Word2Vec

Any word that occurs in the same context, should have similar vector.

- We can implement word2vec in 2 ways
  1. Continuous Bag-of-words (CBOW)
     
     \[
     \begin{align*}
     \text{Context} & : \{ W_{t-2}, W_{t-1}, W_t, W_{t+1}, W_{t+2} \} \\
     \text{output} & : W_t
     \end{align*}
     \]

  2. Skip-gram → better performance
     
     \[
     \begin{align*}
     \text{input} & : W_t \\
     \text{output} & : \{ W_{t-2}, W_{t-1}, W_t, W_{t+1}, W_{t+2} \}
     \end{align*}
     \]
Implementing Word2Vec

• Load the Data

```python
with open('text.txt', 'r') as f:
    text = f.read()
```

• Preprocess

```python
import re
from collections import Counter

def preprocess(text):
    text = text.lower().replace('.', '<PERIOD>').
              replace(':', '<COLON>').
              ... .replace(...)

    words = text.split()
    c = Counter(words)  # counter for word frequency

    ref_words = [word for word in words if c[word] > 5]
    return ref_words
```

```python
words = preprocess(text)  # create a big list of words
```
Create a Dictionary

def lookup_table(words):
    c = Counter(words)
    vocab = Sorted(c, k = c.get, reverse = True)
    int2vocab = {i : word for i, word in enumerate(vocab)}
    vocab2int = {word : i for i, word in int2vocab.items()}
    return vocab2int, int2vocab

vocab2int, int2vocab = lookup_table(words)
words_idx = [vocab2int[w] for w in words]
• Subsample

we don't want to weigh words that are too frequent, very much. this is basically noise

the, and, a, ...

Mikolov Subsampling

\[ P(W_i) = 1 - \sqrt{\frac{t}{f(W_i)}} \]

some threshold

\( t \leftarrow \) for word \( W_i \)

\( f(W_i) \) frequency of \( W_i \) in dataset

\( P \) is the probability of discarding \( W_i \).

Example: \( P(0) = 1 - \sqrt{\frac{1 \times 10^{-5}}{\sqrt{1 \times 10^6 / 16 \times 10^6}}} = 0.98735 \)

word "the"

we want to discard "the" most of the time; but still keeping enough of it to make an embedding.
from collections import Counter
import random
import numpy as np

t = 1e-5
C = Counter(words_idx)
text_size = len(words)

freqs = {word: count / text_size for word, count in C.items()}
pdrop = {word: 1 - np.sqrt(t / freqs[word]) for word in C}

train_words = [word for word in words_idx if
               random.random() < (1 - pdrop[word])
               between 0 & 2
               probability of
               keeping a word.

discarding too
frequent words
from the text

• Create batches of Data

def get_target(words, idx, wsize = 5):
    R = random.choice(range(1, wsize + 1))
    fm = max(0, idx - R)
    to = min(len(words), idx + R)

    return words[fm:idx] + words[idx + 1: to + 1]
```python
def batch (words, bsize, wsize = 5):
    n = len (words) // bsize
    words = words [: n * bsize ]

    for idx in range (0, len (words), bsize ) :
        x, y = [ ], []
        b = words [ idx : idx + bsize ]
        for i in range (len (b)) :
            bx = b [ i ]
            by = get_target (b, i, wsize)
            x. extend ([ bx ] * len (by))
            y. extend (by) # put each by on a new row
        yield x, y
```

• Define a Similarity Metric

```python
def cos_similarity (embed, vsize=16, vwindow=100,
                    device='cpu') :

    \( \mathbf{a} \rightarrow \mathbf{b} \)
    similarity = \cos \theta

    \| \mathbf{a} \| \rightarrow \mathbf{a} = \mathbf{embed} \cdot \text{weight}
    \| \mathbf{b} \| \rightarrow \mathbf{b} = \mathbf{embed} \cdot \text{weight} \cdot \text{pow} (2) \cdot \text{sum} (\text{dim}=1)
    \sqrt{\| \mathbf{a} \| \cdot \| \mathbf{b} \|}
    \text{unsqueeze} (0)
```
pick n words from (0, window) and (1000, window + 1000)

```python
examples = np.array(random.sample(range(vwindow), vsize // 2))
examples = np.append(examples, random.sample(range(1000, 1000 + vwindow), vsize // 2))
examples = torch.LongTensor(examples).to(device)
```

embedding layer

```python
vvecs = embed(examples)
```

Similarity

```python
Sims = torch.mm(vvecs, embed_vecs.t()) / mags
dotprod
```

return examples, sims
import torch
from torch import nn
import torch.optim as optim

class SkipGram(nn.Module):
    def __init__(self, n_vocab, n_embed):
        super().__init__()

        one row for each entry in vocab
        self.embedding = nn.Embedding(n_vocab, n_embed)

        self.output = nn.Linear(n_embed, n_vocab)
        self.norm = nn.LogSoftmax(dim=1)

        dim as input and output size of vocab length.
        This is because the output is a series of word scores.

    def forward(self, x):
        x = self.embedding(x)
        scores = self.output(x)
        log_ps = self.norm(scores)
Train the Model

device = 'cuda' if torch.cuda.is_available() else 'cpu'
embed_dim = 300

model = SkipGram(len(vocab2int), embed_dim).to(device)
criterion = nn.NLLLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)

step = 0
epoch = 5

for e in range(epoch):
    for input, target in batch(train_words, 512):
        step += 1
        input, target = torch.LongTensor(input),
        torch.LongTensor(target)
        input, target = input.to(device),
        target.to(device)

        log_ps = model(input)
        loss = criterion(log_ps, targets)
        optimizer.zero_grad()
loss. backward ()
optimizer. step ()

# doing some validation
if step % 20 == 0:
    examples, sims = cos_similarity (model. embedding,
                                      device = device)
    close = sims . topk (6)
    examples, close = examples . to ('cpu'),
                      close . to ('cpu')
    for i, j in enumerate (examples):
        so, the closest words to int2vocab[j.item ()]
        would be: close_words

• Visualize

    import matplotlib.pyplot as plt
    from sklearn.manifold import TSNE

    embeds = model. embedding . weight . to ('cpu'). data . numpy ()
    viz = 600
    tsne = TSNE()
    embed_tsne = tsne . fit_transform (embeds [:viz, :])
Negative Sampling:

The implementation above can be very slow. We can use negative sampling to speed things up.

1. do not take softmax over all the words. Instead, we can add an embedding layer for the output.

2. use a different loss function that only cares about true examples and a small subset of noise examples.

\[ -\log \sigma(Uv_0^T v_w^1) \]

Correct target loss

embedding vector for input word.

embedding vector for output target word.

sigmoid

dot product
\[
\sum_i^N \mathbb{E}_{w_i \sim P_n(w)}
\]

Sum over all words \(w_i\) that are drawn from a noise distribution.

Vocabulary of words that are not in the content of input word (i.e., irrelevant words).

Putting things together:

\[
\text{loss} = - \log \frac{\text{prob} \left( \mathbf{u}_w \mathbf{T} \mathbf{v}_{w_i} \right)}{\sum_i^N \mathbb{E}_{w_i \sim P_n(w)} \log \left( -\mathbf{u}_{w_i} \mathbf{T} \mathbf{v}_{w_i} \right)}
\]

Correct target

Noisy target

SkipGram with Negative Sampling:

class SkipGramNeg(nn.Module):
    def __init__(self, n_vocab, n_embed, noise_dist=None):
        super().__init__()

        self.n_vocab = n_vocab
        self.n_embed = n_embed
        self.noise_dist = noise_dist
embed layer
for input
Embedding layer for output
Initialization using uniform dist
self. in-embed = nn.Embedding (n_vocab, n-embed)

self. out-embed = nn.Embedding (n_vocab, n-embed)

self. in-embed. weight. data. uniform_ (-1, 1)
self. out-embed. weight. data. uniform_ (-1, 1)

```
def forward_input (self, input_words):
    return self.in-embed (input_words)

def forward_output (self, output_words):
    return self.out-embed (output_words)

def forward_noise (self, bsize, n-samples):
    if self.noise_dist is None:
        noise_dist = torch.ones (self.n_vocab)
    else:
        noise_dist = self.noise_dist

   noise_words = torch.multinomial (noise_dist,
        bsize + n-samples, replacement=True)
```
device = 'cuda' if model.out_embed.weight.is_cuda else 'cpu'
noise_words = noise_words.to(device)
noisevecs = self.out_embed(noise_words)
    .view(bsize, n_samples, self.n_embed)
return noisevecs

We then need to define custom loss class
class NegativeSamplingLoss(nn.Module):
    def __init__(self):
        super().__init__()

def forward(self, input_vecs, output_vecs, noise_vecs):
    bsize, embed_size = input_vecs.shape
    input_vecs = input_vecs.view(bsize, embed_size, 1)
    output_vecs = output_vecs.view(bsize, 1, embed_size)

    out_loss = torch.bmm(output_vecs, input_vecs)
        .sigmoid().log().squeeze()
    so that no empty dims are left in the output
\[
\text{noise_loss} = \text{torch.bmm (noise\_vecs\_neg(), input\_vecs)} \\
\quad . \text{sigmoid()} \cdot \text{log()} \cdot \text{squeeze()} \cdot \text{sum(1)}
\]

\[
\text{return } -(\text{out\_loss + noise\_loss}) \cdot \text{mean()}
\]

- **There are some changes to the training loop**

```python
word_freqs = np.array(sorted(freqs.values(), reverse=True))
unigram_dist = word_freqs / word_freqs.sum()
noise_dist = torch.from_numpy(unigram_dist ** 0.75 / 
    np.sum(unigram_dist ** 0.75))

embed_dim = 300
model = SkipGramNeg(len(vocab2int), embed_dim, 
    noise_dist=noise_dist).to(device)
criterion = NegativeSamplingLoss
optimizer = optim.Adam(model.parameters(), lr=0.003)
```
the changes in the inner training loop:
Instead of just calling `model()`, we call the forward functions.

```python
input_vecs = model.forward(input)
output_vecs = model.forward(output)
noise_vecs = model.forward(noise)
loss = criterion(input_vecs, output_vecs, noise_vecs)
```

rest of the code is similar to before.