Embedding Models from Architecture to Implementation

Let us see the different type of embedding models used in the creation of vector embeddings first lets import the most used packages for this purpose:

“import numpy as np

import matplotlib.pyplot as plt

from sklearn.decomposition import PCA

import torch

from transformers import BertTokenizer, BertModel

from sklearn.metrics.pairwise import cosine_similarity”

Now let us see the embedding models that are to be practically implemented like,

GloVe word embeddings:

“import gensim.downloader as api

word_vectors = api.load('glove-wiki-gigaword-100')

#word_vectors = api.load('word2vec-google-news-300')

word_vectors['king'].shape

word_vectors['king'][:20]

 # Words to visualize

words = ["king", "princess", "monarch", "throne", "crown", "mountain", "ocean", "tv", "rainbow", "cloud", "queen"]

 # Get word vectors

vectors = np.array([word_vectors[word] for word in words])

 # Reduce dimensions using PCA

pca = PCA(n_components=2)

vectors_pca = pca.fit_transform(vectors)

 # Plotting

fig, axes = plt.subplots(1, 1, figsize=(5, 5)) *axes.scatter(vectors_pca[:, 0], vectors_pca[:, 1])*

for i, word in enumerate(words):

axes.annotate(word, (vectors_pca[i, 0]+.02, vectors_pca[i, 1]+.02))

axes.set_title('PCA of Word Embeddings')

plt.show()"

This code will create the embeddings based on GloVe embedding model and now lets see how Word2Vec works:

"result = word_vectors.most_similar(positive=['king', 'woman'],
negative=['man'], topn=1)

 # Output the result

print(f"""   The word closest to 'king' - 'man' + 'woman' is: '{result[0][0]}'   with a similarity score of {result[0][1]}""")"

Now that we have seen two different types of embedding models let us compare Bert and GloVe made embeddings and see the results and find out the more specific purpose used model:

"tokenizer = BertTokenizer.from_pretrained('./models/bert-base-uncased')

model = BertModel.from_pretrained('./models/bert-base-uncased')

 # Function to get BERT embeddings

def get_bert_embeddings(sentence, word):

inputs = tokenizer(sentence, return_tensors='pt')

*outputs = model(**inputs)*

last_hidden_states = outputs.last_hidden_state

word_tokens = tokenizer.tokenize(sentence)

word_index = word_tokens.index(word)

word_embedding = last_hidden_states[0, word_index + 1, :] # +1 to account for [CLS] token
return word_embedding

sentence1 = "The bat flew out of the cave at night."

sentence2 = "He swung the bat and hit a home run."

word = "bat"

bert_embedding1 = get_bert_embeddings(sentence1, word).detach().numpy()

bert_embedding2 = get_bert_embeddings(sentence2, word).detach().numpy()

word_embedding = word_vectors[word]

print("BERT Embedding for 'bat' in sentence 1:", bert_embedding1[:5])

print("BERT Embedding for 'bat' in sentence 2:", bert_embedding2[:5])

print("GloVe Embedding for 'bat':", word_embedding[:5])

bert_similarity = cosine_similarity([bert_embedding1], [bert_embedding2])[0][0]

word_embedding_similarity = cosine_similarity([word_embedding], [word_embedding])[0][0]

print()

print(f"Cosine Similarity between BERT embeddings in different contexts: {bert_similarity}")

print(f"Cosine Similarity between GloVe embeddings: {word_embedding_similarity}")"

Now let us check for similarity in the embeddings by using:

"def cosine_similarity_matrix(features):

norms = np.linalg.norm(features, axis=1, keepdims=True)

normalized_features = features / norms

similarity_matrix = np.inner(normalized_features, normalized_features)
rounded_similarity_matrix = np.round(similarity_matrix, 4)

return rounded_similarity_matrix"

"answers = ["What is the tallest mountain in the world?", "The tallest mountain in the world is Mount Everest.", "Mount Shasta", "I like my hike in the mountains", "I am going to a yoga class"]

question = 'What is the tallest mountain in the world?'

model = SentenceTransformer("sentence-transformers/all-MiniLM-L6-v2")

question_embedding = list(model.encode(question))

sim = []

for answer in answers:

answer_embedding = list(model.encode(answer))
sim.append(cosine_similarity_matrix(np.stack([question_embedding, answer_embedding]))[0,1])

print(sim)best_inx = np.argmax(sim)

print(f"Question = {question}")

print(f"Best answer = {answers[best_inx]}")"

A Dual Encoder is mostly preferred for the purpose of creation of embeddings and seeing as to how useful the embeddings created are for the purpose/task. Now let us use Dual encoder interface onto the previously used examples of questions and answers:

```
"answer_tokenizer = AutoTokenizer \                .from_pretrained("./models/facebook/dpr-ctx_encoder-multiset-base")

answer_encoder = DPRContextEncoder \                .from_pretrained("./models/facebook/dpr-ctx_encoder-multiset-base")

question_tokenizer = AutoTokenizer \                .from_pretrained("./models/facebook/dpr-question_encoder-multiset-base")

question_encoder = DPRQuestionEncoder
\                .from_pretrained("./models/facebook/dpr-question_encoder-multiset-base")

# Compute the question embeddings

question_tokens = question_tokenizer(question, return_tensors="pt")["input_ids"]

question_embedding = question_encoder(question_tokens).pooler_output.flatten().tolist()

print(question_embedding[:10], len(question_embedding))

sim = []

for answer in answers:

answer_tokens = answer_tokenizer(answer, return_tensors="pt")["input_ids"]
answer_embedding = answer_encoder(answer_tokens).pooler_output.flatten().tolist()
sim.append(cosine_similarity_matrix(np.stack([question_embedding,
answer_embedding]))[0,1])

print(sim)best_inx = np.argmax(sim)

print(f"Question = {question}")

print(f"Best answer = {answers[best_inx]}")"
```

Thus implementing this will get you the best results in regards of the embeddings to be compared.