In [1]:
```
import nltk
import string
import os
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import re
from random import sample, shuffle
from sklearn.cluster import KMeans, AgglomerativeClustering
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

In [2]:
```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[2]:
```
True
```

In [3]:
```
# I used google collab for training my model, uncomment the lines below to
# connect to google drive in google colab
# NB corpus_root SHOULD BE CHANGED TO MATC THE CORPUS PATH ON THE SPECIFIC MACHIN

# from google.colab import drive
# drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call driv
```

In [4]:
```
stop_words = set(nltk.corpus.stopwords.words("english"))
stemmer = nltk.SnowballStemmer("english", ignore_stopwords = False)
# Folder path where corpus root should be
corpus_root = r"/content/drive/MyDrive/cw2/product_reviews"
file_pattern = r".*"
original_corpus = nltk.corpus.PlaintextCorpusReader(corpus_root, file_pattern)
print(original_corpus.fileids())
```

```
['Canon_PowerShot_SD500.txt', 'Canon_S100.txt', 'Diaper_Champ.txt', 'Hitachi_route
'norton.txt']
```

In [5]:
```
# Core utility function for document cleaning
# Works recursively, split the text into sentences/review, then for each sentence
def process_doc(text, remove_punctuation, case_fold, stem,
                remove_stopwords, remove_short_tokens, tokenize_by, stem_blacklis
                remove_nonalphabetical = False):

  if (tokenize_by == "sentence"):
    sentences = nltk.RegexpTokenizer("##", gaps = True).tokenize(text)
    sentences = [process_doc(sentence, remove_punctuation, case_fold, stem,
                             remove_stopwords, remove_short_tokens, "words", stem
                 for sentence in sentences]
    return sentences
  if (tokenize_by == "reviews"):
    reviews = nltk.RegexpTokenizer("₩[ t ₩]", gaps = True).tokenize(text)
    reviews = [process_doc(review, remove_punctuation, case_fold, stem,
                           remove_stopwords, remove_short_tokens, "words", ste
               for review in reviews]
    return reviews
  if (tokenize_by == "words"):
    words = nltk.TreebankWordTokenizer().tokenize(text)
    if (remove_punctuation):
      words = [w for w in words if w not in string.punctuation and w != "..."
      words = [w.strip("") for w in words]
      words = [w.strip(".") for w in words]
    if (case_fold):
      words = [w.lower() for w in words]
    if (remove_short_tokens):
      words = [w for w in words if len(w) > 2]
    if (stem):
      words = [w if w in stem_blacklist else stemmer.stem(w) for w in words]
    if (remove_stopwords):
      words = [w for w in words if w not in stop_words and w != "n't"]
    if (remove_punctuation):
      words = [w for w in words if w not in string.punctuation and w != "..."
    if (remove_nonalphabetical):
      words = [w for w in words if w.isalpha()]
    return words

def process_corpus(corpus, remove_punctuation:bool, case_fold:bool, stem:bool,
                  remove_stopwords:bool, remove_short_tokens, tokenize_by:str, re
  docs = [word for fileid in corpus.fileids()
            for word in process_doc(corpus.raw(fileid), remove_punctuation, case
                                    stem, remove_stopwords, remove_short_tokens,
                                    tokenize_by, remove_nonalphabetical)
         ]
  return docs

def most_frequent(words, n, should_print):
  freqDist = nltk.FreqDist(words)
  most_common = freqDist.most_common(n)
  if (should_print):
    i = 1
    for (w, count) in most_common:
      print(i , w , count)
      i += 1
  return most_common
```

In [6]:

```
def get_all_sentences_cleaned(corpus_filepath):
  corpus = nltk.corpus.PlaintextCorpusReader(corpus_filepath, file_pattern)
  out = []
  for fileid in corpus.fileids():
    sentences = process_doc(corpus.raw(fileid), True, True, True, True, True,'
    out.extend(sentences)
  return out

# partitions corpus into sentiments, and cleans the text
def get_all_sentiments_cleaned(corpus_filepath, stemming, stop_words, remove_mixe
  corpus = nltk.corpus.PlaintextCorpusReader(corpus_filepath, file_pattern)
  # pattern used to match sentiments
  pattern = re.compile(r"(([a-z -]*₩[[₩-₩+][0-9]₩],? ?)+#[^(₩[)]+)")
  sentiments = []
  for file_id in corpus.fileids():
    text = corpus.raw(file_id)
    text = re.sub("₩[[a-z]+₩]", "", text)
    text = pattern.findall(text)
    for sentiment in text:
      sentiment_parsed = sentiment[0]
      # Find all labels for whether a sentiment is positive or negative
      matches = re.findall("₩[[₩+₩-][0-9]₩]", sentiment_parsed)
      score = 0
      has_positive = False
      has_negative = False
      for match in matches:
        score += int(match[1:-1])
        if match[1] == "+":
          has_positive = True
        if match[1] == "-":
          has_negative = True
      # if the sum of all scores is 0 discard the sample, since we are doing bina
      # classification, optionally remove all sentiments with mixed labels
      if remove_mixed_sentiments and has_positive and has_negative:
        continue
      if (score == 0): continue
      if (score < 0): score = 0
      if (score > 0): score = 1
      sentiment_parsed = process_doc(sentiment_parsed, True, True, stemming, st
      if (len(sentiment_parsed[:-1]) < 2): continue
      sentiments.append((sentiment_parsed[:-1], score))
  return sentiments

def generate_word_to_indx_and_idx_to_word(corpus):
  word_to_idx = {}
  idx_to_word = {}
  i = 0
  for sentence in corpus:
    for word in sentence[0]:
      if (word not in word_to_idx):
        word_to_idx[word] = i
        idx_to_word[i] = word
        i += 1
  return (word_to_idx, idx_to_word)

def get_context_window_tuples(word_to_idx, sentences, window, key_words):
  tuples = []
  for sentence in sentences:
    for i in range(window, len(sentence) - window):
      # if sentence[i] in key_words:
        context = []
        middle_word = word_to_idx[sentence[i]]
        for j in range (i - window, i + window + 1):
          if i != j:
            context.append(word_to_idx[sentence[j]])
        tuples.append((context, word_to_idx[sentence[i]]))


  return tuples


def get_skipgrams(sentiments, window):
  word = []
  context = []
  for sentiment in sentiments:
    sentence = sentiment[0]
    for i in range(len(sentence)):
      cont = [sentence[idx] for idx in range(max(0, i - window), min(len(sentenc
      word.extend([sentence[i]] * (len(cont)))
      context.extend(cont)
  return(word, context)

def get_batches(words, contexts, batch_size):
  shuffled_idxs = sample(range(0, len(words)), len(words))
  batches = []

  batch_word, batch_context = [], []
  for i in range(len(words)):
    idx = shuffled_idxs[i]
    batch_word.append(words[idx])
    batch_context.append(contexts[idx])
    if (i + 1) % batch_size == 0 or i + 1 == len(words):
      batches.append((
        torch.from_numpy(np.array(batch_word)),
        torch.from_numpy(np.array(batch_context))
      ))
      batch_word, batch_context = [], []
  return batches

def get_x_tensors(x_y_tuples):
  tensors = []
  for tuple in x_y_tuples:
    tensors.append(torch.tensor(tuple[0], dtype=torch.long))
  return tensors


def get_y_tensors(tuples, num_classes):
  tensors = []

  for tuple in tuples:
    tensors.append(F.one_hot(torch.tensor(tuple[1]), num_classes=num_classes))
  return tensors

def get_sentiments_as_word_idxs(sentiments, word_to_idx):
  return [([word_to_idx[word] for word in words], label) for (words, label) in
```

In [7]:

```
# Function to split data into K folds
def k_fold_partititoning(sentiments, k, should_shuffle):
  # shuffle
  # we do not shuffle when we need to compare the results of experiments
  if (should_shuffle):
    shuffle(sentiments)
  folds = []
  # determine fold size
  partition_step = len(sentiments) // k
  remainders = len(sentiments) % k
  start = 0
  # append to each fold
  for i in range(k):
    if (remainders > 0):
      folds.append(sentiments[start : start + partition_step + 1])
      start += partition_step + 1
      remainders -= 0
    else:
      folds.append(sentiments[start : start + partition_step])
      start += partition_step
  return folds

# partition the data into two - the i-th fold and the rest
def split_training_testing_from_k_folds(i, folds):
  # To ensure that not tampering is done
  testing = copy.deepcopy(folds[i])

  training = []
  for j in range(i):
    training.extend(folds[j])
  for j in range(i + 1, len(folds)):
    training.extend(folds[j])
  return (training, testing)

def to_tensors(sentiments):
  shuffle(sentiments)
  start = 0
  batches = [(torch.from_numpy(np.array(sentiment[0])), torch.from_numpy(np.arra
  return batches

# y_hat is a tensor output of a sigmoid (y_hat between: [0, 1])
def get_binary_accuracy(y_hat, y, verbose=False):
  # if y_hat <= 0.5: rounded = 0 else: rounded = 1
  rounded = torch.round(y_hat)
  correct = (rounded == y).float()
  if verbose:
    print("y_hat:", y_hat.data)
    print("y:", y.data)
    print("rounded:", rounded.data)
    print("correct: ", correct.data)
  return correct
```

In [8]:

```
class CNN(nn.Module):
    def __init__(self, vocab_size, n_filters, embedding_dim = None, padding_idx

        super().__init__()

        if (embedding_weights != None):
          self.embedding = nn.Embedding.from_pretrained(embedding_weights, freez
          embedding_dim = embedding_weights.size()[1]
        else:
          self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx =
        self.conv_0 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (3, embedding_dim))
        self.conv_1 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (4, embedding_dim))
        self.conv_2 = nn.Conv2d(in_channels = 1,
                                out_channels = n_filters,
                                kernel_size = (5, embedding_dim))

        self.fc = nn.Linear(3 * n_filters, 1)
        self.dropout = nn.Dropout(dropout_rate)
        self.sigmoid = nn.Sigmoid()


    def forward(self, text, training = False):

        embedding = self.embedding(text)
        #embedding = [len(text) x embedding_size]

        embedding = embedding.unsqueeze(1)
        embedding = embedding.unsqueeze(1)
        embedding = embedding.permute(1, 2, 0, 3)

        conved_0 = F.relu(self.conv_0(embedding).squeeze(3))
        conved_1 = F.relu(self.conv_1(embedding).squeeze(3))
        conved_2 = F.relu(self.conv_2(embedding).squeeze(3))
        #conved_n = [len(text) - kernel_size x number of filters]

        pooled_0 = F.max_pool1d(conved_0, conved_0.shape[2]).squeeze(2)
        pooled_1 = F.max_pool1d(conved_1, conved_1.shape[2]).squeeze(2)
        pooled_2 = F.max_pool1d(conved_2, conved_2.shape[2]).squeeze(2)

        concat = torch.cat((
            pooled_0,
            pooled_1,
            pooled_2)
          , dim = 1)
        # Apply dropout only when training
        if (training):
          concat = self.dropout(concat)

        #concat = [len(text) - kernel_size x number of filters]
        return self.sigmoid(self.fc(concat))
```

In [9]:

```
def train_and_eval(num_filters, embedding_size, epochs, batch_size, learning_rate
  K = len(folds)
  # accuracies will contain the accuracies for each fold for each epoch
  accuracies = np.zeros((K, epochs))
  for k in range(K):
    if (verbose):
      print("FOLD: ", k + 1)
    (training_data, testing_data) = split_training_testing_from_k_folds(k, folds)
    model = CNN(vocab_size + 1, num_filters, embedding_size, padding_idx, dropout
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), learning_rate)
    loss_fn = nn.BCELoss()
    loss_fn = loss_fn.to(device)
    for epoch in range(epochs):
      if (verbose):
        print("EPOCH:", epoch + 1)
      acc = 0
      total_loss = 0
      n = 0
      for sample in to_tensors(training_data):
        optimizer.zero_grad()
        (sentiment, label) = sample
        sentiment = sentiment.to(device)
        label = label.to(device)
        y_hat = model(sentiment, training = True).squeeze()
        acc += get_binary_accuracy(y_hat, label, 1 == 0)
        loss = loss_fn(y_hat, label.float())
        total_loss += loss
        loss.backward()
        optimizer.step()
        n += 1
      if (verbose):
        print("TRAINING: accuracy", (acc / n).item(), "total loss", total_loss.i
      with torch.no_grad():
        accuracy = 0
        for (sentiment, label) in to_tensors(testing_data):
          sentiment = sentiment.to(device)
          label = label.to(device)
          accuracy += get_binary_accuracy(model(sentiment), label)
        accuracies[k][epoch] = (accuracy / len(testing_data)).item()
        if (verbose):
          print("EPOCH VALIDATION ACCURACY", accuracy.item())

  # epoch averages contains the average accuracy for each epoch accross all folds
  epoch_averages = np.mean(accuracies, axis=0)
  best_epoch = np.argmax(epoch_averages)
  return "Best epoch:", best_epoch + 1, "with an average", epoch_averages[best_e
```

In [10]:

```
def run_experiment(num_filters, embedding_size, epochs, batch_size, learning_rate
  # sentiments - a list of tuples, tuple[0] is the cleaned text of a sentiment, t
  sentiments = get_all_sentiments_cleaned(corpus_root, stemming, stopword_removal
  (word_to_idx, idx_to_word) = generate_word_to_indx_and_idx_to_word(sentiments)
  # tuple[0] in sentiments becomes a list of ints, each int represents a token, w
  sentiments = get_sentiments_as_word_idxs(sentiments, word_to_idx)
  PADDING_STR = ""
  PADDING_IDX = len(word_to_idx)
  idx_to_word[PADDING_IDX] = PADDING_STR
  word_to_idx[PADDING_STR] = PADDING_IDX
  vocab_size = len(idx_to_word)
  # The filter size of the CNN is 5, all shorter texts than that need padding
  for sentiment in sentiments:
    while (len(sentiment[0]) < 5):
      sentiment[0].append(PADDING_IDX)
  k_folds = k_fold_partititoning(sentiments, 5, should_shuffle)

  # training and evaluation
  return train_and_eval(num_filters, embedding_size, epochs, batch_size, learnin
```

In [11]:

```
print("Removing mixed sentiments", run_experiment(num_filters = 100, embedding_si
                                    learning_rate=0.001, stemming=False, stopw

print("Keeping mixed sentiments", run_experiment(num_filters = 100, embedding_siz
                                    learning_rate=0.001, stemming=False, stopw
```

```
Removing mixed sentiments ('Best epoch:', 5, 'with an average', 0.708489108085632
Keeping mixed sentiments ('Best epoch:', 5, 'with an average', 0.6722772240638732)
```

In [12]:

```
print("With stemming", run_experiment(num_filters = 100, embedding_size = 300, ep
                                    learning_rate=0.001, stemming=True, stopwo

print("With stop words removal", run_experiment(num_filters = 100, embedding_size
                                    learning_rate=0.001, stemming=False, stopw

print("Baseline", run_experiment(num_filters = 100, embedding_size = 300, epochs
                                    learning_rate=0.001, stemming=False, stopw
```

```
With stemming ('Best epoch:', 13, 'with an average', 0.6985148429870606)
With stop words removal ('Best epoch:', 9, 'with an average', 0.6851130485534668)
Baseline ('Best epoch:', 6, 'with an average', 0.7138613820075989)
```

In [13]:

```
print("Dropout 0", run_experiment(num_filters = 100, embedding_size = 300, epochs
                                    learning_rate=0.001, stemming=False, stopw

print("Dropout 0.25", run_experiment(num_filters = 100, embedding_size = 300, epo
                                    learning_rate=0.001, stemming=False, stopw

print("Dropout 0.50", run_experiment(num_filters = 100, embedding_size = 300, epo
                                    learning_rate=0.001, stemming=False, stopw

print("Dropout 0.75", run_experiment(num_filters = 100, embedding_size = 300, epo
                                    learning_rate=0.001, stemming=False, stopw

print("Dropout 0.85", run_experiment(num_filters = 100, embedding_size = 300, epo
                                    learning_rate=0.001, stemming=False, stopw
```

```
Dropout 0 ('Best epoch:', 20, 'with an average', 0.6866336584091186)
Dropout 0.25 ('Best epoch:', 16, 'with an average', 0.6985148549079895)
Dropout 0.50 ('Best epoch:', 9, 'with an average', 0.7054455399513244)
Dropout 0.75 ('Best epoch:', 6, 'with an average', 0.705940580368042)
Dropout 0.85 ('Best epoch:', 6, 'with an average', 0.7113861203193664)
```

In [16]:

```
print("Dropout 0.95", run_experiment(num_filters = 100, embedding_size = 300, epo
                                    learning_rate=0.001, stemming=False, stopw
```

```
Dropout 0.95 ('Best epoch:', 18, 'with an average', 0.6539603888988494)
```

In [14]:

```
# Running for only 15 epochs to speed up the experiment
print("Filter number 50:", run_experiment(num_filters = 50, embedding_size = 300
                                          learning_rate=0.001, stemming=False, stopw

print("Filter number 100", run_experiment(num_filters = 100, embedding_size = 30
                                          learning_rate=0.001, stemming=False, stopw

print("Filter number 200", run_experiment(num_filters = 200, embedding_size = 30
                                          learning_rate=0.001, stemming=False, stopw

print("Filter number 300", run_experiment(num_filters = 300, embedding_size = 30
                                          learning_rate=0.001, stemming=False, stopw

print("Filter number 400", run_experiment(num_filters = 400, embedding_size = 30
                                          learning_rate=0.001, stemming=False, stopw
```

```
Filter number 50: ('Best epoch:', 9, 'with an average', 0.6836633563041687)
Filter number 100 ('Best epoch:', 6, 'with an average', 0.6985148429870606)
Filter number 200 ('Best epoch:', 8, 'with an average', 0.6693069219589234)
Filter number 300 ('Best epoch:', 9, 'with an average', 0.6623762249946594)
Filter number 400 ('Best epoch:', 9, 'with an average', 0.6757425785064697)
```

In [20]:

```
print("Embedding size 50", run_experiment(num_filters = 100, embedding_size = 50
                                          learning_rate=0.0005, stemming=False, stop

print("Embedding size 100", run_experiment(num_filters = 100, embedding_size = 1
                                          learning_rate=0.0005, stemming=False, stop

print("Embedding size 200", run_experiment(num_filters = 100, embedding_size = 2
                                          learning_rate=0.0005, stemming=False, stop

print("Embedding size 300", run_experiment(num_filters = 100, embedding_size = 3
                                          learning_rate=0.0005, stemming=False, stop

print("Embedding size 400", run_experiment(num_filters = 100, embedding_size = 4
                                          learning_rate=0.0005, stemming=False, stop
```

```
Embedding size 50 ('Best epoch:', 11, 'with an average', 0.6975247383117675)
Embedding size 100 ('Best epoch:', 15, 'with an average', 0.6891089081764221)
Embedding size 200 ('Best epoch:', 10, 'with an average', 0.7064356327056884)
Embedding size 300 ('Best epoch:', 13, 'with an average', 0.7143564343452453)
Embedding size 400 ('Best epoch:', 12, 'with an average', 0.7099009871482849)
```

In [19]:

```
print("Accuracy with best parameters:", run_experiment(num_filters = 100, embeddi
                                          learning_rate=0.0005, stemming=False, stop
```

```
Accuracy with best parameters: ('Best epoch:', 28, 'with an average', 0.7410756111
```