nlp_1

January 18, 2025

1 Task 1: Preparation and Training

[ ]:
```
!python skipgram.py
```

```
[nltk_data] Downloading package brown to /home/jupyter-
[nltk_data]     st125457/nltk_data…
[nltk_data]   Package brown is already up-to-date!
'Len of sentences in news categories: 4623'
voc_size : 13113
Epoch   1000 | Loss: 8.710139
Epoch   2000 | Loss: 9.858974
Epoch   3000 | Loss: 9.525756
Epoch   4000 | Loss: 8.917377
Epoch   5000 | Loss: 9.848055
Training complete in 25m 26s
```

[ ]:
```
!python negative_sampling.py
```

```
[nltk_data] Downloading package brown to /home/jupyter-
[nltk_data]     st125457/nltk_data…
[nltk_data]   Package brown is already up-to-date!
'Len of sentences in news categories: 4623'
100554
voc_size : 13113
Epoch   1000 | Loss: 4.898226
Epoch   2000 | Loss: 4.193873
Epoch   3000 | Loss: 4.071335
Epoch   4000 | Loss: 4.201063
Epoch   5000 | Loss: 4.503699
Training complete in 43m 53s
```

[ ]:
```
!python glove.py
```

```
[nltk_data] Downloading package brown to /home/jupyter-
[nltk_data]     st125457/nltk_data…
[nltk_data]   Package brown is already up-to-date!
'Len of sentences in news categories: 4623'
Vocabulary size: 13113
Epoch   1000 | Loss: 5.752577
```

```
Epoch   2000 | Loss: 24.504128
Epoch   3000 | Loss: 2.692426
Epoch   4000 | Loss: 0.891522
Epoch   5000 | Loss: 0.480459
Training complete in 5m 49s
```

2 Task 2: Model Comparison and Analysis

Compare Skip-gram, Skip-gram negative sampling, GloVe models on training loss, training time

[ ]:
```python
import nltk
# nltk.download('brown')
from nltk.corpus import brown

corpus_token = brown.sents(categories="news")
corpus = [[word.lower() for word in sent] for sent in corpus_token]

flatten = lambda l: [word for sent in l for word in sent]
vocab = list(set(flatten(corpus)))

word2index = {k:v for k, v in enumerate(vocab)}

vocab.append("<UNK>")
word2index["<UNK>"] = len(vocab) - 1

voc_size = len(vocab)
```

[ ]:
```python
import os
import torch
from models import Skipgram, NegativeSampling, Glove

embedding_size = 2

skipgram = Skipgram(voc_size, embedding_size)
neg_sample = NegativeSampling(voc_size, embedding_size)
glove = Glove(voc_size, embedding_size)

all_models = [skipgram, glove, neg_sample]

model_dir = 'model_zoo'
for i, model_name in enumerate(os.listdir(model_dir)):
    if '.pth' in model_name:
        print(model_name)
        model_path = os.path.join(model_dir, model_name)
        state_dict = torch.load(model_path)
        all_models[i].load_state_dict(state_dict)
```

```
all_models
```

```
skipgram.pth

/tmp/ipykernel_940995/931619922.py:18: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  state_dict = torch.load(model_path)

glove.pth
negative_sampling.pth
```

[ ]:
```
[Skipgram(
   (embedding_center): Embedding(13113, 2)
   (embedding_outside): Embedding(13113, 2)
 ),
 Glove(
   (embedding_v): Embedding(13113, 2)
   (embedding_u): Embedding(13113, 2)
   (v_bias): Embedding(13113, 1)
   (u_bias): Embedding(13113, 1)
 ),
 NegativeSampling(
   (embedding_u): Embedding(13113, 2)
   (embedding_v): Embedding(13113, 2)
   (logsigmoid): LogSigmoid()
 )]
```

[ ]:
```python
import numpy as np

with open("word-test.v1.txt", 'r') as f:
    text = f.readlines()

text
# semantic
semantic = text[1:8368]

# syntactic
syntactic = text[15794:17354]
```

```python
def process_data(data):
    corpus = []
    for line in data:
        if line.startswith(':'):
            continue
        corpus.append([w.lower() for w in line.strip().split()])
    return corpus

semantic_data = process_data(semantic)
syntactic_data = process_data(syntactic)

print(semantic_data[:5], syntactic_data[:5])

print(len(semantic_data), len(syntactic_data))

combined_data = semantic_data + syntactic_data
```

```
[['athens', 'greece', 'baghdad', 'iraq'], ['athens', 'greece', 'bangkok',
'thailand'], ['athens', 'greece', 'beijing', 'china'], ['athens', 'greece',
'berlin', 'germany'], ['athens', 'greece', 'bern', 'switzerland']] [['dancing',
'danced', 'decreasing', 'decreased'], ['dancing', 'danced', 'describing',
'described'], ['dancing', 'danced', 'enhancing', 'enhanced'], ['dancing',
'danced', 'falling', 'fell'], ['dancing', 'danced', 'feeding', 'fed']]
8363 1559
```

[ ]:
```python
flatten = lambda l: [w for sent in l for w in sent]
vocabs = list(set(flatten(combined_data)))

word2index = {k:v for k,v in enumerate(vocabs)}
vocabs.append("<UNK>")
word2index["<UNK>"] = len(vocabs) - 1

voc_size = len(vocabs)
voc_size
```

[ ]: 443

[ ]:
```python
from utils import *
import numpy as np
from numpy.linalg import norm

def get_embed(model, word, word2index):
    index = word2index.get(word, word2index['<UNK>'])

    word = torch.LongTensor([index])

    if hasattr(model, 'embedding_center'):
```

```
        embed = (model.embedding_center(word) + model.embedding_outside(word)) /
 ↪ 2
    else:
        embed = (model.embedding_v(word) + model.embedding_u(word)) / 2

    return np.array(embed[0].detach().numpy())

def search_similarity(model, words, word2index, vocabs):
    accuracy = 0
    nw = len(words)
    model_name = model.__class__.__name__

    vocab_embeddings = {vocab: get_embed(model, vocab, word2index) for vocab in␣
 ↪vocabs}

    for word in words:
        word1, word2, word3, word4 = word
        emb_a = get_embed(model, word1, word2index)
        emb_b = get_embed(model, word2, word2index)
        emb_c = get_embed(model, word3, word2index)

        vector = emb_b - emb_a + emb_c
        best_pred = None
        best_similarity = -1

        for vocab, vocab_emb in vocab_embeddings.items():
            if vocab not in [word1, word2, word3]:
                current_sim = cos_sim_np(vector, vocab_emb)
                if current_sim > best_similarity:
                    best_similarity = current_sim
                    best_pred = vocab

        accuracy += 1 if best_pred == word4 else 0

    avg_acc = accuracy / nw
    return model_name, avg_acc

for model in all_models:
    result = search_similarity(model, semantic_data, word2index, vocabs)
    print("semantic_data: ", result)

for model in all_models:
    result = search_similarity(model, syntactic_data, word2index, vocabs)
    print("syntactic_data: ", result)

for model in all_models:
    result = search_similarity(model, combined_data, word2index, vocabs)
```

```
    print("Combined_data: ", result)
```

```
semantic_data:  ('Skipgram', 0.0)
semantic_data:  ('Glove', 0.0)
semantic_data:  ('NegativeSampling', 0.0)
syntactic_data:  ('Skipgram', 0.025016035920461834)
syntactic_data:  ('Glove', 0.025016035920461834)
syntactic_data:  ('NegativeSampling', 0.025016035920461834)
Combined_data:  ('Skipgram', 0.003930659141302157)
Combined_data:  ('Glove', 0.003930659141302157)
Combined_data:  ('NegativeSampling', 0.003930659141302157)
```

[ ]:
```
# Gensim
from gensim.test.utils import datapath
from gensim.models import KeyedVectors
from gensim.scripts.glove2word2vec import glove2word2vec

glove_file = 'glove.6B.50d.txt'
model = KeyedVectors.load_word2vec_format(glove_file, binary=False,␣
 ↪no_header=True)

def search_gensim(model, words):
    tot_acc = 0
    nw = len(words)

    for word in words:
        word1, word2, word3, word4 = word
        result = model.most_similar(positive=[word3, word2], negative=[word1])
        tot_acc += 1 if result[0][0] == word4 else 0

    avg_acc = tot_acc / nw
    return avg_acc

results = search_gensim(model, combined_data)
print(f'combined_data: {results}')

semantic_data = search_gensim(model, semantic_data)
print(f'semantic_data: {semantic_data}')

syntactic_data = search_gensim(model, syntactic_data)
print(f'syntactic_data: {syntactic_data}')
```

```
combined_data: 0.45686353557750453
semantic_data: 0.47219897166088726
syntactic_data: 0.37459910198845414
```

Use the similarity dataset4 to find the correlation between your models' dot product and the provided similarity metrics. (from scipy.stats import spearmanr) Assess if your embeddings correlate with human judgment. (1 points)

```
[ ]: with open('wordsim_relatedness_goldstandard.txt', 'r') as f:
         data = f.readlines()

     def process_data(data):
         corpus = []
         for line in data:
             if line.startswith(':'):
                 continue
             corpus.append([w.lower() for w in line.strip().split()])
         return corpus

     gold_data = process_data(data)

     def get_embed(model, word, word2index):
         index = word2index.get(word, word2index['<UNK>'])

         word = torch.LongTensor([index])

         if hasattr(model, 'embedding_center'):
             embed = (model.embedding_center(word) + model.embedding_outside(word)) /
       ↪ 2
         else:
             embed = (model.embedding_v(word) + model.embedding_u(word)) / 2

         return np.array(embed[0].detach().numpy())

     np_gold = np.array(gold_data)
     wordsim = {}

     for idx, model in enumerate(all_models):
         model_name = model.__class__.__name__
         wordsim[model_name] = [
             np.dot(
                 get_embed(model, row[0], word2index),
                 get_embed(model, row[1], word2index)
             )
             for row in np_gold
         ]

     # wordsim

     glove_file = 'glove.6B.50d.txt'
     gen_sim = KeyedVectors.load_word2vec_format(glove_file, binary=False,␣
       ↪no_header=True)

     wordsim_gen = [
         np.dot(
```

```
        gen_sim.word_vec(row[0]),
        gen_sim.word_vec(row[1])
    )
    for row in np_gold
]
```

```
/tmp/ipykernel_940995/177488539.py:46: DeprecationWarning: Call to deprecated
`word_vec` (Use get_vector instead).
  gen_sim.word_vec(row[0]),
/tmp/ipykernel_940995/177488539.py:47: DeprecationWarning: Call to deprecated
`word_vec` (Use get_vector instead).
  gen_sim.word_vec(row[1])
```

[ ]:
```
from scipy.stats import spearmanr

corr_gold_data = [float(data[-1]) for data in gold_data]

for idx, ws in enumerate(wordsim.keys()):
    corr_coef, p_value = spearmanr(corr_gold_data, wordsim[ws])
    print(f"{all_models[idx].__class__.__name__}:")
    print(f"Spearman correlation: {corr_coef}")
    print(f"P-value: {p_value}")

corr_coef, p_value = spearmanr(corr_gold_data, wordsim_gen)
print(f"Gensim: ")
print(f"Spearman correlation: {corr_coef}")
print(f"P-value: {p_value}")

corr_coef, p_value = spearmanr(corr_gold_data, corr_gold_data)
print(f"Y_true: ")
print(f"Spearman correlation: {corr_coef}")
print(f"P-value: {p_value}")
```

```
Skipgram:
Spearman correlation: nan
P-value: nan
Glove:
Spearman correlation: nan
P-value: nan
NegativeSampling:
Spearman correlation: nan
P-value: nan
Gensim:
Spearman correlation: 0.4763288136072529
P-value: 1.119858688913286e-15
Y_true:
Spearman correlation: 1.0
P-value: 0.0
```

```
/tmp/ipykernel_940995/1811239402.py:6: ConstantInputWarning: An input array is
constant; the correlation coefficient is not defined.
  corr_coef, p_value = spearmanr(corr_gold_data, wordsim[ws])
```

Train models

Model	Window Size	Training Loss	Training time	Syntactic Accuracy	Semantic Accuracy
Skipgram	2	9.525756	25m 26s	0	0
Skipgram (Neg)	2	2.071335	43m 53s	0	0
Glove	2	0.480459	58m 23s	0	0
Glove (Gensim)	10	-	-	0.745	0.375

Spearman correlation

Model	Skipgram	NEG	GloVe	GloVe (gensim)	Y_true
Correlation	nan	nan	nan	0.47	1.0