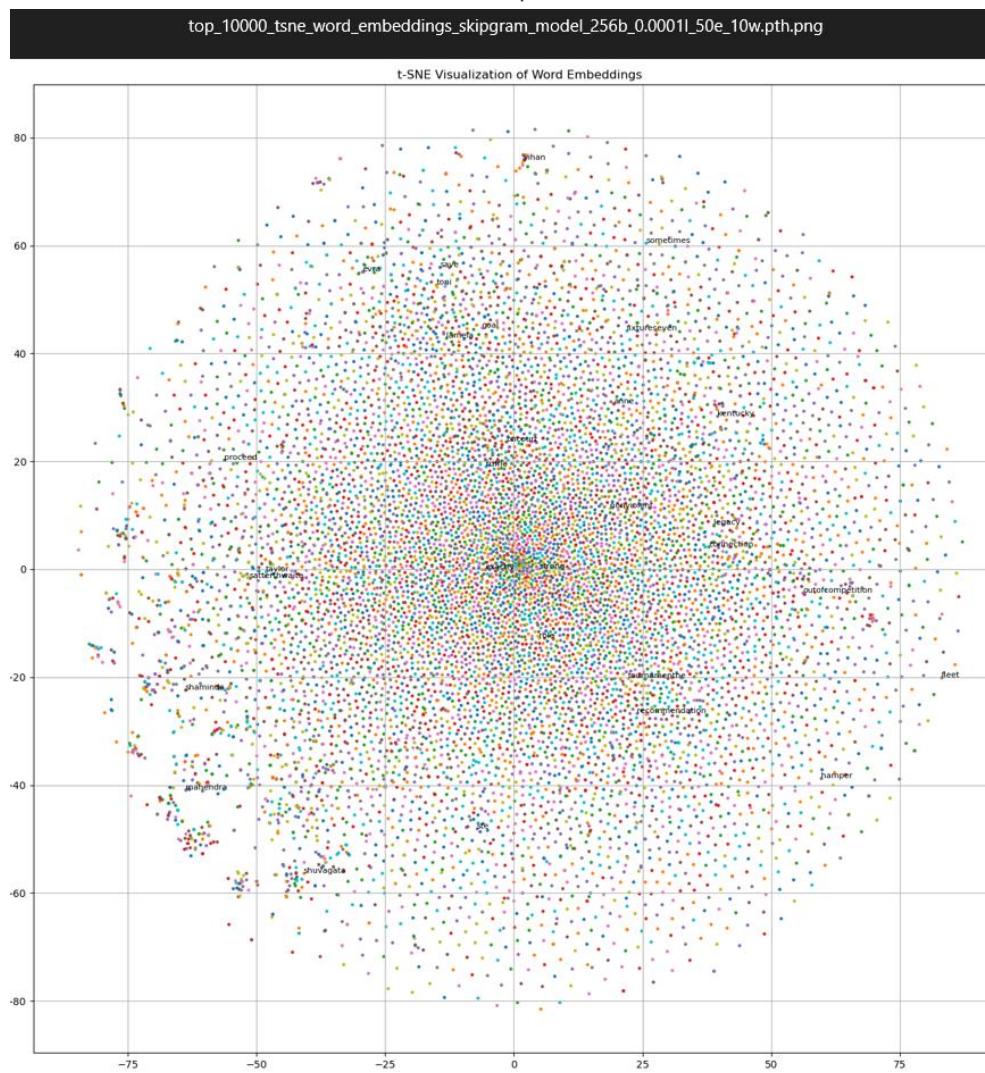
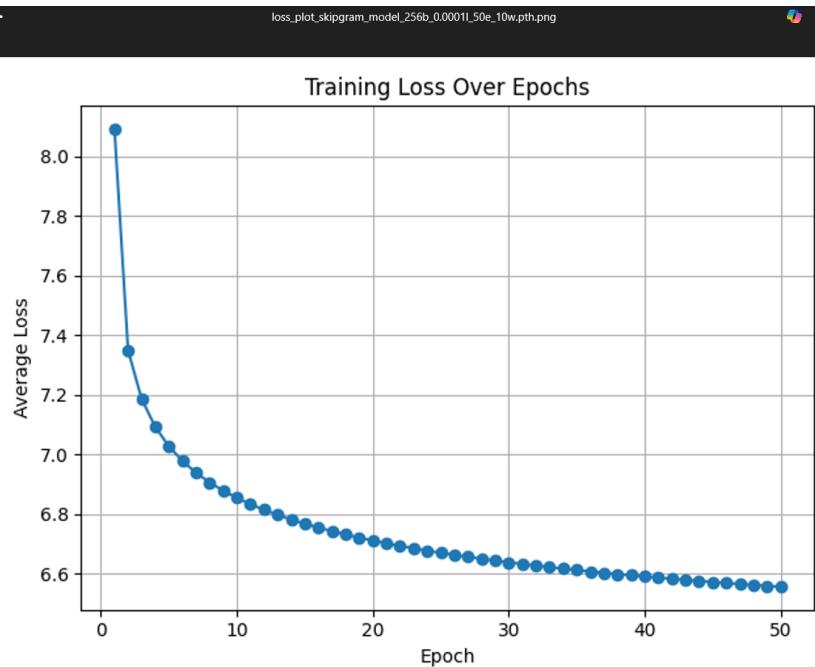


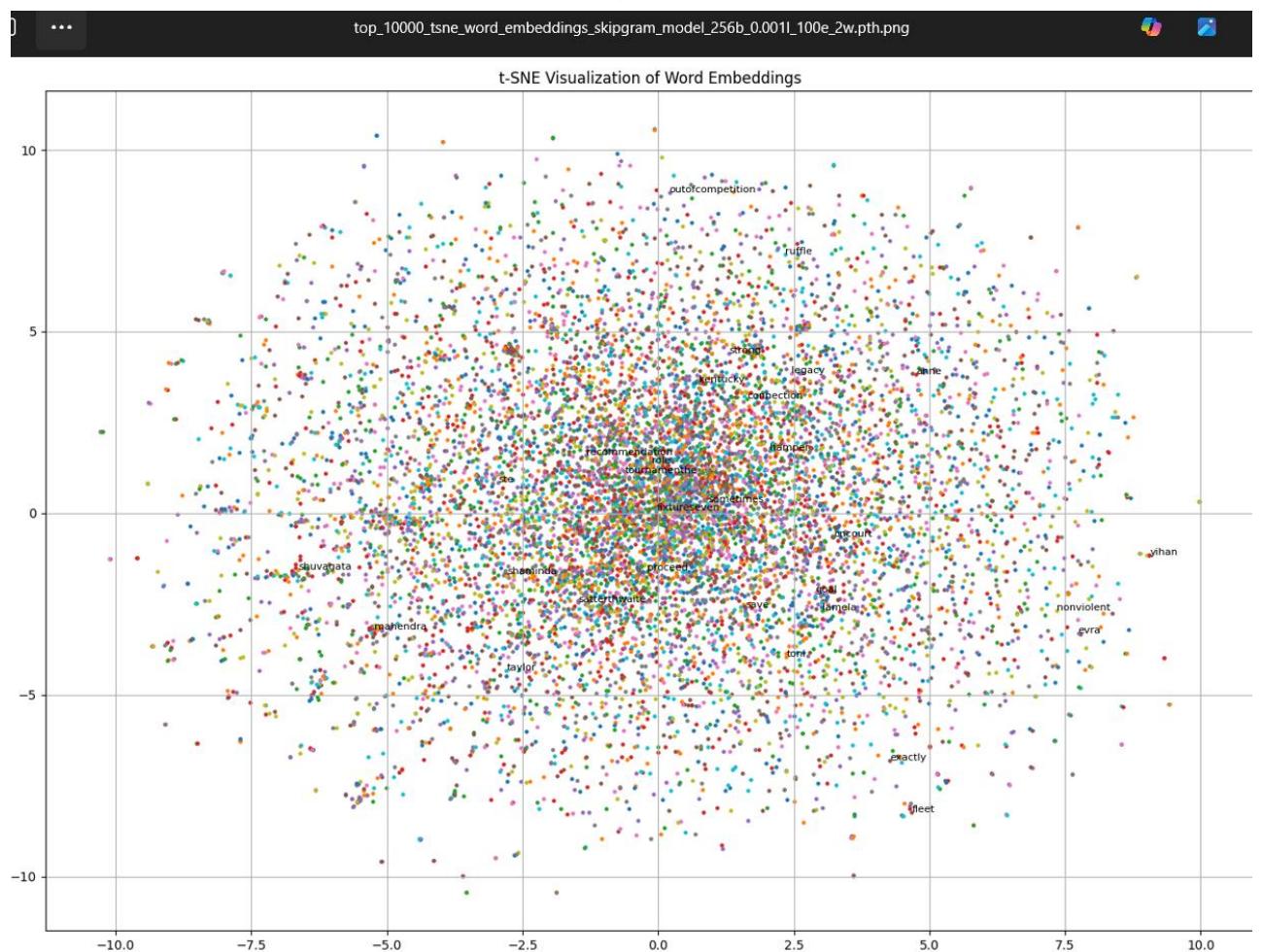
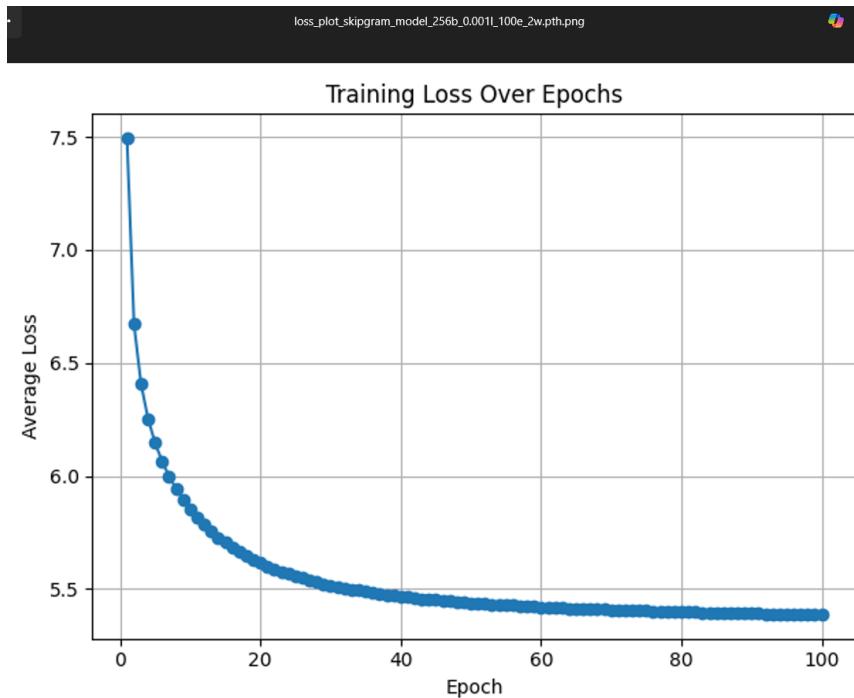
top_10000_tsne_word_embeddings_skipgram_model_64b_0.01l_50e_2w.pth.png

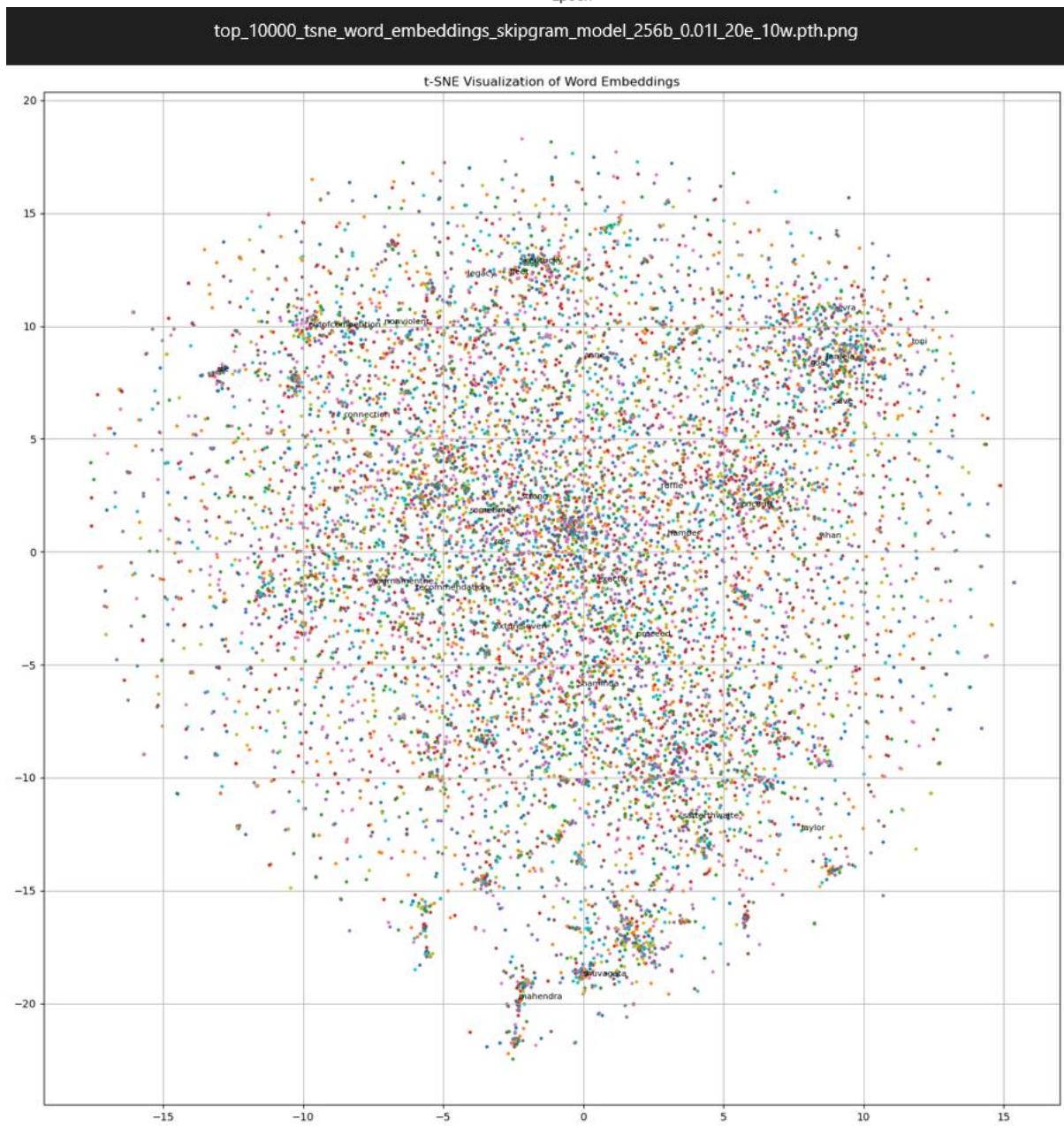
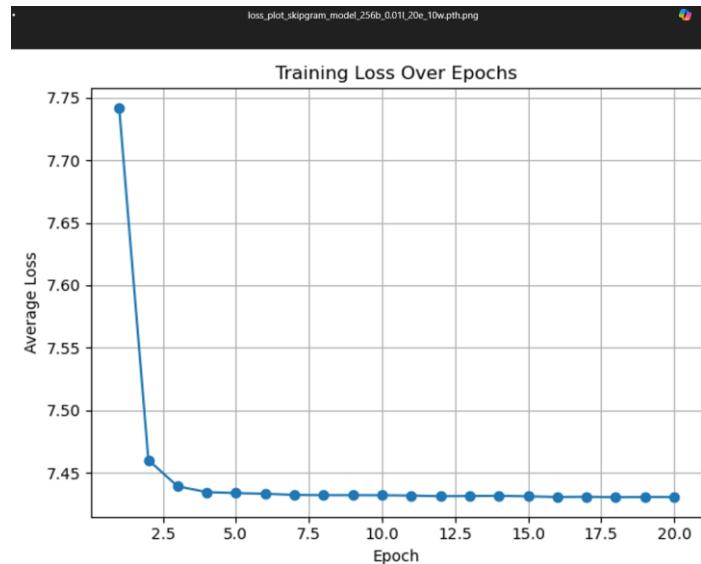


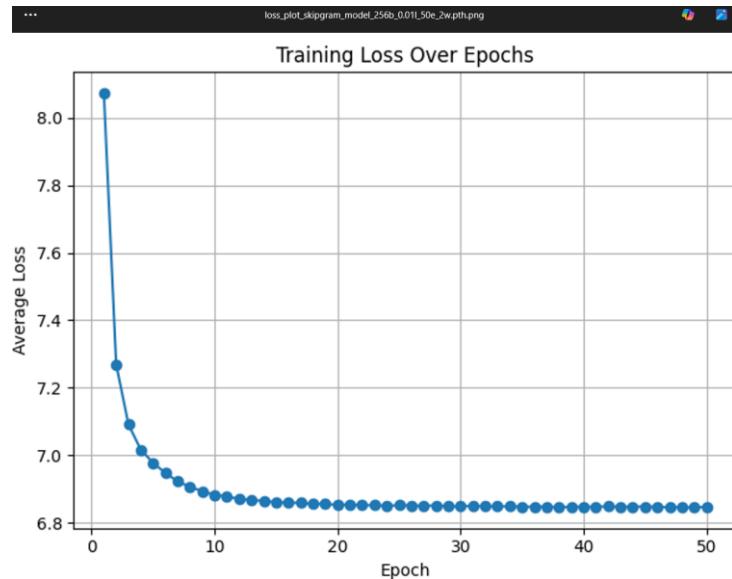




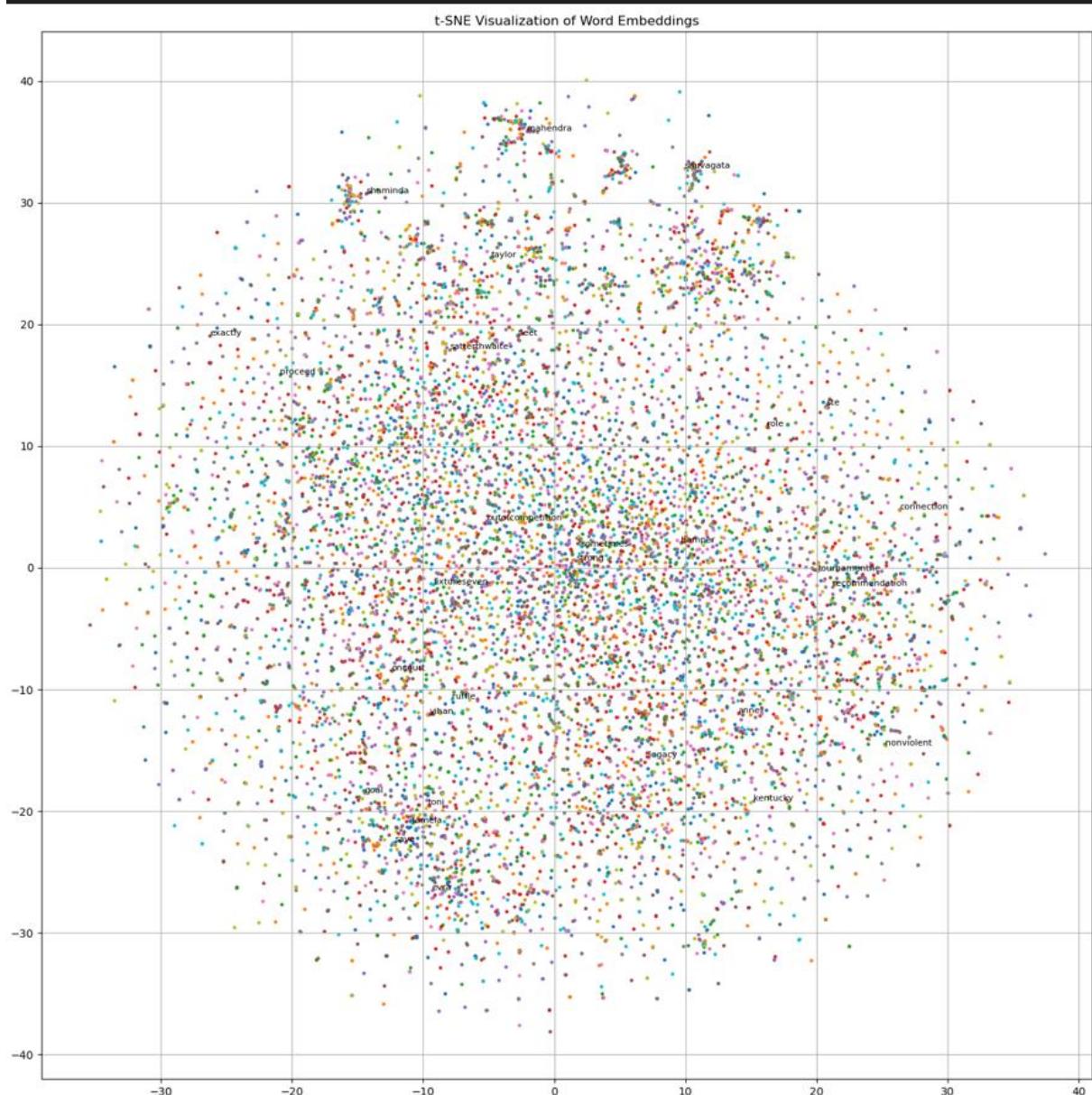


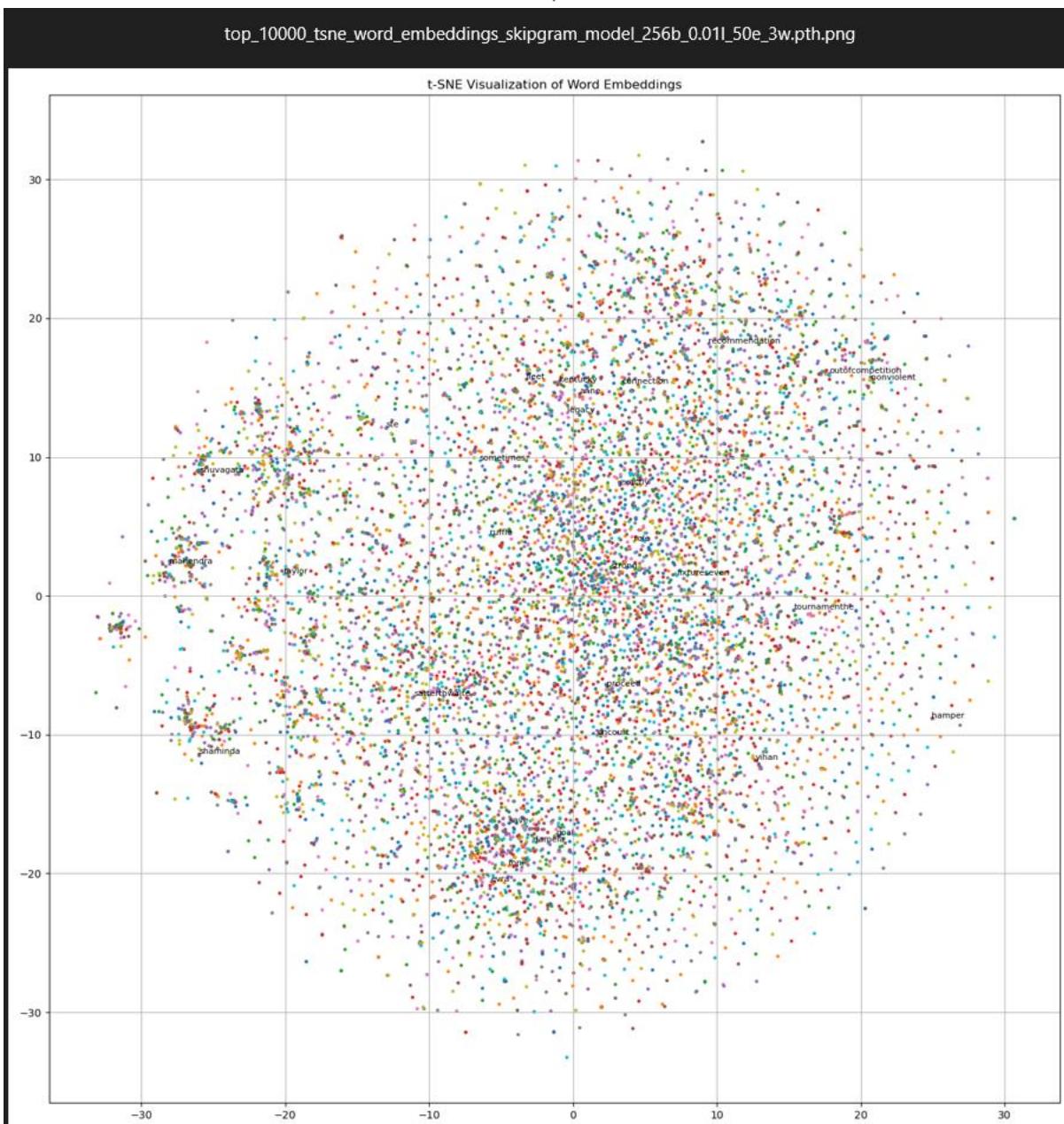
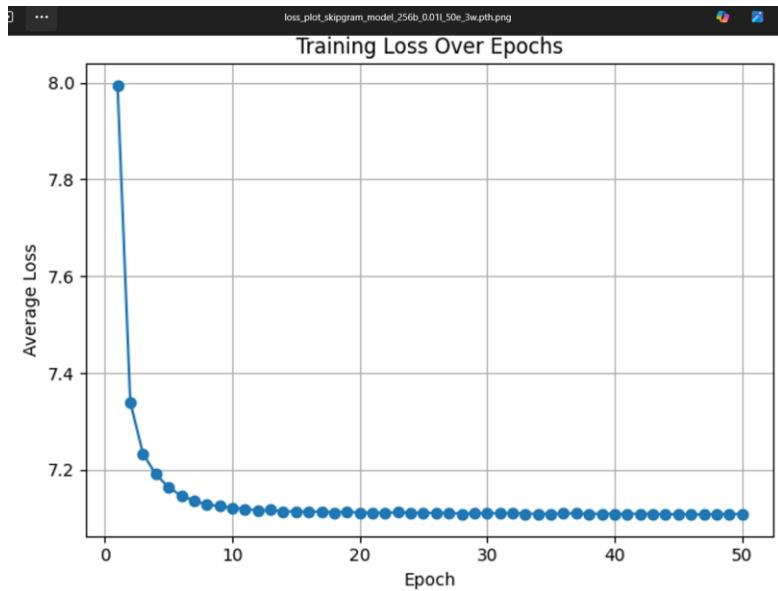


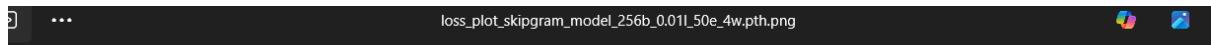


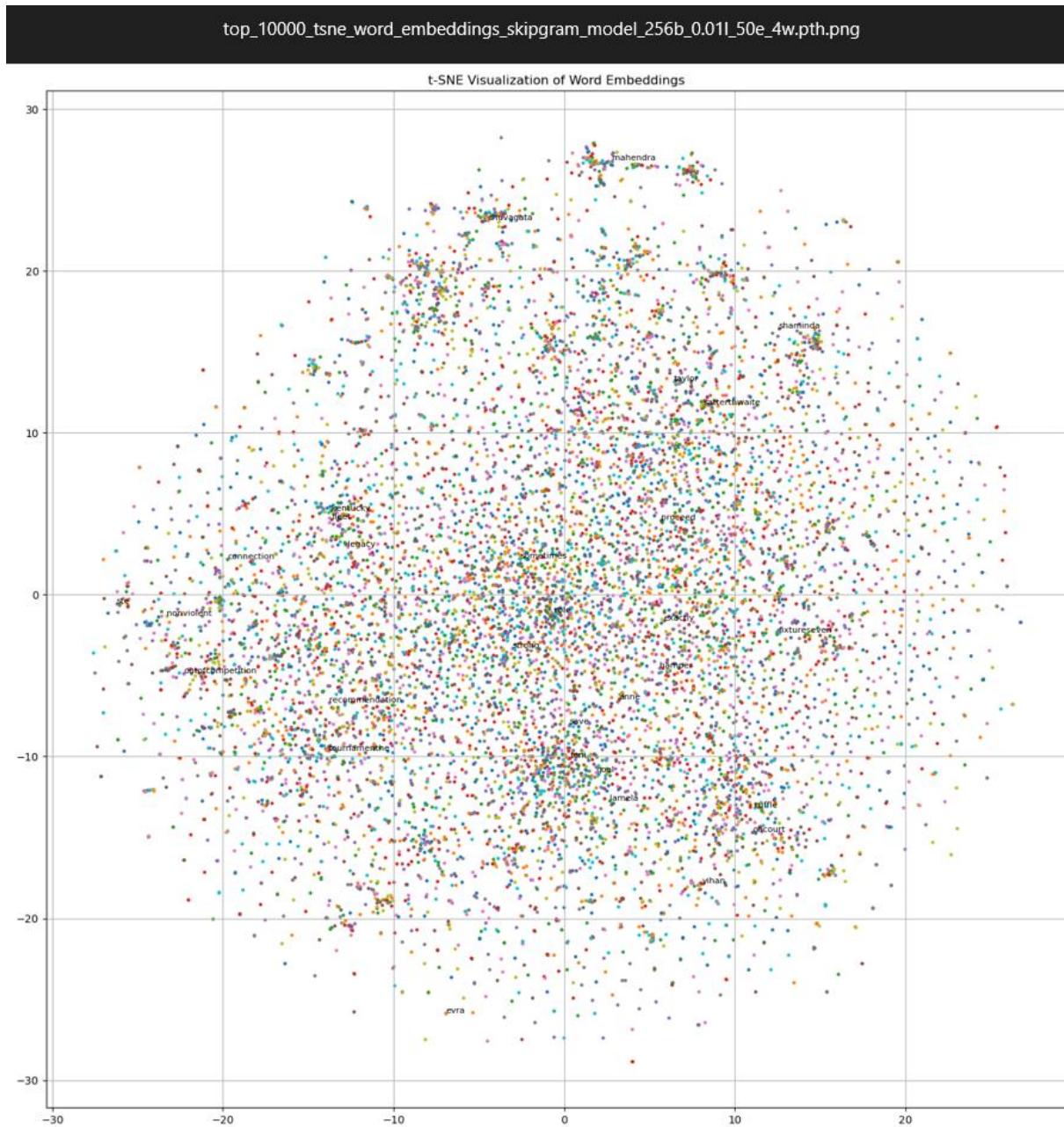


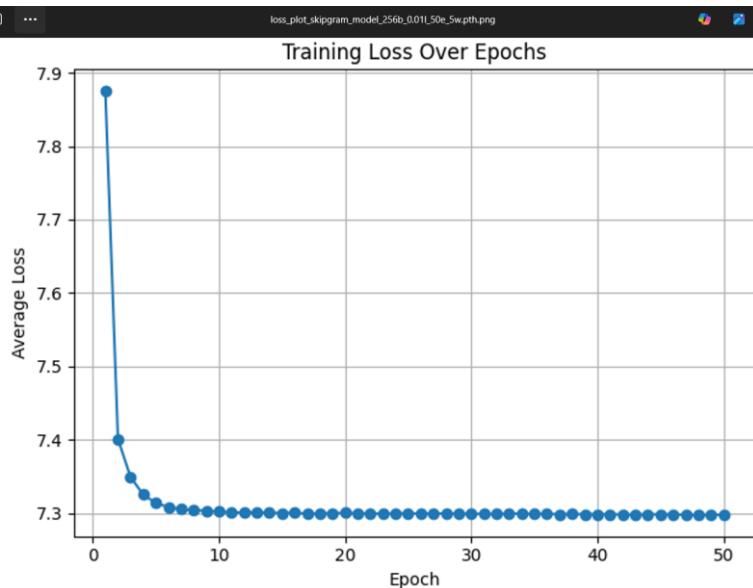
top_10000_tsne_word_embeddings_skipgram_model_256b_0.01l_50e_2w.pth.png



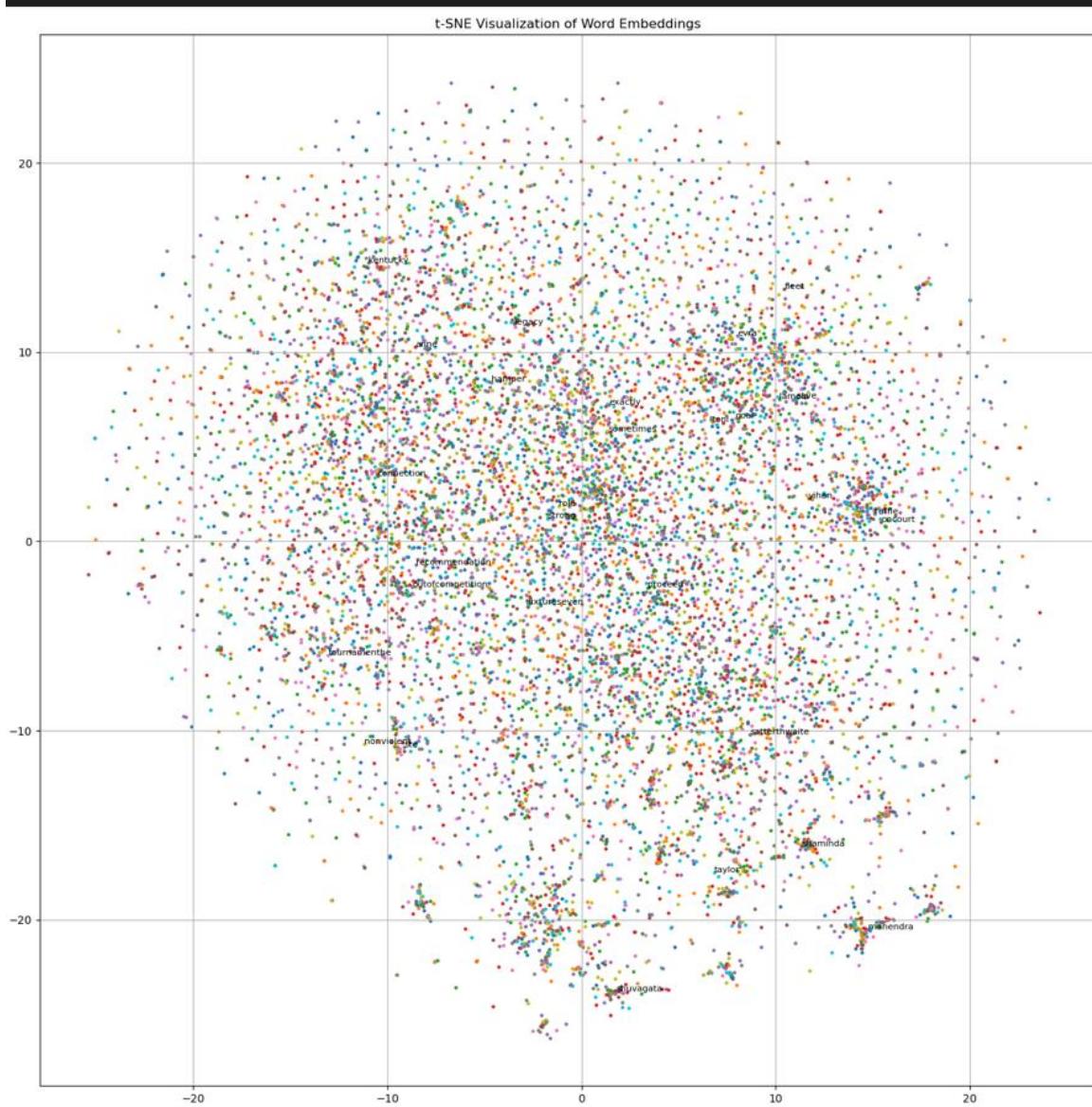


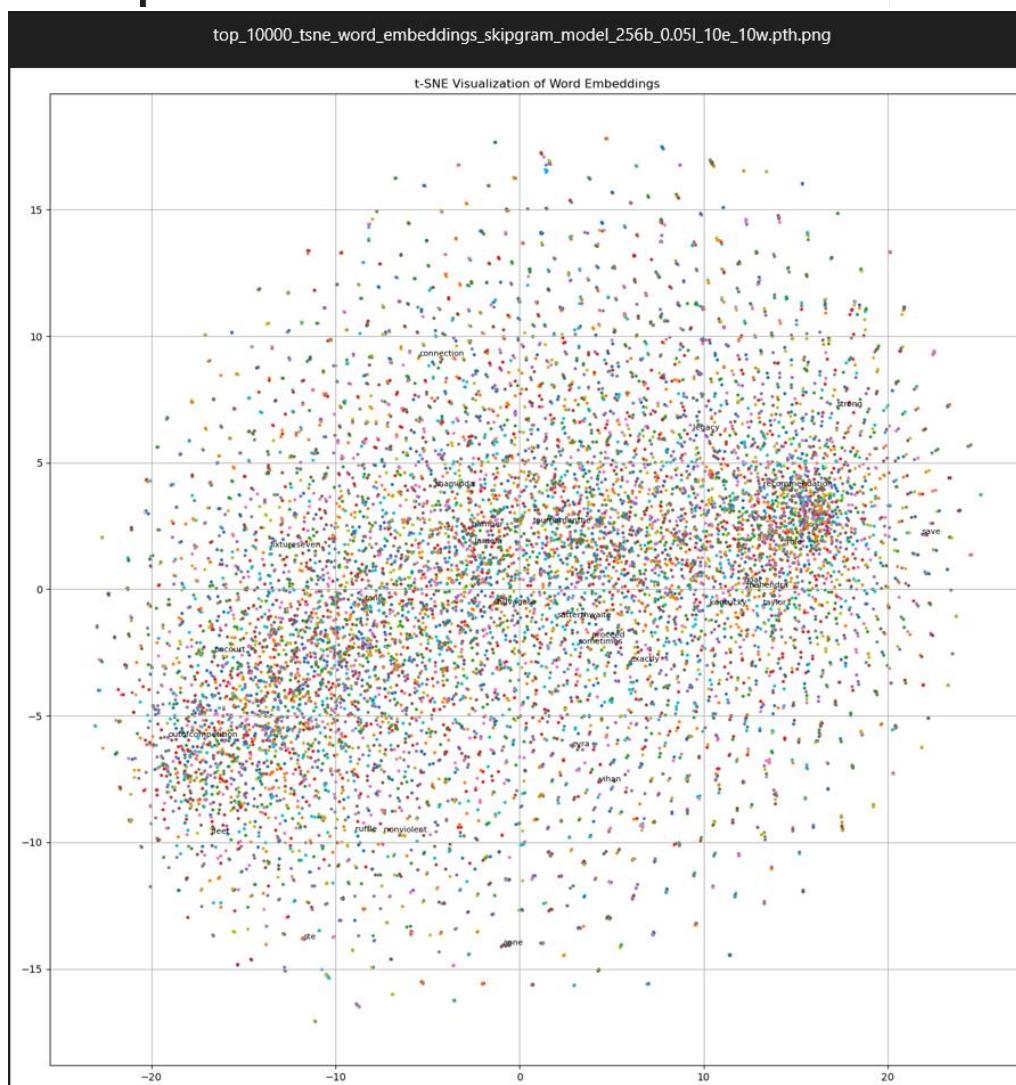
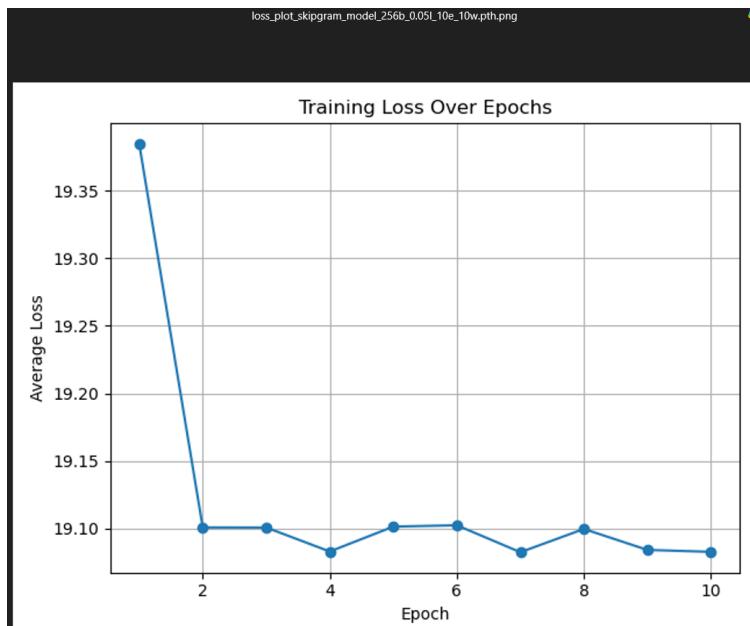






top_10000_tsne_word_embeddings_skipgram_model_256b_0.01l_50e_5w.pth.png





Source Code Appendix

1_data_pre_processing.py

```
import nltk
import spacy
import pandas as pd
import re
from collections import Counter
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer, SnowballStemmer
from tqdm import tqdm
from models.consts import MAX_VOCAB_SIZE, UNK_TOKEN
import pickle
tqdm.pandas()

# download packages for pre-processing
nltk.download('punkt')
nltk.download('stopwords')
spacy.cli.download("en_core_web_sm")
nlp = spacy.load("en_core_web_sm")
nltk.download('punkt_tab')
STOPWORDS = set(stopwords.words("english"))

lemmatizer = WordNetLemmatizer()
stemmer = SnowballStemmer('english')
nlp = spacy.load("en_core_web_sm", disable=["parser", "ner"])

# Task 1: Dataset Acquisition and Pre-Processing
# data importing & initial filtering by "business"/"sports" NewsType
news_type = "sports"
file_path = "Data/Articles.csv"
df = pd.read_csv(file_path, encoding='ISO-8859-1')
df = df[df['NewsType'] == news_type]
df = df.dropna(subset=['Article'])

def clean_text(text):
    # https://www.geeksforgeeks.org/machine-learning/python-efficient-text-data-cleaning/
    # text normalisation (lower-case, etc)
    text = text.lower()
    # ignore non-ascii characters
    text = text.encode('ascii', 'ignore').decode()
    # keep only words
    text = re.sub(r'[^a-z\s]', '', text)
    # trim spaces
    text = re.sub(r'\s+', ' ', text).strip()
    return text

df['clean_text'] = df['Article'].dropna().apply(clean_text)

# tokenisation & stop-word removal
df['tokens'] = df['clean_text'].apply(lambda x: [t for t in
word_tokenize(x) if t not in STOPWORDS])

# vocabulary limiting/filtering: for a more concise dictionary apply
```

```

lemmatisation
# https://www.kaggle.com/code/kishalmandal/all-about-stemming-and-
lemmatization-cleaning lemmatisation favoured over
# stemming because we want to learn word relationships. stemming removes
such relationships and removes meaning (e.g.
# program vs programmer -> both same stem(program), but have different
meanings).
# https://www.geeksforgeeks.org/python/python-lemmatization-with-nltk/

# lemmatisation using spaCy
def lemmatise_tokens(tokens):
    doc = nlp(" ".join(tokens))
    return [token.lemma_ for token in doc]

df['lemmas'] = df['tokens'].progress_apply(lemmatise_tokens)

# vocabulary creation - most common 10k words and mapped to indices
all_lemmas = [lemma for doc in df['lemmas'] for lemma in doc]
lemma_freq = Counter(all_lemmas)

most_common = lemma_freq.most_common(MAX_VOCAB_SIZE - 1) # -1 for UNK
vocab = {word: idx + 1 for idx, (word, _) in enumerate(most_common)} # start from 1
vocab[UNK_TOKEN] = 0 # UNK mapped to 0

# indexing tokens into numerical indices based on the constructed vocabulary
def convert_to_indices(lemmas):
    return [vocab.get(token, vocab[UNK_TOKEN]) for token in lemmas]

df['lemmas_indices'] = df['lemmas'].apply(convert_to_indices)

# print some info
print(f"Total business articles processed: {len(df)}")

# total number of tokens
total_tokens = sum(len(lemmas) for lemmas in df['lemmas'])
print(f"Total tokens: {total_tokens}")

# vocabulary size
print(f"Vocabulary size: {len(vocab)}")

# top 20 most common words
print("\nTop 20 most frequent lemmas:")
for word, freq in Counter([lemma for doc in df['lemmas'] for lemma in doc]).most_common(20):
    print(f"{word}<15} {freq}")

# keep only df["lemmas"] and df["lemmas_indices"]
df = df[['lemmas', 'lemmas_indices']]

# save
df.to_csv(f"Data/preprocessed_{news_type}_articles.csv", index=False)
df.to_pickle(f"Data/preprocessed_{news_type}_articles.pkl")

with open(f"Data/vocab_{news_type}_dict.pkl", "wb") as f:
    pickle.dump(vocab, f)

```

2_and_3_pytorch_model_training_and_optimisations.py

```
import torch.nn as nn
import random
import pandas as pd
import pickle
import numpy as np
import matplotlib.pyplot as plt
import os
import torch
import torch.optim as optim
from torch.utils.data import DataLoader
from models.cbow import CBOWModel, CBOWDataset, generate_cbow_data
from models.skipgram import SkipGramModel, SkipGramDataset,
generate_skipgram_data

# Chat GPT suggested:
torch.set_num_threads(torch.get_num_threads()) # Use all available CPU
threads
torch.backends.mkldnn.enabled = True # Enable MKLDNN (optimized CPU
kernels)

if __name__ == "__main__":
    def set_seed(seed=42):
        random.seed(seed)
        np.random.seed(seed)
        torch.manual_seed(seed)
        torch.backends.cudnn.deterministic = True
        torch.backends.cudnn.benchmark = False

    set_seed(42)
    #https://medium.com/@RobuRishabh/learning-word-embeddings-with-cbow-
    and-skip-gram-b834bde18de4
    #https://www.geeksforgeeks.org/nlp/word-embeddings-in-nlp/
    embedding_dim = 256
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    # load dataframe
    model_name = "cbow" #"skipgram"
    # hyperparameters
    batch_size = 128 #256
    learning_rate = 0.01
    epochs = 100 #12
    window_size = 2 #5

    category = "sports" # business
    suffix_category = f"{category}_"
    data_file = f"Data/preprocessed_{category}_articles.pkl"
    vocab_file = f"Data/vocab_{category}_dict.pkl"

    df = pd.read_pickle(data_file)

    # load & setup
    export_file_name =
    f"{model_name}_{model}_{batch_size}b_{learning_rate}l_{epochs}e_{window_size}_
    w.pth"
    export_folder_name =
    f"{model_name}_{model}_{batch_size}b_{learning_rate}l_{epochs}e_{window_size}_
    w_files"
    save_dir = f"models/{category}/{model_name}/{export_folder_name}"
    os.makedirs(save_dir, exist_ok=True)
    os.makedirs(f"{save_dir}/checkpoints", exist_ok=True)
```

```

with open(vocab_file, "rb") as f:
    vocab = pickle.load(f)

# prepare data and Model
if model_name == "cbow":
    model_data = generate_cbow_data(df['lemmas'].tolist(),
window_size=window_size)
    dataset = CBOWDataset(model_data, vocab)
    model = CBOWModel(len(vocab), embedding_dim).to(device)
else:
    model_data = generate_skipgram_data(df['lemmas'].tolist(),
window_size=window_size)
    dataset = SkipGramDataset(model_data, vocab)
    model = SkipGramModel(len(vocab), embedding_dim).to(device)

# mini batches
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True,
num_workers=4)

# loss, and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
loss_history = []

# training loop
for epoch in range(epochs):
    total_loss = 0
    model.train()

        # adapted from https://github.com/FraLotito/pytorch-continuous-bag-
        # of-words/blob/master/cbow.py
        # adapted training loop from both https://leshem-
        ido.medium.com/skip-gram-word2vec-algorithm-explained-85cd67a45ffa and
        https://github.com/FraLotito/pytorch-continuous-bag-of-
        words/blob/master/Cbow.py
        # also optimised by ChatGPT
        for context_idxs, target_idx in dataloader:
            context_idxs = context_idxs.to(device)
            target_idx = target_idx.to(device)
            optimizer.zero_grad()
            # forward
            output = model(context_idxs)  # output shape: (batch_size,
vocab_size)
            loss = criterion(output, target_idx)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        avg_loss = total_loss / len(dataloader)
        loss_history.append(avg_loss) # log the average loss
        print(f"Epoch {epoch + 1}/{epochs}, Loss: {avg_loss:.4f}")
        # save the trained model
        if (epoch + 1) % 10 == 0:
            os.makedirs(f"{save_dir}/checkpoints/checkpoint_{epoch}e",
exist_ok=True),
            torch.save(model.state_dict(),
f"{save_dir}/checkpoints/checkpoint_{epoch}e/checkpoint_{epoch}e_{export_fi
le_name}")
            loss_df = pd.DataFrame({'epoch': range(1, epoch + 2), 'loss':
loss_history})

```

```
loss_df.to_csv(f"{save_dir}/checkpoints/checkpoint_{epoch}e/training_loss_log_checkpoint_{epoch}e_{export_file_name}.csv", index=False)

torch.save(model.state_dict(), f"{save_dir}/{export_file_name}")
loss_df = pd.DataFrame({'epoch': range(1, epochs + 1), 'loss': loss_history})
loss_df.to_csv(f"{save_dir}/training_loss_log_{export_file_name}.csv",
index=False)

plt.plot(range(1, epochs + 1), loss_history, marker='o')
plt.title("Training Loss Over Epochs")
plt.xlabel("Epoch")
plt.ylabel("Average Loss")
plt.grid(True)
plt.tight_layout()
plt.savefig(f"{save_dir}/loss_plot_{export_file_name}.png")
plt.show()

print(f"Model saved to {save_dir}")
```

4_tSNE_visualisation_and_analysis.py

```
import torch
import random
import pickle
from models.consts import UNK_TOKEN
from models.cbow import CBOWModel
from models.skipgram import SkipGramModel
from sklearn.manifold import TSNE
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os

# Chat GPT suggested:
torch.set_num_threads(torch.get_num_threads())
torch.backends.mkldnn.enabled = True

def set_seed(seed=42):
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

set_seed(42)
device = torch.device("cpu")

# set params
model_name = "cbow" # "skipgram"
batch_size = "128" #256
lr = "0.01"
epochs = "100" #"12"
window = "2" # "5"
filepath =
f"models/sports/{model_name}/{model_name}_model_{batch_size}b_{lr}l_{epochs}e_{window}w_files/"
filename =
f"{model_name}_model_{batch_size}b_{lr}l_{epochs}e_{window}w.pth"

embedding_dim = 256

convergence_visual_path = f"{filepath}loss_plot_{filename}.png"
if os.path.exists(convergence_visual_path) == False:
    df = pd.read_csv(f"{filepath}training_loss_log_{filename}.csv")
    loss = df["loss"].astype(float).tolist()
    plt.plot(range(1, int(epochs) + 1), loss, marker='o')
    plt.title("Training Loss Over Epochs")
    plt.xlabel("Epoch")
    plt.ylabel("Average Loss")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(convergence_visual_path)
    plt.show()

    print(f"Model saved to {convergence_visual_path}")

# load data
with open("Data/vocab_sports_dict.pkl", "rb") as f:
```

```

vocab = pickle.load(f)

if model_name == "cbow":
    model = CBOWModel(len(vocab), embedding_dim).to(device)
else:
    model = SkipGramModel(len(vocab), embedding_dim).to(device)

# load saved state dict
model.load_state_dict(torch.load(filepath + filename, map_location=device))
print(model.eval())

# get embedding weights: shape (vocab_size, embedding_dim)
embeddings = model.embeddings.weight.data.cpu().numpy()

# dimensionality reduction:
# use t-SNE to reduce the high-dimensional embeddings to 2D space.
# https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
tsne = TSNE(n_components=2, random_state=42, perplexity=30, max_iter=1000)
embeddings_2d = tsne.fit_transform(embeddings)

# Invert vocab dictionary to get word from index
idx_to_word = {idx: word for word, idx in vocab.items()}
plt.figure(figsize=(14, 14))

# Plot entire vocab, but plot only top 30 words
n = 30
N = 10000
for i in range(N):
    x, y = embeddings_2d[i, 0], embeddings_2d[i, 1]
    word = idx_to_word.get(i, UNK_TOKEN)
    plt.scatter(x, y, s=5)

# Randomly select n unique indices to label
label_indices = random.sample(range(N), n)

for i in label_indices:
    x, y = embeddings_2d[i, 0], embeddings_2d[i, 1]
    word = idx_to_word.get(i, UNK_TOKEN)
    plt.text(x + 0.002, y + 0.002, word, fontsize=8)

# plt.xlim(-8, 8)
# plt.ylim(-8, 8)

plt.title("t-SNE Visualization of Word Embeddings")
plt.grid(True)
plt.tight_layout()
plt.savefig(f"{filepath}top_{N}_tsne_word_embeddings_{filename}.png")

# define groups for better analysis
categories = {
    'Gender': ["man", "woman", "boy", "girl", "male", "female", "men",
               "women", "womens"],
    'Royalty': ["king", "queen", "man", "woman", "prince", "princess"],
    'Business': ["businessman", "man", "business", "businessmenin"],
    'Sports': ["foot", "ball", "football", "soccer", "american", "cricket",
               "india", "basketball", "america", "states",
               "olympics", "major", "tournament", "handball"],
    'Country': ["america", "american", "british", "english", "england",
                "united", "states", "ozil", "mueller",
                "germany", "germans", "turkey", "bonucci", "armonucci",
}

```

```

"italy"],
    'Colors': ["yellow", "gold", "red", "green", "blue", "angry"],
    'Position': ["silver", "second", "gold", "first", "third", "bronze",
"win", "lose"]
}

# plot of filtered word embeddings for analysis
for key, word_list in categories.items():
    words_subset = [word for word in word_list if word in vocab]
    indices = [vocab[word] for word in words_subset]
    coords = embeddings_2d[indices]
    plt.figure(figsize=(10, 8))
    for i, word in enumerate(words_subset):
        x, y = coords[i]
        plt.scatter(x, y, color='blue')
        plt.text(x + 0.02, y + 0.02, word, fontsize=10)

    plt.title("t-SNE Plot of Selected Word Embeddings")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(f"{filepath}{key}_subset_words_tsne_{filename}.png")

# save embeddings to csv
data = []
for i in range(len(embeddings_2d)):
    word = idx_to_word.get(i, UNK_TOKEN)
    x, y = embeddings_2d[i]
    data.append((word, x, y))

df = pd.DataFrame(data, columns=['word', 'x', 'y'])
csv_path = f"{filepath}tsne_embeddings_{filename}.csv"
df.to_csv(csv_path, index=False)

```

models/word_embedding.py

```

import torch.nn as nn

class WordEmbeddingModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim):
        super(WordEmbeddingModel, self).__init__()
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)

    def get_embedding(self, input_ids):
        return self.embeddings(input_ids)

```

skipgram.py

```

import torch
import torch.nn as nn
from torch.utils.data import Dataset

from models.consts import UNK_TOKEN
from .word_embedding import WordEmbeddingModel
import torch.nn.functional as F

# Chat GPT suggested:
torch.set_num_threads(torch.get_num_threads())
torch.backends.mkldnn.enabled = True

```

```

# Boilerplate code from ChatGPT, but later adapted to incorporate code
from:
# https://leshem-ido.medium.com/skip-gram-word2vec-algorithm-explained-
85cd67a45ffa
class SkipGramModel(WordEmbeddingModel):
    def __init__(self, vocab_size, embedding_dim):
        super(SkipGramModel, self).__init__(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, vocab_size)

    def forward(self, center_idxs):
        # center_idxs: (batch_size)
        # shape: (batch_size, emb_dim)
        embeds = self.embeddings(center_idxs)
        out = self.linear(embeds)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs

# Boilerplate code from ChatGPT, but later adapted to incorporate code from
# https://leshem-ido.medium.com/skip-gram-word2vec-algorithm-explained-
85cd67a45ffa
class SkipGramDataset(Dataset):
    def __init__(self, data, vocab):
        self.data = data
        self.vocab = vocab

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        center, context = self.data[idx]
        center_idx = torch.tensor(self.vocab.get(center,
self.vocab[UNK_TOKEN]), dtype=torch.long)
        context_idx = torch.tensor(self.vocab.get(context,
self.vocab[UNK_TOKEN]), dtype=torch.long)
        return center_idx, context_idx

#https://leshem-ido.medium.com/skip-gram-word2vec-algorithm-explained-
85cd67a45ffa
def generate_skipgram_data(tokens_list, window_size=2):
    data = []
    for tokens in tokens_list:
        for i in range(len(tokens)):
            center = tokens[i]
            for j in range(max(0, i - window_size), min(len(tokens), i + window_size + 1)):
                if j != i:
                    data.append((center, tokens[j]))
    return data

```

consts.py

```

MAX_VOCAB_SIZE = 10000
UNK_TOKEN = "<UNK>"

```

cbow.py

```
import torch
import torch.nn as nn
from .word_embedding import WordEmbeddingModel
from torch.utils.data import Dataset
from .consts import UNK_TOKEN
import torch.nn.functional as F

# Chat GPT suggested:
torch.set_num_threads(torch.get_num_threads())
torch.backends.mkldnn.enabled = True # For optimized CPU kernels

# Boilerplate code from ChatGPT, but later adapted to incorporate code from
# https://github.com/FraLotito/pytorch-continuous-bag-of-
words/blob/master/cbow.py
class CBOWModel(WordEmbeddingModel):
    def __init__(self, vocab_size, embedding_dim):
        super(CBOWModel, self).__init__(vocab_size, embedding_dim)
        self.linear = nn.Linear(embedding_dim, vocab_size)

    def forward(self, context_idxs):
        # context_idxs: (batch_size, context_window*2)
        # shape: (batch_size, context_size, emb_dim)
        embeds = self.embeddings(context_idxs)
        # average over context
        context_embed = embeds.mean(dim=1)
        out = self.linear(context_embed)
        log_probs = F.log_softmax(out, dim=1)
        return log_probs

# Dataset class for CBOW
# Boilerplate code from ChatGPT, but later adapted to incorporate code
# from:
# https://github.com/FraLotito/pytorch-continuous-bag-of-
words/blob/master/cbow.py
class CBOWDataset(Dataset):
    def __init__(self, data, vocab):
        self.data = data
        self.vocab = vocab

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        context, target = self.data[idx]
        context_idxs = torch.tensor([self.vocab.get(w,
self.vocab[UNK_TOKEN]) for w in context], dtype=torch.long)
        target_idx = torch.tensor(self.vocab.get(target,
self.vocab[UNK_TOKEN]), dtype=torch.long)
        return context_idxs, target_idx

# altered from https://github.com/FraLotito/pytorch-continuous-bag-of-
words/blob/master/cbow.py
def generate_cbow_data(tokens_list, window_size=2):
    data = []
    for tokens in tokens_list:
        for i in range(window_size, len(tokens) - window_size):
            context = tokens[i - window_size:i] + tokens[i + 1:i +
```

```
window_size + 1]
        target = tokens[i]
        data.append((context, target))
    return data
```

References

- [1] J. Abela, ‘JA117 NLP Transformers & Large Language Models (LLMs) Part 1 – Introduction and Concepts’, University Of Malta, 2025.
- [2] J. Abela, ‘JA117 NLP Transformers & Large Language Models (LLMs) Part 2 – Preliminaries’, University Of Malta, 2025.
- [3] J. Abela, ‘JA117 NLP Transformers & Large Language Models (LLMs) Part 3 – Word Embeddings’, University Of Malta, 2025.
- [4] G. C. J. D. Tomas Mikolov Kai Chen, “Efficient Estimation of Word Representations in Vector Space.” 2013. [Online]. Available: <https://arxiv.org/abs/1301.3781>
- [5] J. Pennington, R. Socher, and C. D. Manning, “Glove: Global vectors for word representation,” Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), pp. 1532–1543, 2014
- [6] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, “FastText.zip: Compressing text classification models.” 2016. [Online]. Available: <https://arxiv.org/abs/1612.03651>
- [7] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, “Enriching Word Vectors with Subword Information.” 2017. [Online]. Available: <https://arxiv.org/abs/1607.04606>
- [8] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding.” 2019. [Online]. Available: <https://arxiv.org/abs/1810.04805>
- [9] M. E. Peters et al., “Deep contextualized word representations.” 2018. [Online]. Available: <https://arxiv.org/abs/1802.05365>
- [10] A. Vaswani et al., “Attention is All you Need,” in Advances in Neural Information Processing Systems, I. Guyon, U. von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
- [11] F. Almeida and G. Xexéo, “Word Embeddings: A Survey.” 2023. [Online]. Available: <https://arxiv.org/abs/1901.09069>

ChatGPT usage

Search terms for the literature review were suggested

The screenshot shows a dark-themed ChatGPT interface. On the left, a user prompt asks for search terms for Google Scholar. Below it, three sections of suggested search terms are listed:

- 1. Understand Word Embeddings**: Includes a tip to add "survey" or "review".
- 2. Implement Classic Architectures (CBOW and Skip-Gram)**: Includes a note about including "gensim".
- 3. Evaluate Semantic Quality (t-SNE + visualization)**: Includes a note about trying "PCA vs t-SNE".

Each section provides a list of search terms enclosed in code-like boxes.

ChatGPT also aided in the highlighting of results from Pennington et al. [5], suggesting what to focus on in the literature review.

The screenshot shows a dark-themed ChatGPT interface where the user has asked ChatGPT to highlight key results from a paper. ChatGPT has provided a summary and highlighted specific sections of the paper's text.

highlight the key results for my literature review. I would like a summary of 2-4 sentences. Base everything off this text:

4.4 Model Analysis: Vector Length and Context Size
In Fig. 2, we show the results of experiments that vary vector length and context window. A context window that extends to the left and right of a target word will be called symmetric, and one which extends only to the left will be called asymmetric. In (a), we observe diminishing returns for vectors larger than about 200 dimensions. In (b) and (c), we examine the effect of varying the window size for symmetric and asymmetric context windows. Performance is better on the syntactic subtask for small and asymmetric context windows, which aligns with the intuition that syntactic information is mostly drawn from the immediate context and can depend strongly on word order. Semantic information, on the other hand, is more frequently non-local, and more of it is captured with larger window sizes.

4.5 Model Analysis: Corpus Size
In Fig. 3, we show performance on the word analogy task for 300-dimensional vectors trained on different corpora. In the syntactic subtask, there is a monotonic increase in performance as the corpus size increases. This is to be expected since

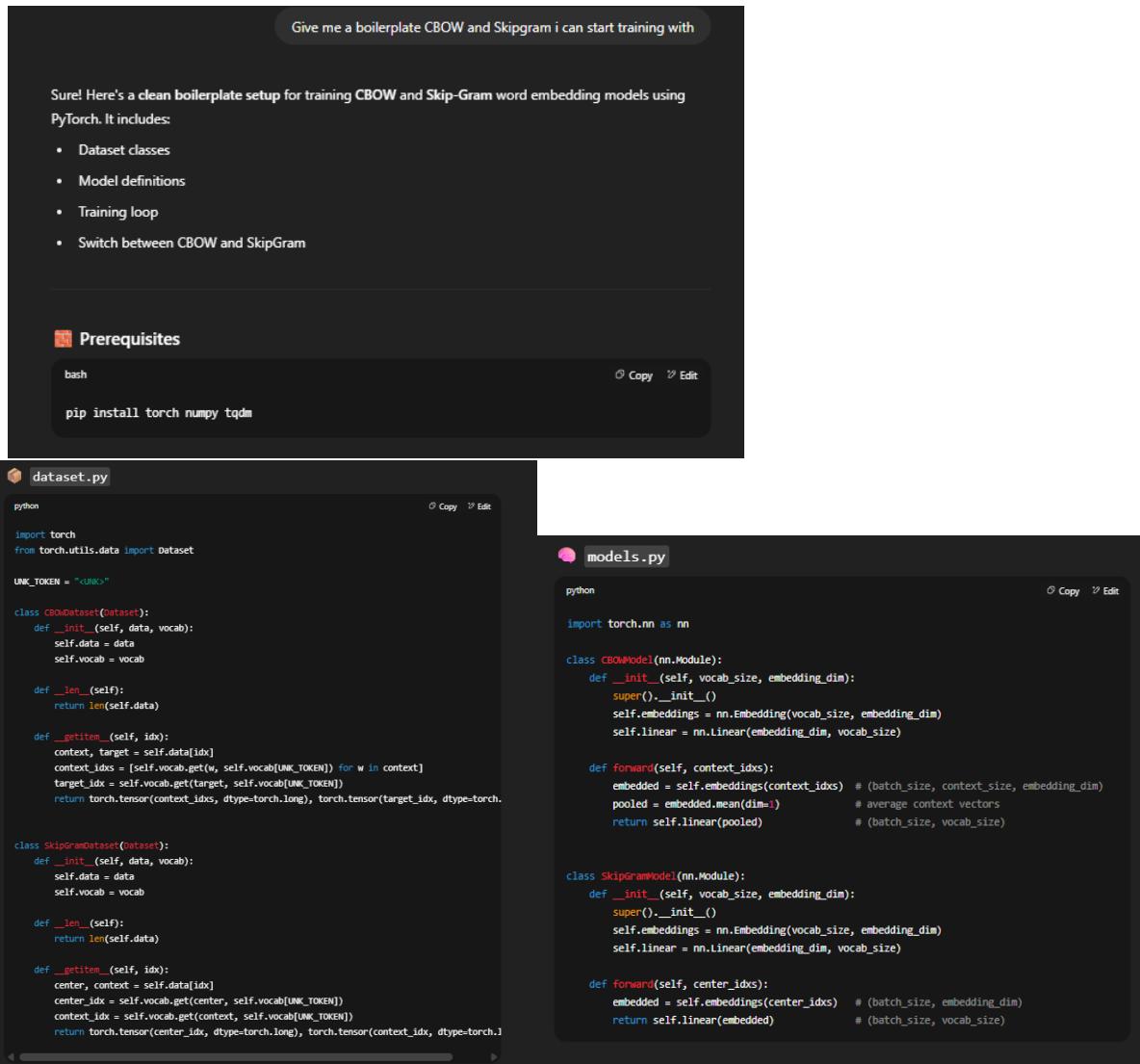
We construct a model that utilizes this main benefit of count data while simultaneously capturing the meaningful linear substructures prevalent in recent log-bilinear prediction-based methods like word2vec. The result, GloVe, is a new global log-bilinear regression model for the unsupervised learning of word representations that outperforms other models on word analogy, word similarity, and named entity recognition tasks.

Here's a concise 2-4 sentence summary of the key results from the literature you provided, suitable for inclusion in a literature review:

The GloVe model shows that increasing vector dimensions beyond 200 yields diminishing performance gains, and that syntactic tasks benefit from smaller, asymmetric context windows, while semantic tasks require larger, symmetric windows. Corpus size significantly affects syntactic performance, though Wikipedia outperforms larger corpora on semantic tasks due to richer entity coverage. In runtime comparisons, GloVe trains more efficiently than word2vec and consistently outperforms both CBOW and Skip-Gram in word analogy accuracy when trained on the same corpus and settings. Overall, GloVe demonstrates that combining the advantages of count-based statistics with predictive modeling leads to superior and faster word representations.

ChatGPT provided a starting point in establishing a boilerplate model to train. Refinements were done by referring to other sources like:

- [pytorch-continuous-bag-of-words/cbow.py at master · FraLotito/pytorch-continuous-bag-of-words · GitHub](#)
 - [Skip-Gram Word2Vec Algorithm Explained | by Ido Leshem | Medium](#)



ChatGPT helped during error debugging, pointing out that the Colab training in the evaluation script needed to be loaded correctly.

```
C:\Users\steve\anaconda3\envs\cis5231\python.exe
C:/Users/steve/OneDrive/Desktop/CIS5231/4_tSNE_visualisation_and_
analysis.py
Traceback (most recent call last):
  File
"C:\Users\steve\OneDrive\Desktop\CIS5231\4_tSNE_visualisation_and_
analysis.py", line 48, in <module>
    model.load_state_dict(torch.load(filepath+filename))
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 1516, in load
    return _load(
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 2114, in _load
    result = unpickler.load()
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\_weights_only_unpickler.py", line 532, in load
    self.append(self.persistent_load(pid))
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 2078, in persistent_load
    typed_storage = load_tensor(
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 2044, in load_tensor
    wrap_storage = restore_location(storage, location)
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 698, in default_restore_location
    result = fn(storage, location)
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 636, in _deserialize
    device = _validate_device(location, backend_name)
  File "C:\Users\steve\anaconda3\envs\cis5231\lib\site-
packages\torch\serialization.py", line 605, in _validate_device
    raise RuntimeError(
RuntimeError: Attempting to deserialize object on a CUDA device but
torch.cuda.is_available() is False. If you are running on a CPU-only
machine, please use torch.load with map_location=torch.device('cpu')
to map your storages to the CPU.

Process finished with exit code 1
```

This error means that you're trying to load a PyTorch model that was saved on a **CUDA (GPU)** device, but you're running the code on a **CPU-only** machine (i.e., `torch.cuda.is_available()` is `False`).

Fix:

To load the model on CPU, you need to modify your `torch.load(...)` call like this:

```
python
model.load_state_dict(torch.load(filepath + filename, map_location=torch.device('cpu')))
```

Why this works:

The `map_location=torch.device('cpu')` tells PyTorch to remap all the GPU-stored tensors to the CPU, avoiding any need for CUDA availability.

Let me know if you'd also like help checking your model definition or verifying the model path.

Code-wise, ChatGPT helped as well in the design choices:

```
Do I need one hot encoding to design CBOW and SkipGram? here is  
my code: import torch.nn as nn  
  
class WordEmbeddingModel(nn.Module):  
    def __init__(self, vocab_size, embedding_dim):  
        super(WordEmbeddingModel, self).__init__()  
        self.embeddings = nn.Embedding(vocab_size, embedding_dim)  
  
    def get_embedding(self, input_ids):  
        return self.embeddings(input_ids)  
  
class CBOWModel(WordEmbeddingModel):  
    def __init__(self, vocab_size, embedding_dim):  
        super(CBOWModel, self).__init__(vocab_size, embedding_dim)  
        self.linear = nn.Linear(embedding_dim, vocab_size)  
  
    def forward(self, context_idxs):  
        # context_idxs: (batch_size, context_window*2)  
        # shape: (batch_size, context_size, emb_dim)  
        embeds = self.embeddings(context_idxs)  
        # average over context  
        context_embed = embeds.mean(dim=1)  
        out = self.linear(context_embed)  
        log_probs = F.log_softmax(out, dim=1)  
        return log_probs  
  
class SkipGramModel(WordEmbeddingModel):  
    def __init__(self, vocab_size, embedding_dim):  
        super(SkipGramModel, self).__init__(vocab_size, embedding_dim)  
        self.linear = nn.Linear(embedding_dim, vocab_size)  
  
    def forward(self, center_idxs):  
        # center_idxs: (batch_size)  
        # shape: (batch_size, emb_dim)  
        embeds = self.embeddings(center_idxs)  
        out = self.linear(embeds)  
        log_probs = F.log_softmax(out, dim=1)  
        return log_probs
```

No, you do not need one-hot encoding to implement CBOW or Skip-Gram in PyTorch — and your code is correctly designed without it.



✓ Why One-Hot Is Not Needed:

In older or theoretical implementations, one-hot vectors were used as input, and then multiplied by a weight matrix to get the embedding. But in PyTorch (and other modern frameworks):

- You use `nn.Embedding(vocab_size, embedding_dim)`.
- You simply pass word indices as input.
- The `nn.Embedding` layer internally does what one-hot + matrix multiplication would do — more efficiently.

This is not just convenient — it's also faster and more memory-efficient, since one-hot vectors are large and sparse.

➊ Summary of What You Did Right:

Part	Your Implementation	What It Replaces
<code>nn.Embedding</code>	Learns dense embeddings directly from indices	Replaces one-hot + weight matrix
<code>context_idxs</code> / <code>center_idxs</code>	Tensor of word indices	Would be one-hot vectors in older methods
<code>nn.Linear</code> + <code>F.log_softmax</code>	Maps embeddings to vocab size logits	Used for predicting target word(s)

✳ Tiny Suggestion:

Add `import torch.nn.functional as F` at the top, if you haven't already, since you're using `F.log_softmax`.

✓ Verdict:

Your CBOW and Skip-Gram classes are well-structured and do not need one-hot encoding. You're using the standard and preferred PyTorch pattern. You're good to go!



ChatGPT also helped to provide Pytorch lines of code that help with acceleration:

The screenshot shows a ChatGPT interface with a dark theme. On the left, there is a code editor containing Python code for a SkipGramModel and its dataset. On the right, there are three sections of advice with checkboxes:

- 1. Use PyTorch's CPU optimizations**:
A checkbox is checked. Below it is a snippet of Python code:

```
python
torch.set_num_threads(torch.get_num_threads()) # Use all available CPU threads
torch.backends.mkldnn.enabled = True # Enable MKLDNN (optimized CPU kernels)
```
- 2. Use DataLoader for efficient batching**:
A checkbox is checked. Below it is a snippet of Python code:

```
python
from torch.utils.data import DataLoader

# Assuming `dataset` is your SkipGramDataset
dataloader = DataLoader(dataset, batch_size=512, shuffle=True, num_workers=4)

# num_workers > 0 enables parallel data loading.
# Adjust batch_size depending on your RAM capacity.
```
- 3. Vectorize loss computation**:
A checkbox is checked. Below it is a snippet of Python code:

```
Instead of computing log-probabilities and loss manually, use PyTorch's efficient nn.NLLLoss or nn.CrossEntropyLoss directly.
```

At the bottom right, there is a link labeled "Optimize your model".

Summary: Code Optimization Cheatsheet

This screenshot shows a summary of code optimization techniques. It features a code editor with a dark background containing Python code for training a neural network. At the bottom, there is a message and a set of navigation icons.

```
python
torch.set_num_threads(torch.get_num_threads())
torch.backends.mkldnn.enabled = True
model = torch.compile(model) # If using PyTorch 2.0+
dataloader = DataLoader(dataset, batch_size=512, shuffle=True, num_workers=4)

criterion = nn.CrossEntropyLoss()
for center_idxs, context_idxs in dataloader:
    center_idxs = center_idxs.to(device)
    context_idxs = context_idxs.to(device)

    optimizer.zero_grad()
    output = model(center_idxs)
    loss = criterion(output, context_idxs)
    loss.backward()
    optimizer.step()
```

Let me know if you want a fully optimized training loop sample or integration with CPU-specific quantization techniques.

Navigation icons: back, forward, search, etc.

Ideas were provided for future enhancements:

- what other improvements/enhancements are there for CBOW/Skipgram experimentation? This is what I have so far
- Possible Enhancements/Further work
- Feed larger datasets as per the literature review (6 billion) and
 - Decrease the epoch count
 - Better data preprocessing regimen
 - Make use of different models as GloVe or ELMo
 - Make use of a different optimiser, like Adagrad, used by Mikolov et al. [4]
 - Make use of a different loss function, instead of CrossEntropyLoss
 - Make more use of GPUs
 - Further hyperparameter experimentation:
 - Mini-batches can be made smaller for CBOW
 - Assess with a larger dataset if this separates clusters better
 - Further window sizes for SkipGram
 - Assess with a larger dataset if this separates clusters better
 - Analyse quality of embeddings through word similarity and analogy evaluation, using same datasets as literature



Your list of enhancements is already strong and well-informed. Here are **additional enhancements** and **refinements** you can include to elevate your CBOW/Skip-gram experimentation and align better with state-of-the-art practices:

Architectural & Training Enhancements

1. Negative Sampling or Hierarchical Softmax

- Replace full softmax with:
 - **Negative Sampling:** Sample a few negative examples instead of computing loss over the entire vocab.
 - **Hierarchical Softmax:** Speeds up training by using a binary tree.
- Both were used in Mikolov et al.'s original word2vec.

2. Subsampling Frequent Words

- Remove high-frequency words (e.g., "the", "is") to reduce noise.
- Improves embedding quality for rarer, more meaningful words.

3. Position-Weighted Context

- Give closer context words higher weight (as done in some GloVe variants).
- Especially helpful in CBOW.

4. Context Concatenation Instead of Averaging

- Instead of averaging context vectors in CBOW, **concatenate** them (increases model expressiveness).

Evaluation Enhancements

5. Quantitative Evaluation on Benchmark Datasets

- Use standard datasets:
 - WordSim-353
 - MEN dataset
 - Google analogy dataset
- Evaluate cosine similarity and analogy-solving performance.

6. Intrinsic vs. Extrinsic Evaluation

- Intrinsic: similarity/analogy benchmarks.
- Extrinsic: downstream task performance (e.g., text classification, NER, sentiment analysis).

7. Cluster Quality Metrics

- Use Silhouette Score, Davies-Bouldin Index, or DBSCAN to assess cluster quality in embedding space after t-SNE.

Training and Infrastructure Enhancements

8. Use Mixed Precision Training

- If using GPU: leverage `torch.cuda.amp` for faster, memory-efficient training.

9. Train on GPU with Multi-threaded Dataloader

- Boost performance with GPU + `num_workers > 4`.

10. Early Stopping and Learning Rate Scheduling

- Use **early stopping** to avoid overfitting.
- **Scheduler** (like `ReduceLROnPlateau` or cosine annealing) to fine-tune training dynamics.

Model & Feature Enhancements

11. Character-level Embeddings or Subword Units

- Use **FastText-like** architecture to handle rare and OOV (Out of Vocabulary) words.

12. Multi-lingual or Cross-lingual Embedding Training

- Extend to **multi-lingual corpora** with alignment objectives.

13. Fine-tuning Pretrained Word Embeddings

- Load GloVe/word2vec embeddings as initialization, then fine-tune on your domain corpus (e.g., sports/business).

14. Domain-Specific Stopword Removal

- Build a custom stopword list that aligns with your category (e.g., "team", "game" in sports).

Visualization Enhancements

15. Animate t-SNE Over Epochs

- Create a t-SNE animation to observe **embedding space evolution** across training.

16. Interactive Visualizations

- Use **TensorBoard**, **Plotly**, or projector.tensorflow.org for embedding exploration.

Would you like this refined into a bulletproof section for a report or presentation?



English check:

improve my english :

- Hyperparameter optimisation was carried out with both CPUs and GPUs offered by Google Colab and Kaggle. Despite that, time constraints were a problem, and further hyperparameter optimisation was not possible.
- Establishing the right epoch size/batch size/window size
- Interactive tSNE plots (using plotly package) and kNN plots were being added. Due to the size of the vocabulary and hardware requirements, this was not achieved, and errors arose.
- How to evaluate which model was the best performing was an issue.

Here's an improved and more professional version of your text with clearer phrasing and grammar:

- Hyperparameter optimization was performed using both CPU and GPU resources available through Google Colab and Kaggle. However, due to time constraints, extensive tuning was not feasible.
- Determining the optimal values for epoch count, batch size, and context window size proved to be a challenge.
- Interactive t-SNE plots (via the Plotly package) and k-NN visualizations were planned, but could not be completed due to the large vocabulary size and limited hardware capabilities, which resulted in errors.
- Evaluating which model performed best was also a challenge, as consistent and meaningful comparison metrics were difficult to establish.

