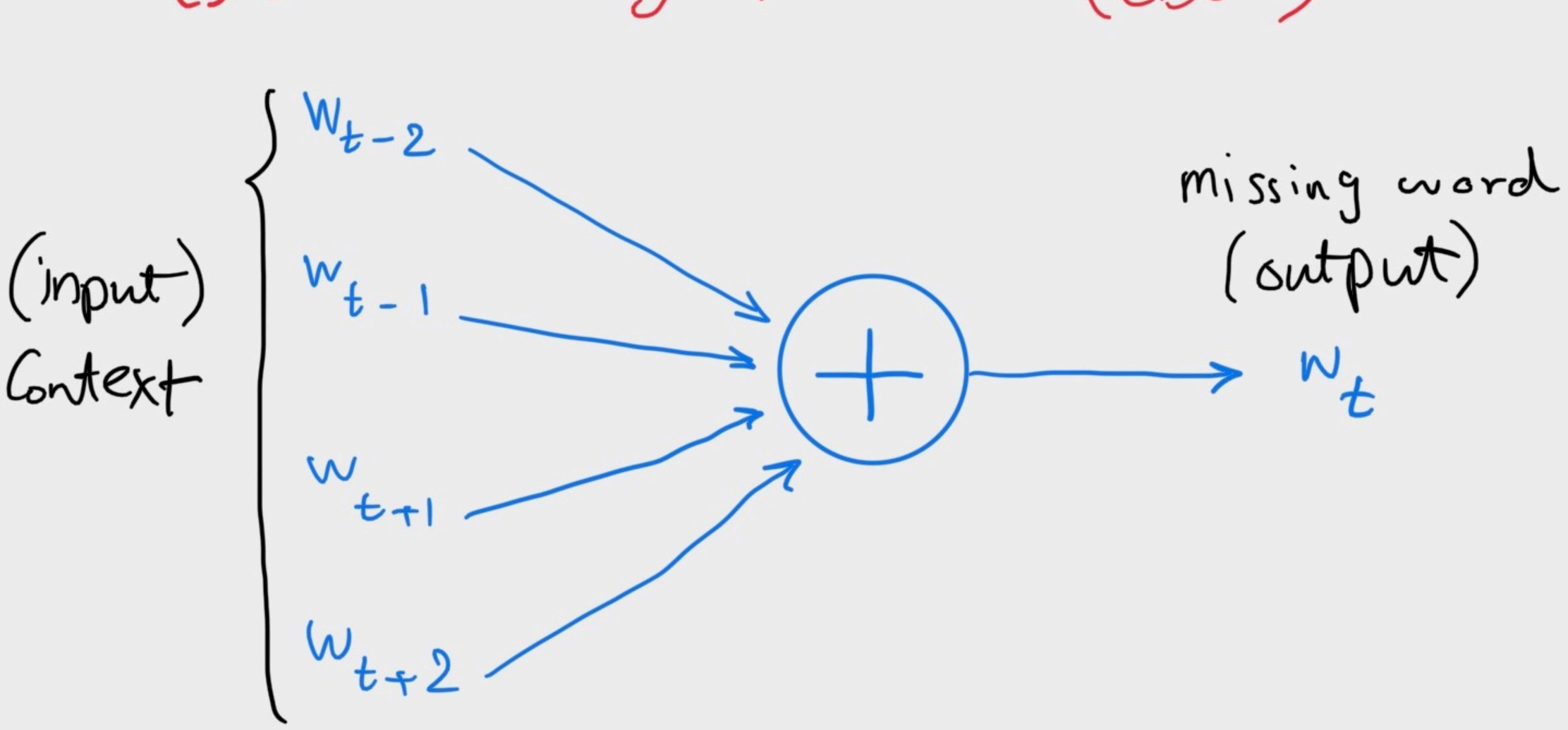
Word2Vec

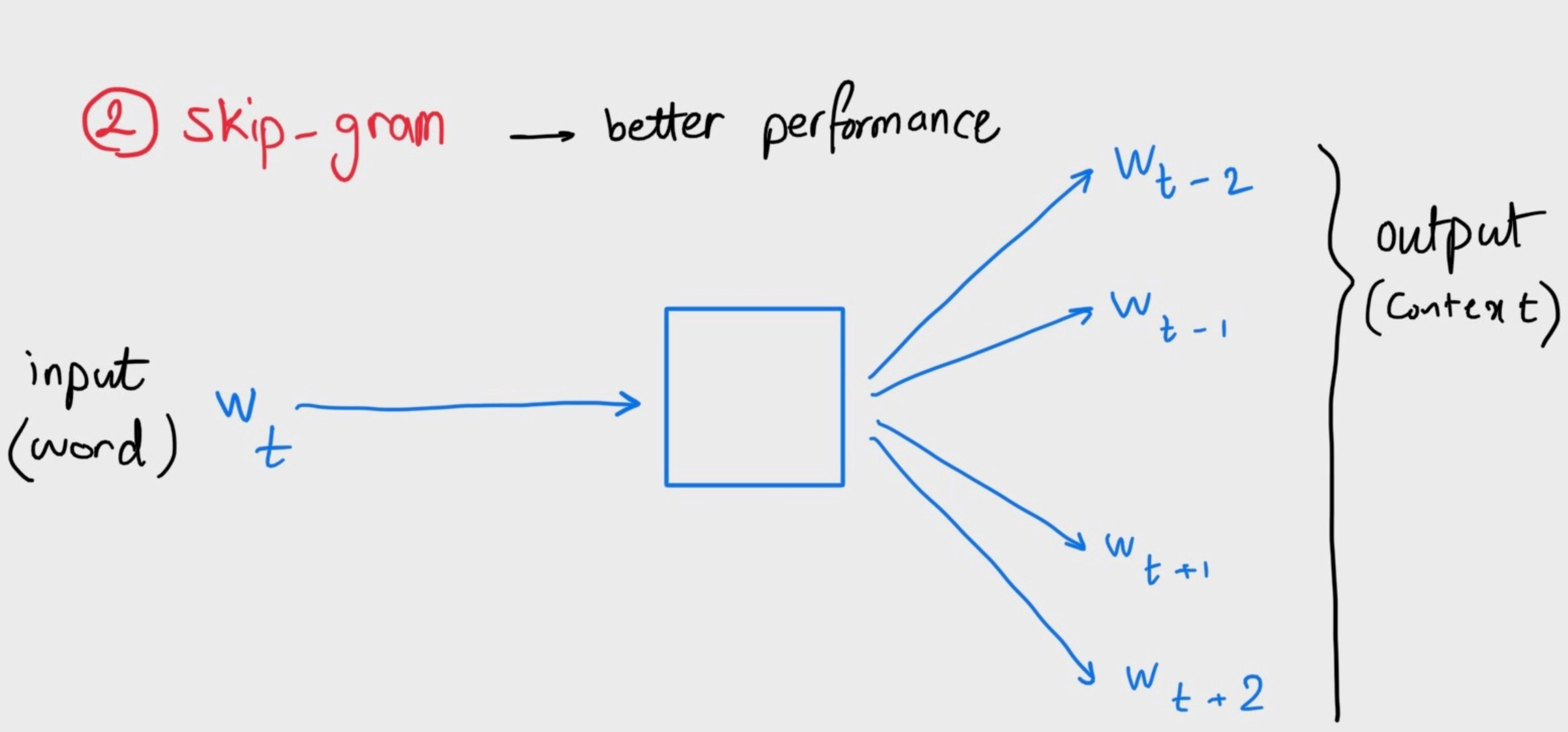
Any word that occurrs in the same context, (↓ words before and after) should have similar vector.

- We can implement word2vec in 2 ways

(1) Continuous Bag-of-words (CBOW)



(2) skip-gram → better performance



Implementing Word2Vec

- Load the Data

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

- Preprocess

```
import re
from collections import Counter
def preprocess(text):
    text = text.lower().replace('.', ' <PERIOD> ')
                .replace(':','<COLON>').... .replace(....)
    words = text.split()
    c = Counter(words)
    ref_words = [word for word in words if c[word]>5]
    return ref_words
```

note the space

counter for word frequency

```
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
```

Sort by Frequency

```
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

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

t → some threshold for word w_i

$f(w_i)$ → frequency of w_i in dataset

P is the probability of discarding w_i.

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

$P(0)$ ↓ 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])
```

↓ discarding too frequent words from the text

random.random() → between 0 & 1

(1 − pdrop[word]) → probability of keeping a word.

- Create batches of Data

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

    return words[frm:idx] + words[idx+1:
                                  to+1]
```

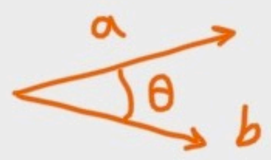
select a wsize, randomly select a number R from [1:wsize], select R token in future & past from words[idx]

```
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

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



v_w → `embed_vecs = embed.weight`

$|\vec{v}|$ → `mags = embed_vecs.pow(2).sum(dim=1)`
`.sqrt().unsqueeze(0)`

similarity = $\cos\theta = \frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}$

pick N words from (0, window) and (1000, widow + 1000)

```
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

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

Similarity →

```
sims = torch.mm(vvecs, embed_vecs.t()) / mags
```

dotprod

```
return examples, sims
```

- Skip Gram

```
torch.nn.Embedding(num_embeddings,   → aka vocab size → # of rows in the lookup matrix
                   embedding_dim,    → # of columns in the lookup matrix
                   ...)
```

```
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)   # this FC layer takes the embedding dim as input and output size of vocab length.
        self.norm = nn.LogSoftmax(dim=1)
```

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)
```

```
        return log_ps
```

- 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):
        close_words = [int2vocab[idx.item()]
                       for idx in close[i][1:]
```

So, the closest words to int2vocab[j.item()] would be: close_words

- Visualize

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

TSNE → t-distributed stochastic neighbor embeddings

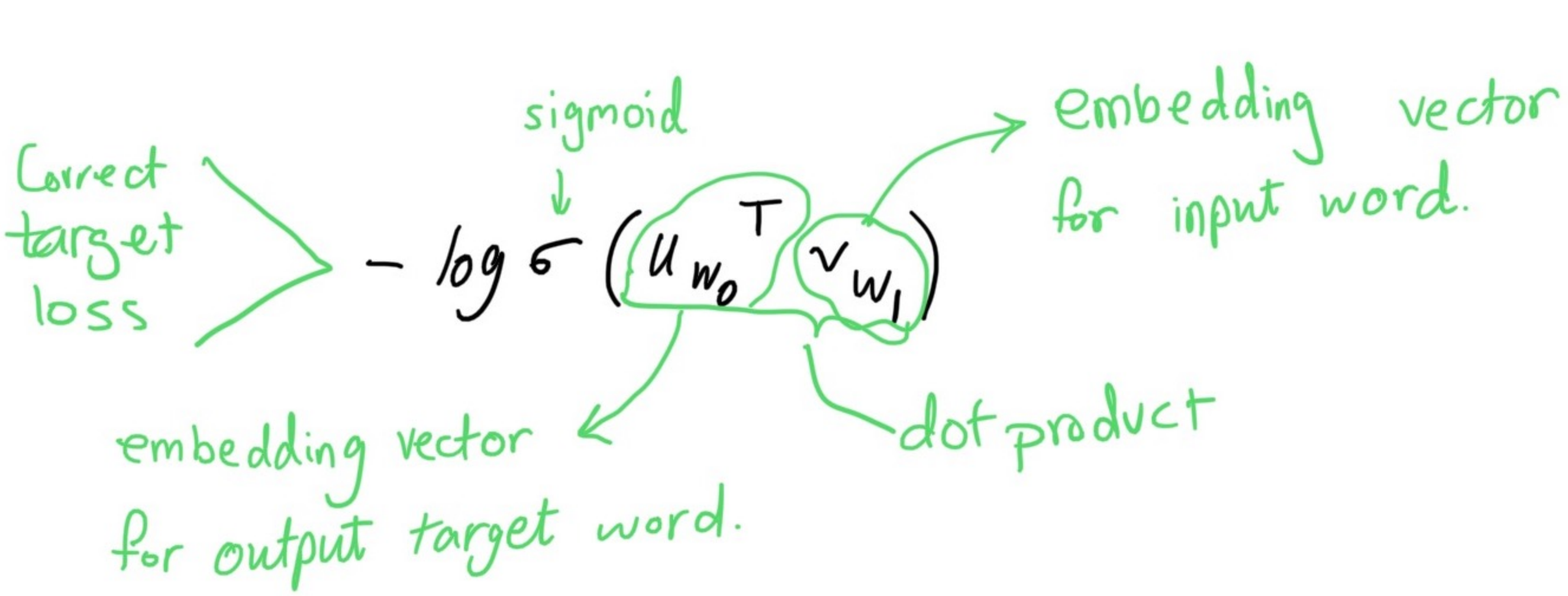
```
embeds= model.embedding.weight.to('cpu').data.numpy()
viz = 600
tsne = TSNE()
embed_tsne = tsne.fit_transform(embeds[:viz,:])
```

- 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.



$$\sum_{i}^{N} \mathbb{E}_{w_i \sim P_n(w)}$$

sum over all words w_i that are drawn from a noise distribution

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

Putting thing together:

$$\text{loss} = -\log \underbrace{\sigma(u_{w_O}^{T} v_{w_I})}_{\text{Correct target}} - \sum_{i}^{N} \mathbb{E}_{w_i \sim P_n(w)} \underbrace{\log \sigma(-u_{w_i}^{T} v_{w_I})}_{\text{Noisy target}}$$

prob (correct target)

prob (noisy target)

Skip Gram 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 → `self.in_embed = nn.Embedding(n_vocab, n_embed)`

embedding layer for output → `self.out_embed = nn.Embedding(n_vocab, n_embed)`

Initialization using uniform dist:

```
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)
```

Set noise dist

Sampling words

```
device = 'cuda' if model.out_embed.weight.is_cuda
                else 'cpu'
noise_words = noise_words.to(device)
noise_vecs = self.out_embed(noise_words)
                .view(bsize, n_samples,
                      self.n_embed)
return noise_vecs
```

- 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
```

a batch of Column vectors →

```
        input_vecs = input_vecs.view(bsize, embed_size, 1)
```

a batch of row vectors →

```
        output_vecs = output_vecs.view(bsize, 1, embed_size)
```

First Term of Loss Function

```
        out_loss = torch.bmm(output_vecs, input_vecs)
                    .sigmoid().log().squeeze()
```

batch mm

so that no empty dims are left in the output

Second Term of Loss Function

```
noise_loss = torch.bmm(noise_vecs.neg(), input_vecs)
                .sigmoid().log().squeeze().sum(1)
```

```
return -(out_loss + noise_loss).mean()
```

- There are some changes to the training loop

```
word_freqs = np.array(sorted(freqs.values(), reverse=True))
unigram_dist = word_freqs / word_freqs.sum()
noise_dist = torch.from_numpy(unigram_dist**.75 /
                              np.sum(unigram_dist**.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.

```
input_vecs = model.forward_input(input)
output_vecs = model.forward_output(target)
noise_vecs = model.forward_noise(inputs.shape[0],5)

loss = criterion(input_vecs, output_vecs, noise_vecs)
```

rest of the code is similar to before.