In [1]:
```
import zipfile
import collections
import random
import math
import torch
import numpy as np
```

In [20]:
```
f = zipfile.ZipFile('data/ptb.zip', 'r')
raw_text = f.read('ptb/ptb.train.txt').decode("utf-8")
sentences = [line.split() for line in raw_text.split('\n')]
tokens = [tk for line in sentences for tk in line]
counter = collections.Counter(tokens)
uniq_tokens = [token for token, freq in list(counter.items()) if counter
idx_to_token, token_to_idx = [], dict()
```

In [21]:
```
sentences = [line.split() for line in raw_text.split('\n')]
tokens = [tk for line in sentences for tk in line]
counter = collections.Counter(tokens)
uniq_tokens = [token for token, freq in list(counter.items()) if counter
idx_to_token, token_to_idx = [], dict()
```

In [22]:
```
for token in uniq_tokens:
    idx_to_token.append(token)
    token_to_idx[token] = len(idx_to_token) - 1
s = [[idx_to_token[token_to_idx.get(tk, 0)] for tk in line] for line in
tokens = [tk for line in s for tk in line]
counter = collections.Counter(tokens)
num_tokens = sum(counter.values())
subsampled = [[tk for tk in line if random.uniform(0, 1) < math.sqrt(1e-
corpus = [[token_to_idx.get(tk) for tk in line] for line in subsampled]
tokens = [tk for line in corpus for tk in line]
counter = collections.Counter(tokens)
sampling_weights = [counter[i]**0.75 for i in range(len(counter))]
population = list(range(len(sampling_weights)))
candidates = random.choices(population, sampling_weights, k=10000)
max_window_size = 5
K = 5
j = 0
data = []
maxLen = 0
for line in corpus:
    if len(line) < 2:
        continue
    for i in range(len(line)):
        window_size = random.randint(1, max_window_size)
        indices = list(range(max(0, i - window_size), min(len(line), i +
        indices.remove(i)
        for idx in indices:
            context = [line[idx] for idx in indices]
        neg = []
        while len(neg) < len(context) * K:
            ne = candidates[j]
            j += 1
            if j >= 10000:
                j = 0
            if ne not in context:
                neg.append(ne)
        data.append([line[i], context, neg])

max_len = max(len(c) + len(n) for _, c, n in data)
centers, contexts_negatives, labels = [], [], []
for center, context, negative in data:
    cur_len = len(context) + len(negative)
    centers += [center]
    contexts_negatives += [context + negative + [0] * (max_len - cur_len
    labels += [[1] * len(context) + [0] * (max_len - len(context))]
```

In [24]:
```
def batchify(data):
    max_len = max(len(c) + len(n) for _, c, n in data)
    centers, contexts_negatives, labels = [], [], []
    for center, context, negative in data:
        cur_len = len(context) + len(negative)
        centers += [center]
        contexts_negatives += [context + negative + [0] * (max_len - cur_
        masks += [[1] * cut_len + [0] * (max_len - cur_len)]
        labels += [[1] * len(context) + [0] * (max_len-len(context))]
    return (np.array(centers).reshape(-1, 1), np.array(context_negatives
            np.array(masks), np.array(labels))
```

In [25]:
```
max_len= max(len(c) + len(n) for _, c, n in data)
centers, contexts_negatives, labels = [], [], []
for center, context, negative in data:
    cur_len= len(context) + len(negative)
    centers += [center]
    contexts_negatives+= [context + negative + [0] * (max_len-cur_len)]
    labels += [[1] * len(context) + [0] * (max_len-len(context))]
```

In [26]:
```
def load_data_ptb(batch_size, max_window_size, num_noise_words):
    sentences = read_ptb()
    vocab = d21.Vocab(sentences, min_freq= 10)
    subsampled = subsampling(sentences, vocab)
    corpus = [vocab[line] for line in subsampled]
    all_centres, all_contexts = get_centres_and_contexts(
            corpus, max_window_size)
    all_negatives = get_negatives(all_contexts, corpus, num_noise_words)
    dataset = gluon.data.ArrayDataset(
            all_centres, all_contexts, all_negatives)
    data_iter = gluon.data.DataLoader(dataset, batch_size, shuffle= true
                                      batchify_fn=batchify)
    return data_iter, vocab
```

In [27]:
```
class PTBdataset(torch.utils.data.Dataset):
    def __init__(self):
        super(PTBdataset).__init__()
        self.centers = np.array(centers).reshape(-1, 1)
        self.contexts_negatives = np.array(contexts_negatives)
        self.labels = np.array(labels)

    def __len__(self):
        return len(self.centers)

    def __getitem__(self, idx):
        return self.centers[idx], self.contexts_negatives[idx], self.lab
```

In [30]:
```
import torch.nn as nn
import torch.optim as optim
pdata = PTBdataset()
data_iter = torch.utils.data.DataLoader(pdata, batch_size=32, shuffle=Tr
vocab_size = len(idx_to_token)
embed_size = 100
import torch.nn as nn
import torch.optim as optim
net = nn.Sequential(
    nn.Embedding(vocab_size, embed_size),
    nn.Embedding(vocab_size, embed_size))
```

In [32]:
```
def skip_gram(center, contexts_and_negatives, embed_v, embed_u):
    v = embed_v(center)
    u = embed_u(contexts_and_negatives)
    pred = npx.batch_dot(v, u.swapaxes(1,2))
    return pred
```

In [35]:
```
loss = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(net.parameters(), 0.01)
m = nn.Sigmoid()
```

In [48]:
```
for epoch in range(5):
    for i, batch in enumerate(data_iter):
        center, context_negative, label = batch
        v = net[0](center.to(torch.int64))
        u = net[1](context_negative.to(torch.int64))
        pred = torch.tensordot(v, torch.transpose(u, 1, 2))
        l = loss(m(pred), label.to(torch.float32))
        optimizer.zero_grad()
        l.backward()
        optimizer.step()
        if (i + 1) % 50 == 0:
            print(epoch, i, float(l))
```

```
0 49 0.9402065277099609
0 99 0.9058647155761719
0 149 0.8934478163719177
0 199 0.9536991119384766
0 249 0.9259445667266846
0 299 0.9144635796546936
0 349 0.9567026495933533
0 399 0.9884507656097412
0 449 0.9946126341819763
0 499 0.9190007448196411
0 549 0.9105140566825867
0 599 0.9254959225654602
0 649 1.076527714729309
0 699 0.9692239165306091
0 749 1.020203948020935
0 799 0.9534463882446289
0 849 1.0168224573135376
0 899 0.9782829880714417
0 949 0.965518593788147
```

In [44]:
```
def get_similar_tokens(query_token, k, embed):
        W = embed.weight
        x = W[token_to_idx[query_token]]
        #cos = np.dot(W,x)/np.sqrt(np.sum(W*W, axis=1) * np.sum(x * x) +
        cos = nn.CosineSimilarity(dim=0, eps=1e-9)
        cos_f = cos(W,x)
        topk = torch.topk(cos_f, k=k+1, dim=0)
        for i in topk.indices:
            print('cosine sim=%.3f: %s' % (cos_f[i],(idx_to_token[i])))
```

In [49]:
```
get_similar_tokens('chip', 3, net[0])
```

```
cosine sim=0.489: n.v.
cosine sim=0.396: an
cosine sim=0.285: a
cosine sim=0.175: 're
```

In [ ]:
```
```

In [ ]:
```
#change this
def get_similar_tokens(query_token, k, embed):
        W = embed.weight.data()
        x = W[vocab[query_token]]
        cos = np.dot(W,x)/np.sqrt(np.sum(W*W, axis1) * np.sum(x * x) + l
        topk = npx.topk(cos, k=k+1, ret_typ='indices').asnumpy().astype(
        for i in topk[1:]:
            print('cosine sim=%.3f: %s' % (cos[i],(vocab.idx_to_token[i]
get_similar_tokens('chip', 3, net[0])
```