Word Embedding Reduction and Fine-Tuning for Semantic Enhancement

Rayan Hanader Sami Taider

December 27, 2024

Abstract

This report presents our attempt to reduce word embeddings from 300 to 30 dimensions while maintaining semantic quality. Although the 30-dimensional embeddings do not surpass the 300-dimensional ones, the results are promising for a 10x reduction. We discuss the methodology, challenges, and key findings, as well as provide an outlook for future improvements.
The whole project is publicly available on github `(here)`

1 Introduction

The project aimed to explore efficient reduction of high-dimensional word embeddings and evaluate their performance after fine-tuning. By compressing FastText's 300-dimensional French word embeddings to 30 dimensions, we sought to achieve semantic closeness while saving computational resources.

Our approach involved creating an AutoEncoder for dimensionality reduction, training a Word2Vec model for fine-tuning, and evaluating the embeddings through cosine similarity and clustering visualization.

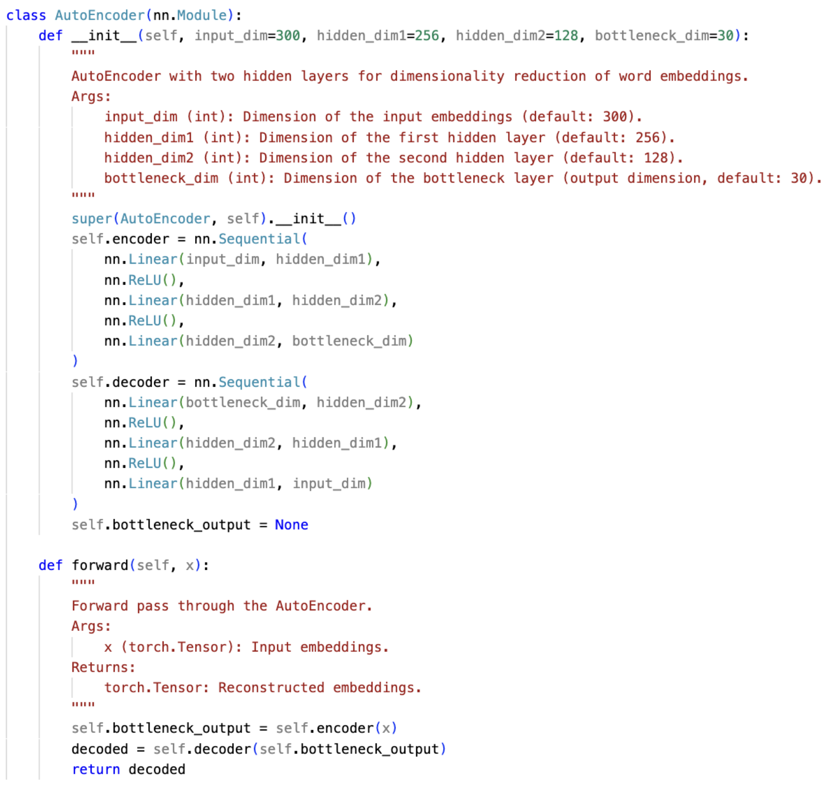
2 Methodology

2.1 AutoEncoder Architecture

In order to surpass the original embeddings, we aimed to achieve the best possible reduction. To do so, we initially planned to :

- Try multiple AutoEncoder architectures to choose the best one.
- Test all the hyper-parameters combinations to keep the best for optimal training.

However, due to time constraints, we focused solely on hyper-parameter tuning. We adopted a simple two-layer architecture, as illustrated in the following screenshot:



```python
class AutoEncoder(nn.Module):
    def __init__(self, input_dim=300, hidden_dim1=256, hidden_dim2=128, bottleneck_dim=30):
        """
        AutoEncoder with two hidden layers for dimensionality reduction of word embeddings.
        Args:
            input_dim (int): Dimension of the input embeddings (default: 300).
            hidden_dim1 (int): Dimension of the first hidden layer (default: 256).
            hidden_dim2 (int): Dimension of the second hidden layer (default: 128).
            bottleneck_dim (int): Dimension of the bottleneck layer (output dimension, default: 30).
        """
        super(AutoEncoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, bottleneck_dim)
        )
        self.decoder = nn.Sequential(
            nn.Linear(bottleneck_dim, hidden_dim2),
            nn.ReLU(),
            nn.Linear(hidden_dim2, hidden_dim1),
            nn.ReLU(),
            nn.Linear(hidden_dim1, input_dim)
        )
        self.bottleneck_output = None

    def forward(self, x):
        """
        Forward pass through the AutoEncoder.
        Args:
            x (torch.Tensor): Input embeddings.
        Returns:
            torch.Tensor: Reconstructed embeddings.
        """
        self.bottleneck_output = self.encoder(x)
        decoded = self.decoder(self.bottleneck_output)
        return decoded
```

Figure 1: AutoEncoder Architecture

The tuning process was conducted by testing values for the 2 hidden layers of the AutoEncoder, as well as for the learning rate and the batch size. After comparing test losses and cosine similarity, we selected the final configuration:

- `hidden_dim1 = 256`
- `hidden_dim2 = 128`
- `learning_rate = 0.001`
- `batch_size = 64`

2.2 Data Engineering Pipeline Creation

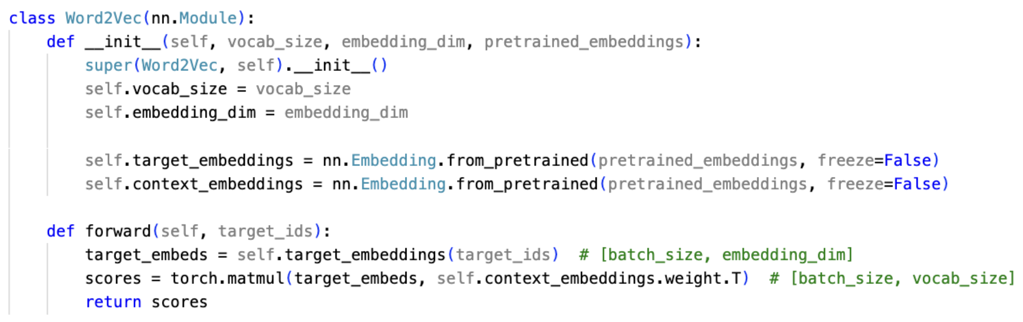
- Wikipedia corpora were fetched using the `wikipedia-api` library and pre-processed with `SpaCy`.

- We designed Skip-gram word pairs construction algorithm for a window-size of 2 based on the Wikipedia corpora. Due to time constraints, we couldn't achieve to properly divide the corpora into phrases for a more accurate Skip-gram division. This leads to an unperfect contextualization of each words, but is something we would definitly fix for global results improvment.

- Testing datasets, containing semantically-close word pairs, were generated using LLMs (GPT-4o and Gemini 1.5 Flash) based on the corpus. We wanted to use STS state-of-the-art benchworks, but as you will see in the next part, we could not achieve to train the entire vocabulary, hence the benchworks would contain too many words that would not exist in our embeddings matrix.

2.3 Embedding Fine-Tuning

Our Word2Vec model takes, as wanted, the 30-dimensions Embeddings reduced from the AutoEncoder, and gives out the same structure, but improved thanks to the Skip-gram pairs. Here is the global structure.



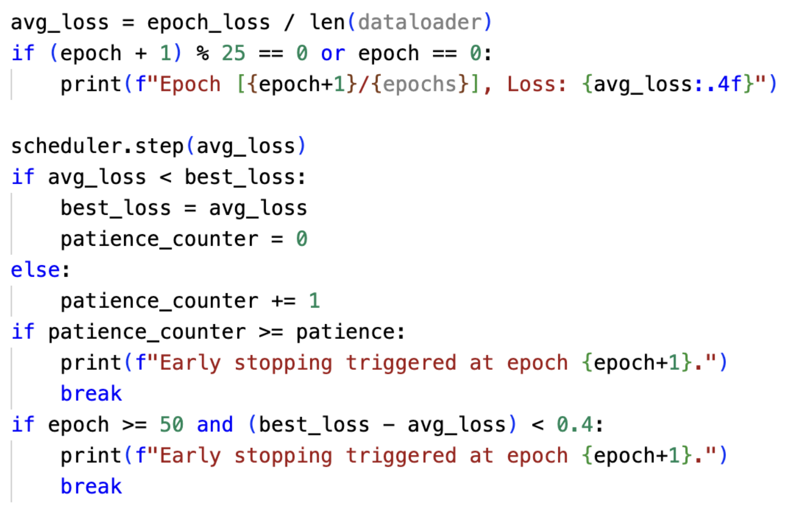
```python
class Word2Vec(nn.Module):
    def __init__(self, vocab_size, embedding_dim, pretrained_embeddings):
        super(Word2Vec, self).__init__()
        self.vocab_size = vocab_size
        self.embedding_dim = embedding_dim

        self.target_embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=False)
        self.context_embeddings = nn.Embedding.from_pretrained(pretrained_embeddings, freeze=False)

    def forward(self, target_ids):
        target_embeds = self.target_embeddings(target_ids)  # [batch_size, embedding_dim]
        scores = torch.matmul(target_embeds, self.context_embeddings.weight.T)  # [batch_size, vocab_size]
        return scores
```

Figure 2: Word2Vec Architecture

Our hardware struggled to train even with a corpus of 20,000 embeddings (over 30 minutes), let alone the 2M original words. To address this, we kept the number of words trained very low (20,000), and employed optimization techniques such as early stopping for epoch number optimization and learning-rate scheduling. Despite these efforts, we had to stop the training earlier than desired. Although the losses continued to decrease, they did so at an unsatisfactory rate.



```
avg_loss = epoch_loss / len(dataloader)
if (epoch + 1) % 25 == 0 or epoch == 0:
    print(f"Epoch [{epoch+1}/{epochs}], Loss: {avg_loss:.4f}")

scheduler.step(avg_loss)
if avg_loss < best_loss:
    best_loss = avg_loss
    patience_counter = 0
else:
    patience_counter += 1
if patience_counter >= patience:
    print(f"Early stopping triggered at epoch {epoch+1}.")
    break
if epoch >= 50 and (best_loss - avg_loss) < 0.4:
    print(f"Early stopping triggered at epoch {epoch+1}.")
    break
```

Figure 3: Early-stopping and lr scheduling

2.4 Evaluation Techniques

- **Cosine Similarity:**

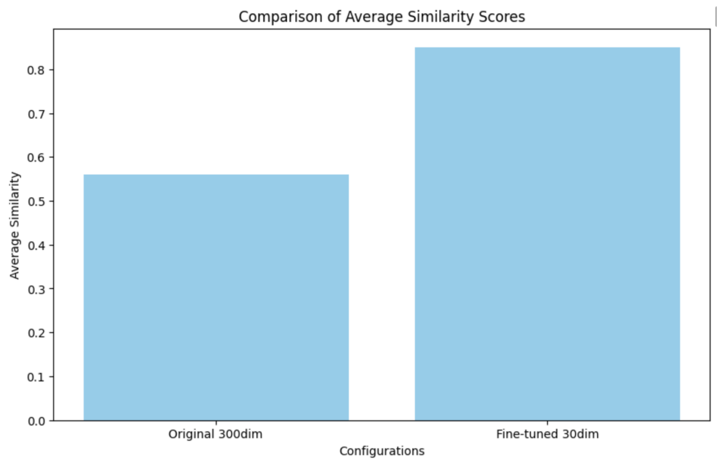


Figure 4: Cosine-similarity plotting

This method demonstrates that our Word2Vec architecture performed adequately for strinctly semantic evaluation.

- **PCA Visualization:**

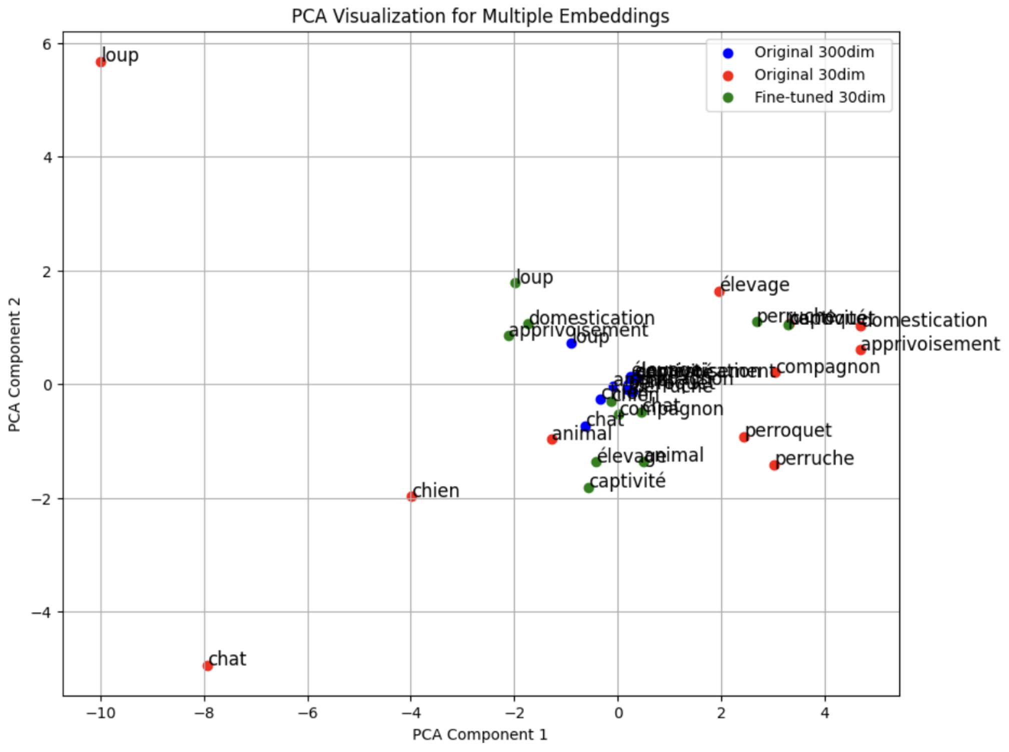


Figure 5: PCA Visualization

We observe here, for a single cluster (domestic animals related) that the 300-dimensions embeddings tend to understand better the similarity between related words.

However, we also see in green that our model still does a really great work. Indeed, even though the main cluster is not as packed as the blue one, our final embeddings remarkably create very relevent sub-clusters : `domestication, apprivoisement` are put together, same for `élevage, captivité` and finally `perruche, perroquet` are also really close.

Finally, we see that our model corrects in a very efficient way the originally reduced 30-dim Embeddings which are, as we see, completely wrong.

The reduced embeddings do not yet achieve the complexity of the original 300-dimensional embeddings, probably because of lots of unperfect parameters such as the reduced number of words for fine-tuning, and the lack of proper phrases separation in the Skip-gram pairs construction.

3 Results

3.1 Reduction Quality

The AutoEncoder achieved a promising reduction for a 10x decrease in dimensions. However, the fine-tuned 30-dimensional embeddings still lagged behind the original 300-dimensional embeddings in capturing complex relationships.

3.2 Observations

- Smaller reduction factors (e.g., x5 or x3) might outperform the original FastText model if better fine-tuning is applied. - Early stopping was due to computational constraints, not overfitting concerns. - PCA visualization showed that 300-dimensional embeddings formed better-defined clusters, capturing subtle relationships.

4 Future Work

To improve results, future efforts should: - Experiment with smaller reduction factors (x5 or x3). - Increase training data size and computational resources to avoid premature stopping. - Extend evaluation to include real-world benchmarks.

5 Conclusion

Despite the limitations, the project demonstrates that significant dimensionality reduction is feasible while retaining semantic information. With further improvements, reduced embeddings could offer a computationally efficient alternative for NLP applications.