Technical Report

Word2Vec Embedding Explorer

Building a From-Scratch NLP System: Training, Serialization & Live Deployment

Component	Technology	Detail
Model	PyTorch	Skip-Gram with Negative Sampling
Dataset	WikiText-2	29,119 sentences, ~2M tokens
Vocabulary	29,780 tokens	min_freq=5, sorted by frequency
Embeddings	100-dimensional	L2-normalised, exported to JSON
Deployment	GitHub Pages	Static HTML, zero backend
Interface	Vanilla JS + Plotly	Cosine sim, NN search, analogy
Checkpoints	Epoch 2, 5, 10	Top 5,000 nouns & adjectives

1. Introduction

This report documents the design, implementation, training, and deployment of a Word2Vec embedding system built from scratch. The project spans two interconnected artefacts: a Google Colab notebook implementing the full training pipeline in PyTorch, and a browser-based interactive visualisation deployed on GitHub Pages.

The central motivation was to go beyond notebook-level experimentation. Training a model produces numbers. Deploying it as an interactive system produces understanding — both for the builder and for anyone who encounters it. This project was designed to make the learning dynamics of word embeddings visible, navigable, and tangible.

> *Models are powerful. Systems are transformative. The gap between 'I trained a model' and 'I built something around it' is where real understanding happens.*

The pipeline is implemented entirely from first principles — no pre-built Word2Vec wrappers, no gensim, no shortcuts. Every component (tokenisation, frequency filtering, subsampling, pair generation, negative sampling distribution, batched training loop, JSON export, client-side inference) is written and reasoned through explicitly.

2. Dataset & Preprocessing

2.1 Dataset

The WikiText-2 dataset (detokenised variant) was loaded from HuggingFace Datasets. It contains Wikipedia article text split into train, validation, and test partitions.

Split	Sentences	Headings	Empty
Train	23,767	6,211	0
Validation	2,461	620	0
Test	2,891	708	0
Combined	29,119	7,539	0

All three splits were concatenated into a single training corpus. WikiText-2 uses `=` characters to denote article section headings (e.g. `= Valkyria Chronicles III =`). These are not natural language sentences and were removed.

2.2 Cleaning

After heading removal and lowercasing, hyphens were replaced with spaces using `str.maketrans('-', ' ')`. This treats hyphenated compounds (e.g. 'role-playing') as two separate tokens, consistent with the whitespace tokenisation strategy used throughout. Post-cleaning the corpus contains 21,580 usable sentences with no empty lines and no heading markers.

2.3 Tokenisation

Tokenisation uses Python's `str.split()` — whitespace splitting with no punctuation stripping. This is a deliberate design choice consistent with the original Word2Vec implementation. Punctuation attached to words (e.g. 'game,', 'album."') becomes part of the token. This produces a raw vocabulary of 145,192 unique tokens.

2.4 Frequency Filtering

Words appearing fewer than 5 times in the corpus are removed. This eliminates noise from hapax legomena and near-unique tokens that cannot develop meaningful embedding geometry from so few training signals.

Metric	Value
Raw vocabulary size	145,192
Words with freq < 5	115,412 (removed)
Filtered vocabulary size	29,780
Minimum word frequency	5 (e.g. 'senjō')

Metric	Value
Total tokens remaining	1,935,519

The filtered vocabulary is sorted in descending frequency order to assign indices. The five most frequent words are: the (159,625) · of (69,265) · and (60,523) · in (53,869) · to (48,198). These are the words that will be most aggressively subsampled in the next phase.

3. Frequency Subsampling

Extremely common words — function words like 'the', 'of', 'in' — appear in almost every context window. They provide little discriminative information about the meaning of surrounding words. Training on them wastes computation and dilutes the signal from rarer, more semantically meaningful words.

Following Mikolov et al. (2013), each word w is stochastically discarded from the corpus during preprocessing with probability:

```
P_discard(w) = max(0, 1 − √(t / f(w)))

where:
  f(w) = frequency of word w in the filtered corpus
  t    = subsampling threshold hyperparameter = 1e-5
```

For 'the' (f = 0.0825), P_discard ≈ 0.989 — it is discarded ~99% of the time. For rare words at the frequency boundary (f ≈ 5/1,935,519 ≈ 2.6e-6), P_discard = 0 — they are never discarded.

Metric	Before	After
Total tokens	1,935,519	568,118
Sentences	21,580	20,475
'the' frequency	~159,625	~1,800

Subsampling is applied with `random.seed(42)` for reproducibility. Sentences that fall below 2 tokens after subsampling are discarded entirely, since no (center, context) pairs can be formed.

> *An important design decision: the negative sampling probability distribution is computed from the PRE-subsampling frequency counts (updated_word_count). This is correct.*

> *Subsampling artificially reduces the corpus frequency of common words, which would distort the negative sampling distribution if computed post-subsampling.*

4. Training Architecture

4.1 Corpus Indexing

After subsampling, each word in every sentence is replaced with its integer index from word2idx. This converts the corpus from a list of string lists to a list of integer lists — the form required by PyTorch's nn.Embedding layers.

4.2 Sliding Window Pair Generation

The Skip-Gram objective trains on (center word, context word) pairs. A sliding window of size 3 is passed over each sentence. For a center word at position i, all words within positions [i−3, i+3] (excluding i itself) are valid context words.

```
window_size = 3

for sentence in indexed_corpus:
    for i, center in enumerate(sentence):
        for j in range(max(0, i-3), min(len(sentence), i+4)):
            if i != j:
                pairs.append((center, sentence[j]))

Total pairs generated: 3,164,394
```

All pairs are pre-computed and stored in a flat list. This trades memory for speed — avoiding repeated generator traversal during each epoch.

4.3 Negative Sampling Distribution

For each positive (center, context) pair, k=5 negative context words are sampled. The sampling distribution follows the formula from Mikolov et al.:

```
P_neg(w) ∝ f(w) ** 0.75

freq_tensor = torch.tensor(freq_list)  # shape (V,)
freq_tensor = freq_tensor ** 0.75
neg_sampling_probs = freq_tensor / freq_tensor.sum()   # normalised

# Sampling at training time:
neg_idxs = torch.multinomial(neg_sampling_probs,
```

```
                                num_samples=batch_size * k,
                                replacement=True).view(batch_size, k)
```

The 0.75 exponent compresses the distribution — it gives rare words a higher relative sampling probability than their raw frequency would suggest, while still favouring common words overall. The validation that neg_sampling_probs.sum() == 1 is confirmed.

4.4 SkipGram Model — Two Implementations

The notebook develops the model in two stages, showing the evolution from a single-pair formulation to an efficient batched version.

Single-pair (Vanilla)

```
class SkipGram(nn.Module):
    def __init__(self, vocab_size, embed_dim):
        self.input_embeddings  = nn.Embedding(vocab_size, embed_dim)
        self.output_embeddings = nn.Embedding(vocab_size, embed_dim)
        nn.init.normal_(self.input_embeddings.weight,  mean=0, std=0.01)
        nn.init.normal_(self.output_embeddings.weight, mean=0, std=0.01)

    def forward(self, center_idx, pos_idx, neg_indices):
        center_vec = self.input_embeddings(center_idx)   # (d,)
        pos_vec    = self.output_embeddings(pos_idx)     # (d,)
        neg_vecs   = self.output_embeddings(neg_indices) # (k, d)

        pos_score  = torch.dot(center_vec, pos_vec)
        neg_score  = torch.matmul(neg_vecs, center_vec)  # (k,)

        loss = -(F.logsigmoid(pos_score) + F.logsigmoid(-neg_score).sum())
        return loss
```

Batched (Final)

The batched version processes B pairs simultaneously, enabling GPU parallelism and significantly faster training:

```
    def forward(self, center_idxs, pos_idxs, neg_idxs):
        # center_idxs: (B,)  |  pos_idxs: (B,)  |  neg_idxs: (B, k)

        center_vecs = self.input_embeddings(center_idxs)   # (B, d)
        pos_vecs    = self.output_embeddings(pos_idxs)     # (B, d)
        neg_vecs    = self.output_embeddings(neg_idxs)     # (B, k, d)

        pos_scores  = torch.sum(center_vecs * pos_vecs, dim=1)          # (B,)
        neg_scores  = torch.sum(neg_vecs * center_vecs.unsqueeze(1), 2) # (B, k)
```

```
        loss_pos    = F.logsigmoid(pos_scores)
        loss_neg    = F.logsigmoid(-neg_scores).sum(dim=1)
        return -(loss_pos + loss_neg).mean()
```

Two separate embedding matrices are used — `input_embeddings` for center words and `output_embeddings` for context and negative words. This is the standard Word2Vec architecture. At inference time, only `input_embeddings` is used.

4.5 Training Configuration

Hyperparameter	Value	Rationale
Embedding dimension	100	Standard Word2Vec size; balances capacity vs. compute
Window size	3	±3 words around center; captures local context well
Negative samples k	5	Recommended range 5–20; sufficient for this vocab size
Batch size	256	Efficient GPU utilisation on Colab T4
Epochs	10	Sufficient convergence; checkpoints at 2, 5, 10
Optimiser	Adam	lr = 0.001; adaptive, robust to learning rate tuning
Device	CUDA	Google Colab T4 GPU

4.6 Training Loss

Epoch	Avg Loss	Δ Loss
1	2.7062	—
2	2.3077	−0.3985 ← checkpoint saved
3	2.0196	−0.2881
4	1.8137	−0.2059
5	1.6730	−0.1407 ← checkpoint saved
6	1.5796	−0.0934
7	1.5151	−0.0645
8	—	converging
9	—	converging
10	—	← checkpoint saved

The loss decreases rapidly in the first five epochs and slows thereafter — a classic pattern indicating the model has learned the bulk of the distributional signal and is refining lower-frequency

associations. The diminishing delta is not a sign of failure; it reflects that the most common co-occurrence patterns are captured early.

5. Checkpoint Export

5.1 POS Filtering

The full vocabulary of 29,780 words is not exported. Many tokens are punctuation-attached artifacts of the whitespace tokenisation strategy (e.g. 'game,', 'album.'"). Instead, NLTK's averaged perceptron tagger is used to perform Part-of-Speech tagging on the entire vocabulary, and the top 5,000 tokens classified as nouns or adjectives (NN, NNS, NNP, NNPS, JJ, JJR, JJS) by frequency are selected.

This selection strategy ensures the demo vocabulary is semantically rich — nouns and adjectives carry the most meaning-bearing content — while keeping JSON file sizes manageable for browser loading.

5.2 Normalisation & PCA

```
# L2-normalise so cosine similarity = dot product in the browser
norms       = np.linalg.norm(vecs, axis=1, keepdims=True)
vecs_normed = vecs / np.where(norms == 0, 1e-8, norms)

# PCA to 2D for the scatter plot
pca       = PCA(n_components=2, random_state=42)
vecs_2d   = pca.fit_transform(vecs_normed)
```

L2-normalisation is applied before export so that all dot products in the browser are equivalent to cosine similarity — no division by norms needed at runtime, enabling fast nearest-neighbour search in vanilla JavaScript.

5.3 JSON Schema

```
{
  "epoch":       10,
  "vocab_size":  5000,
  "embed_dim":   100,
  "pca_variance": [0.042, 0.031],
  "words":       ["science", "music", ...],
  "embeddings":  { "science": [0.23, -0.87, ..., 0.64] },
  "coords_2d":   { "science": [-0.41, 1.23] }
}
```

Each file is serialised with compact separators (`separators=(',', ':')`) to minimise file size. The three files (embeddings_epoch2.json, embeddings_epoch5.json, embeddings_epoch10.json) are the sole interface between the Python training environment and the JavaScript runtime.

6. Model Evaluation

6.1 Cosine Similarity

After training, embeddings are L2-normalised in-place. Cosine similarity is then computed as a dot product:

```
embeddings      = model.input_embeddings.weight.detach()
norm_embeddings = embeddings / embeddings.norm(dim=1, keepdim=True)

def get_similarity(word1, word2):
    idx1, idx2 = word2idx[word1], word2idx[word2]
    return torch.dot(norm_embeddings[idx1], norm_embeddings[idx2]).item()

get_similarity('player', 'game')  # → 0.4444
```

6.2 Nearest Neighbour Search

Nearest neighbour search is implemented as a matrix-vector multiply, computing cosine similarity between a query word and every word in the vocabulary simultaneously:

```
def get_nearest(word, top_k=10):
    query_vec    = norm_embeddings[word2idx[word]]
    similarities = torch.matmul(norm_embeddings, query_vec)  # (V,)
    top_indices  = torch.topk(similarities, top_k+1).indices.tolist()
    return [idx2word[i] for i in top_indices if i != word2idx[word]][:top_k]
```

Query Word	Top Neighbors (Epoch 10)
gaming	sequel, famitsu, informer, visuals, reprise, puzzles, esports, dota, pc, gameplay.
monarchs	parthian, constantinople, greeks, palmyrene, rebellion., rulers, pope,, esarhaddon
music	awards, recordings, musical, mccartney, album.", shimomura's, christmas"

6.3 Analogy Reasoning

Vector arithmetic for analogy solving:

```
def analogy(a, b, c, top_k=5):
    vec = norm_embeddings[word2idx[a]]
          - norm_embeddings[word2idx[b]]
          + norm_embeddings[word2idx[c]]
    similarities = torch.matmul(norm_embeddings, vec)
    top_indices  = torch.topk(similarities, top_k).indices.tolist()
    return [idx2word[i] for i in top_indices]

analogy('sony', 'shimomura', 'video')  # → ['video', 'sony', 'rockstar',
'online']
```

The analogy results reflect the corpus domain. WikiText-2 is Wikipedia text about video games, history, and music — so the model finds analogies grounded in those domains. The classic 'king − man + woman = queen' test is not available as these words fall outside the top-5000 noun/adjective filter, but domain-appropriate analogies (e.g. within the gaming or music vocabulary) are demonstrable.

6.4 What the Data Reveals

The nearest neighbors of 'science' at Epoch 10 are: fiction, magazines, stories, children's, web, pohl, tv, technology, publishers, issues. These are not the physics/chemistry neighbors one might expect. They reflect how Wikipedia discusses science — as a publishing and media domain, not a technical one.

> *This is one of the most instructive findings of the project: embedding geometry is a mirror of the training corpus, not the world. The model learned exactly what it was shown. Changing the corpus — to academic papers, textbooks, or research abstracts — would produce fundamentally different neighbors for the same word.*

6.5 PCA Visualisation

A static PCA scatter plot is generated in the notebook using matplotlib, showing 33 words across three semantic clusters (gaming, monarchs, music) with their nearest neighbours. Cluster boundaries are drawn using matplotlib.patches.Circle, with centroids computed from the mean of cluster coordinates. This notebook visualisation is the prototype for the interactive version in the web demo.

7. Interactive Web Demo

7.1 Architecture

The demo is a single self-contained HTML file (word2vec_demo_epochs.html) with no server-side component. On load, it fetches the three JSON checkpoint files, caches them in memory, and runs all computations in the browser using vanilla JavaScript.

Concern	Solution
Hosting	GitHub Pages (static file serving)
Visualisation	Plotly.js (CDN, single external dependency)
State management	Plain JS objects; all epoch data cached in memory after first load
Inference	Client-side dot products on pre-normalised vectors
Epoch switching	Swap active dataset reference; re-render all panels
Fallback	Mock embeddings if JSON files unavailable — demo still functional

7.2 Feature Panels

Explore Space — PCA Scatter

An interactive Plotly 2D scatter plot renders all 5,000 words coloured by semantic category (auto-assigned from seed word lists). Words are positioned using the PCA 2D coordinates pre-computed during export. Cluster visibility is toggleable via a custom legend. Switching epochs redraws the plot with new coordinates, making the cluster formation visible.

Similarity

Users enter any two words in the vocabulary and receive their cosine similarity score, a colour-coded interpretation (Strongly/Moderately/Weakly related / Dissimilar), and a cross-epoch comparison bar chart showing how the same pair's similarity evolves from epoch 2 to epoch 10. This is the most direct visualisation of learning dynamics.

Nearest Neighbors

Top-k nearest neighbour search for any query word, rendered as animated horizontal bars (cosine similarity scores) with clickable results. Switching epochs reshuffles and re-ranks the neighbor list, showing how the model's notion of 'similar context' stabilises.

Analogy Solver

Users enter three words A, B, C. The demo computes A − B + C in embedding space, L2-normalises the result, and returns the top 5 nearest words. A live equation display updates as the user types. Pre-built example analogies are provided.

How It Works

An educational tab explaining embeddings, Skip-Gram, negative sampling, frequency subsampling, cosine similarity, and vector arithmetic — with the actual mathematical formulas from the paper, tailored to the specific implementation in this project.

7.3 Epoch Switching

A sticky epoch switcher bar (always visible, independent of the active tab) allows users to switch between Epoch 2, Epoch 5, and Epoch 10. All five panels update synchronously. JSON files are loaded lazily — each epoch's data is fetched once on first access and cached.

```
async function switchEpoch(ep) {
    if (!epochData[ep]) {
        const res = await fetch(`embeddings_epoch${ep}.json`);
        epochData[ep] = await res.json();
    }
    activeEpoch = ep;
    activeData  = epochData[ep];
    updateAllUI();   // redraws scatter, similarity, neighbors, analogy
}
```

7.4 Client-Side Inference

All mathematical operations are reimplemented in JavaScript. Since vectors are pre-normalised, cosine similarity reduces to a dot product:

```
function cosine(a, b) {
    let dot = 0;
    for (let i = 0; i < a.length; i++) dot += a[i] * b[i];
    return Math.max(-1, Math.min(1, dot));
}

// Nearest neighbour search:
const scores = Object.entries(embeddings)
    .filter(([w]) => w !== query)
    .map(([w, emb]) => [w, cosine(queryVec, emb)])
    .sort((a, b) => b[1] - a[1])
    .slice(0, k);
```

7.5 Deployment

The demo is deployed on GitHub Pages by placing the HTML file and three JSON files in the repository root and enabling Pages from the repository settings. The entire system is static — there is no build step, no server, no API. A reader who forks the repository and enables Pages gets a working demo immediately.

8. Key Observations & Insights

8.1 Training Dynamics Across Epochs

The most striking observation from the multi-epoch comparison is that neighbour scores for 'science' are highest at epoch 2 and decrease by epoch 10. The interpretation is counterintuitive on first reading but correct: early-epoch models assign high similarity scores broadly because the embeddings have not yet differentiated. Later-epoch models are more selective — they assign high similarity only to genuinely related words, and lower scores to the rest. The model has learned what 'science' is NOT near, which is exactly right.

Epoch	Top Neighbor (science)	Score	Observation
2	fiction	0.756	High but noisy — duplicates, tokenisation artifacts
5	fiction	0.653	Scores settling, list cleaning up
10	fiction	0.570	'technology' surfaces — semantically correct neighbor

8.2 Corpus-Embedded Geometry

The demo makes explicit something that is easy to forget when using pre-trained embeddings: the geometry of an embedding space is a direct reflection of the training corpus. WikiText-2's treatment of 'science' as a publishing domain (neighbors: fiction, magazines, stories) is not a defect — it is an accurate representation of how Wikipedia discusses science. This is a lesson that transfers directly to production contexts: understanding what your embeddings learned requires understanding what your data contained.

8.3 From PyTorch to JavaScript

The most substantial systems engineering challenge was the bridge between the training environment (PyTorch, Python, GPU) and the deployment environment (browser, JavaScript, no backend). Every mathematical operation had to be reimplemented. The key insight that made this tractable: pre-normalise vectors during export so that all browser-side computations are simple dot products with no division. This reduces a potentially complex numerical operation to a tight inner loop.

8.4 LLM-Assisted Development

Large language models were used as collaborators during the frontend development phase — generating UI structure, debugging JavaScript edge cases, suggesting improvements to the JSON schema. This is noted explicitly because it represents an honest account of how modern development increasingly works. The architectural and educational decisions (multi-epoch comparison, the POS filter, the cross-epoch similarity panel) were human-driven. LLMs accelerated execution without supplanting judgement.

9. Limitations & Future Work

9.1 Current Limitations

- Whitespace tokenisation leaves punctuation attached to words ('game,' appears as a separate token from 'game'). A proper tokeniser (e.g. spaCy) would clean this.
- WikiText-2 is a small corpus (~2M tokens). Larger corpora (WikiText-103, Common Crawl) would produce higher-quality embeddings with richer neighbourhood geometry.
- The demo vocabulary is fixed at export time. Dynamic vocabulary expansion would require re-training or incremental updates.
- No evaluation against standard word similarity benchmarks (WordSim-353, SimLex-999) is included — these would allow quantitative comparison with published results.
- The PCA projection discards 95%+ of the variance in 100-dimensional space. UMAP or t-SNE would produce more faithful 2D representations of local structure.

9.2 Future Directions

- Phase 2: Train on a larger corpus (WikiText-103) and compare embedding quality.
- Train for more epochs and observe whether additional convergence produces meaningfully better neighbours.
- Add a UMAP projection tab to the demo as an alternative to PCA.
- Implement an evaluation tab with standard analogy test sets (Google Analogy Dataset).
- Add a model checkpoint upload feature — allow users to drop their own .pt file and explore their own trained embeddings.
- Extend to FastText (subword embeddings) and compare neighbourhood geometry for morphologically complex words.

10. Conclusion

This project demonstrates that building a complete NLP system — from raw data to a deployed interactive tool — is achievable as a learning exercise, and that the process of doing so produces understanding that training alone cannot.

The Word2Vec implementation follows the original paper faithfully: sliding window pair generation, frequency-based subsampling, negative sampling with the 0.75-power distribution, separate input and output embedding matrices. The training dynamics (3.16M pairs per epoch, converging from loss 2.71 to ~1.5 over 10 epochs) are consistent with what is expected for a corpus of this size.

The multi-epoch interactive demo makes the most important insight visible without requiring the viewer to understand the mathematics: embeddings learn. The geometry changes. Early epochs are scattered and noisy. Later epochs show structure. That progression — from random initialisation to semantic neighbourhood — is the core phenomenon that Word2Vec (and all distributional representation learning) is built on.

> *The progression from model → pipeline → serialisation layer → deployed interactive tool is not just a software engineering exercise. Each step forces a different kind of understanding.*

> *Together, they produce something neither a notebook nor a deployment alone could: a system that makes learning visible.*

The live demo is accessible at:

ruthuraraj-ml.github.io/Embedding_Search/

References

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G., & Dean, J. (2013). Distributed Representations of Words and Phrases and their Compositionality. arXiv:1310.4546.

Merity, S., Xiong, C., Bradbury, J., & Socher, R. (2016). Pointer Sentinel Mixture Models. arXiv:1609.07843. [WikiText-2 dataset]

HuggingFace Datasets: dlwh/wikitext_2_detokenized. https://huggingface.co/datasets/dlwh/wikitext_2_detokenized

NLTK: Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media.