Natural Language Processing

Sentiment Analysis for IMDB Movie Reviews

Peijun Xiao

Overview

- Website: https://courses.engr.illinois.edu/ie534/fa2019/NLP.html
- Data set: Large Movie Review Dataset
- Goal:
  - Natural Language Processing (NLP)
  - Use neural networks with sequential data
- Part 1 **Bag of Words** (HW5)
- Part 2 **Recurrent Neural Network** (HW5)
- Part 3 **Language Model** (HW6)

Basic Terminology

- Question: How do we represent the meaning of a word?

- Answer 1: Use a **one-hot vector**
  (one-hot vector is a vector of all zeros except for one element containing a one)

- For example, we want to predict the sentiment of "This movie is bad" and "This movie is good".

  vocab = ['this','movie','is','bad','good', unknown]

  [0,1,0,0,0,0] → this
  [0,0,1,0,0,0] → movie
  [0,0,0,1,0,0] → is
  [0,0,0,0,0,1] → good

Issues 1: vocabulary size might be **large**

Issues 2: no natural notion of **similarity** for one-hot vectors

- For example,

  [0,0,0,0,0,1] --> good
  [0,0,0,0,1,0] --> great

- These two vectors are **orthogonal**, i.e. the dot product of these two vectors are zero.

Solution:

Learn to encode **similarity** in the vectors themselves

expect =

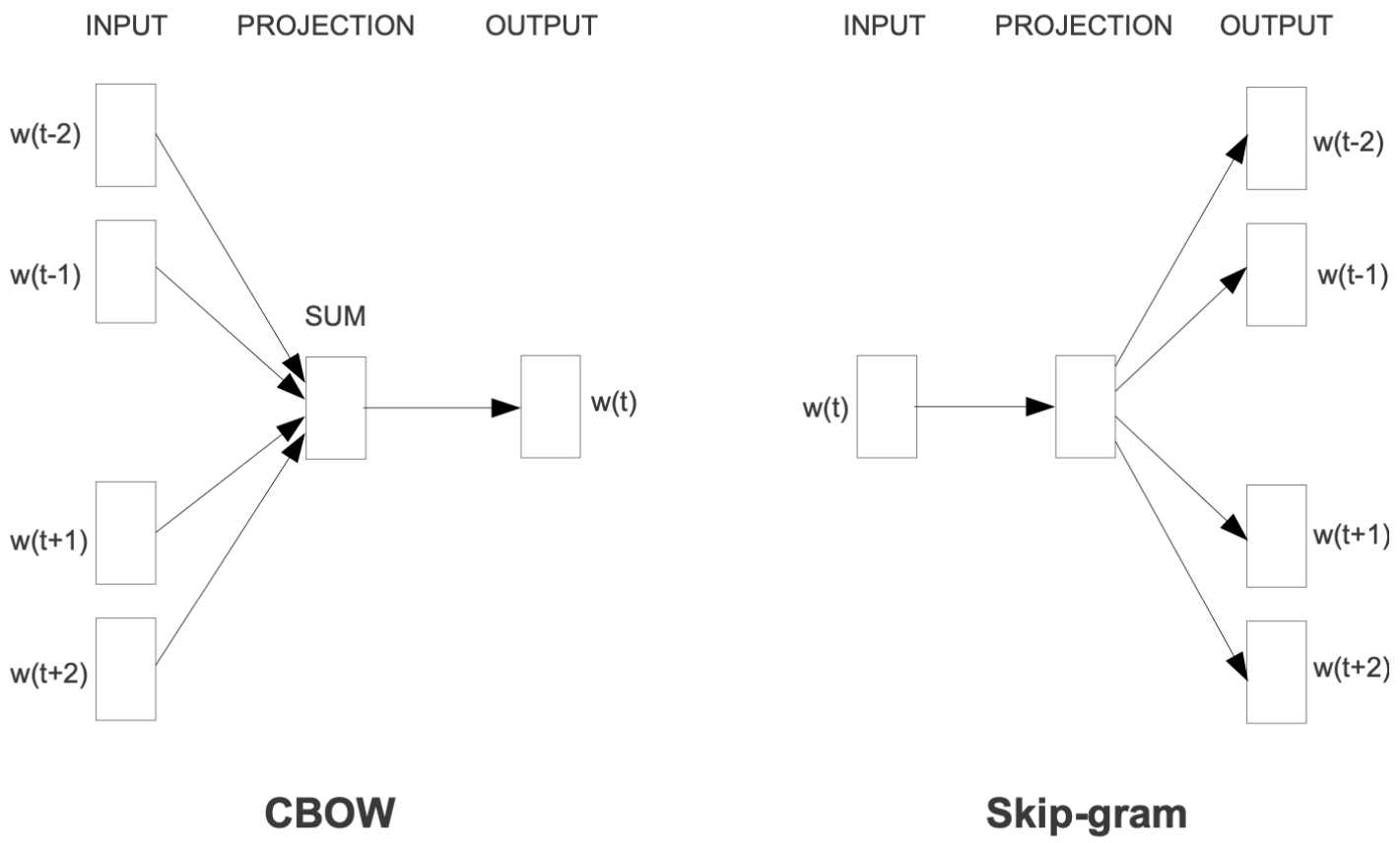
$$\begin{pmatrix} 0.286 \\ 0.792 \\ -0.177 \\ -0.107 \\ 0.109 \\ -0.542 \\ 0.349 \\ 0.271 \\ 0.487 \end{pmatrix}$$

Word embedding (also called word vectors)

- We will build a fixed length vector for each word that has real value (as opposed to just 1 or 0), such that it is **similar** to vectors of words that appear in **similar contexts**
- When a word appears in a text, its **context** is **the set of words that appear nearby** (within a fixed-size window).
- For example, “stand on the _____ of giants”
- **Word2vec** and **GloVe** are examples of word embeddings.

Word2vec (Mikolov et al. 2013)

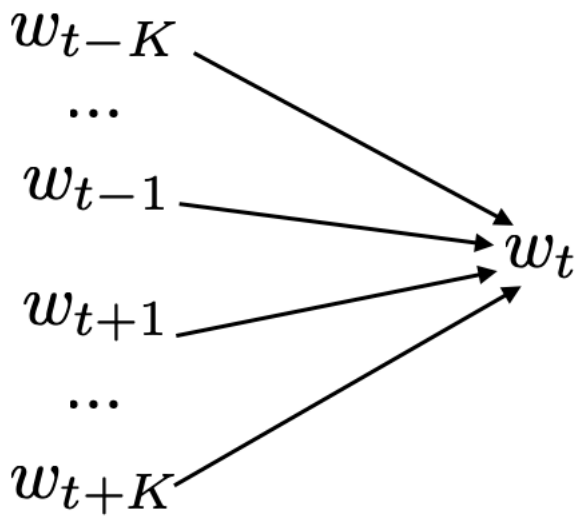
- Prediction between every word and its context



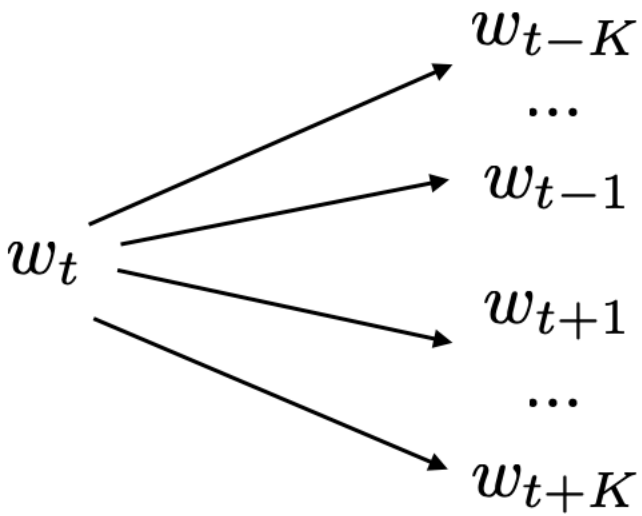
CBOW (continuous bags-of-words)

Predict the target word from bag-of-words context.

$$\max_{u,v} \prod_{t=1}^{T} p\left(w_t | w_{t-K}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+K}\right)$$



Skip-Gram



Predict context words from the a target word.

$$\max_{u,v} \prod_{t=1}^{T} p\left(w_{t-K}, \ldots, w_{t-1}, w_{t+1}, \ldots, w_{t+K} \middle| w_t\right)$$

GloVe (Pennington et al., EMNLP 2014, stands for “Global Vector”)

- Word2Ve is a prediction based method. GloVe is a **count** based method.
- We count the numbers of times a word j occurs in the context of word i, denoted the number as X_{ij}
- From all the X_{ij}, we derive the co-occurrence probabilities P_{ij}

  where $\mathrm{P_{ij}} = \mathrm{P(j|i)} = \frac{\mathrm{X_{ij}}}{\sum_{\mathrm{k}} \mathrm{X_{ik}}}$

GloVe (continue)

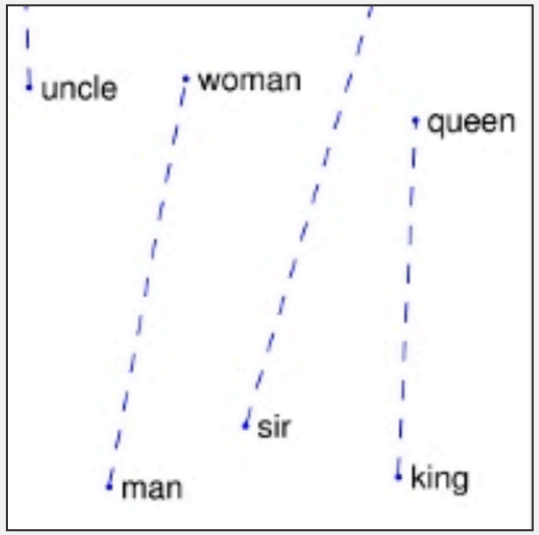
- Idea: ratios of **co-occurrence probabilities** can encode **meaning components**
- For example, consider a document that includes “ice”, “solid”, “gas”, “waster”, “steam”, and “fashion”.

	x = solid	x = gas	x = water	x = fashion
$P(x\|\text{ice})$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(x\|\text{steam})$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$\frac{P(x\|\text{ice})}{P(x\|\text{steam})}$	8.9	8.5×10^{-2}	1.36	0.96

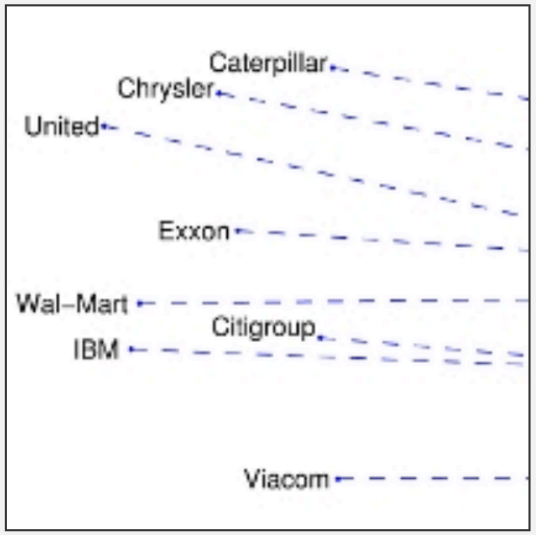
Link of Table

Encoding linear relationship

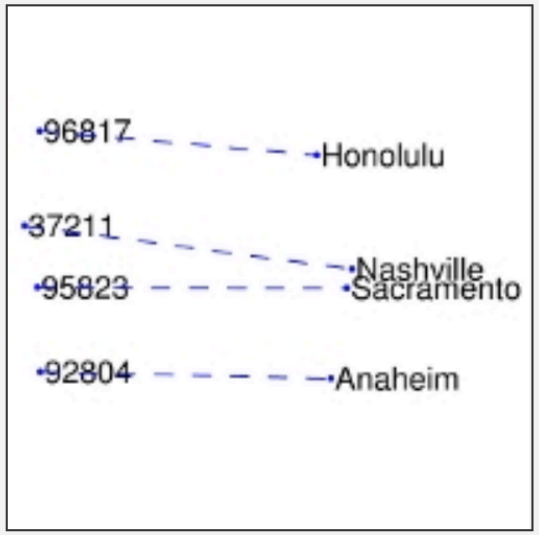
- Q: How can we capture ratios of co-occurrence probabilities to learn linguistic patterns as **linear relationships** between the word vectors?



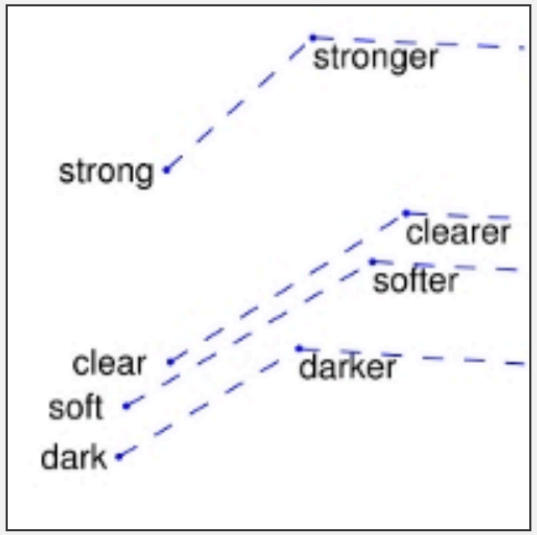
man - woman



company - ceo



city - zip code



comparative - superlative

[Link of Figure: linear relationships]

Log-linear Model in GloVe

Idea: Use log-linear model: $w_i * w_j = \log P(i|j)$ with **vector differences** $w_x * (w_a - w_b) = \log \frac{P(x|a)}{P(x|b)}$

$$J = \sum_{i,j=1}^{V} f\left(X_{ij}\right)\left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij}\right)^2$$

where w_i are word vector, and $\tilde{w}_j$ is the context vector, b_i and $\tilde{b}_j$ are some bias terms, f is a weighting function.

You can find details of model in GloVe.

In this homework, we use **pre-trained** Glove features.

Pre-process the Data

1. Obtain movie reviews from files
2. Tokenize the reviews using the nltk package
3. Organize the tokens by their frequency
4. Assign text to ID
5. Assign index ID 0 to Unknown token
6. Convert ID back to text

You need to

- Install NLTK (Natural Language Toolkit) python package
- Take pieces of code provided on website into a file called **preprocess_data.py** within your assignment directory
- Run the file on Bluewater (data set is already provided) with a **GPU** job

Part 1 Bags of Words Model

- A bag of words model is one of the most basic models for document classification.
- Take varying length inputs (sequence of token ids) to get a single output (positive or negative sentiment)
- The token IDs can be thought of as an alternative representation of a 1-hot vector.

1a Without GloVe Features

- Create a new directory ‘1a/’ within your assignment directory.
- Create two files:
  - **BOW_model.py**
  - **BOW_sentiment_analysis.py**

```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable
import torch.distributed as dist

class BOW_model(nn.Module):
    def __init__(self, vocab_size, no_of_hidden_units):
        super(BOW_model, self).__init__()
        ## will need to define model architecture

    def forward(self, x, t):
        # will need to define forward function for when model gets called
```

1a BOW_model.py and **Embedding Layer**

- A word embedding layer is a matrix multiplication with a bias term
- The input to the embedding layer is just an **index** while a linear layer requires an appropriately sized vector.
- The **token IDs** can be thought of as an alternative representation of a 1-hot vector.

```
def __init__(self, vocab_size, no_of_hidden_units):
    super(BOW_model, self).__init__()

    self.embedding = nn.Embedding(vocab_size,no_of_hidden_units)

    self.fc_hidden = nn.Linear(no_of_hidden_units,no_of_hidden_units)
    self.bn_hidden = nn.BatchNorm1d(no_of_hidden_units)
    self.dropout = torch.nn.Dropout(p=0.5)

    self.fc_output = nn.Linear(no_of_hidden_units, 1)

    self.loss = nn.BCEWithLogitsLoss()
```

```
def forward(self, x, t):

    bow_embedding = []
    for i in range(len(x)):
        lookup_tensor = Variable(torch.LongTensor(x[i])).cuda()
        embed = self.embedding(lookup_tensor)
        embed = embed.mean(dim=0)
        bow_embedding.append(embed)
    bow_embedding = torch.stack(bow_embedding)

    h = self.dropout(F.relu(self.bn_hidden(self.fc_hidden(bow_embedding))))
    h = self.fc_output(h)

    return self.loss(h[:,0],t), h[:,0]
```

1a BOW_sentiment_analysis.py

- Convert any token id greater than the dictionary size to the **unknown** token **ID 0**
- Only the first 25,000 are labeled
- The first 125,000 are labeled 1 for positive
- The last 12,500 are labeled 0 for negative

```
#imdb_dictionary = np.load('../preprocessed_data/imdb_dictionary.npy')
vocab_size = 8000

x_train = []
with io.open('../preprocessed_data/imdb_train.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    x_train.append(line)
x_train = x_train[0:25000]
y_train = np.zeros((25000,))
y_train[0:12500] = 1
```

1a Notes

- Take about 15-30 minutes to train
- Report results for this model as well as **at least two other trials** (run more but only report on two interesting cases).
- Try to get a model that **overfits** (high accuracy on training data, lower on testing data) as well as one that **underfits** (low accuracy on both).

Other things to notice (listed on website):

- Early stopping - notice how sometimes the test performance will actually get worse the longer you train
- Longer training - this might be necessary depending on the choice of hyperparameters
- Learning rate schedule - after a certain number of iterations or once test performance stops increasing, try reducing the learning rate by some factor
- Larger models may take up too much memory meaning you may need to reduce the batch size
- Play with the dictionary size - see if there is any difference utilizing a large portion of the dictionary (like 100k) compared to very few words (500)
- You could try removing stop words, `nltk.corpus.stopwords.words('english')` returns a list of stop words. You can find their corresponding index using the `imdb_dictionary` and convert any of these to 0 when loading the data into `x_train` (remember to add 1 to the index to compensate for the unknown token)
- Try SGD optimizer instead of adam. This might take more epochs. Sometimes a well tuned SGD with momentum performs better

1b - Using GloVe Features

- Create a directory ‘1b/’.
- Copy **BOW_model.py** and **BOW_sentiment_analysis.py** from 1a and edit them
- We don’t need the embedding layer in **BOW_model.py**

```
class BOW_model(nn.Module):
    def __init__(self, no_of_hidden_units):
        super(BOW_model, self).__init__()

        self.fc_hidden1 = nn.Linear(300,no_of_hidden_units)
        self.bn_hidden1 = nn.BatchNorm1d(no_of_hidden_units)
        self.dropout1 = torch.nn.Dropout(p=0.5)

        # self.fc_hidden2 = nn.Linear(no_of_hidden_units,no_of_hidden_units)
        # self.bn_hidden2 = nn.BatchNorm1d(no_of_hidden_units)
        # self.dropout2 = torch.nn.Dropout(p=0.5)

        self.fc_output = nn.Linear(no_of_hidden_units, 1)

        self.loss = nn.BCEWithLogitsLoss()

    def forward(self, x, t):

        h = self.dropout1(F.relu(self.bn_hidden1(self.fc_hidden1(x))))
        # h = self.dropout2(F.relu(self.bn_hidden2(self.fc_hidden2(h))))
        h = self.fc_output(h)

        return self.loss(h[:,0],t), h[:,0]
```

1b BOW_sentiment_analysis.py

- Load in the glove_embeddings matrix
- Convert all out-of-dictionary tokens to the unknown token for each review
- Extract the embedding for each token in the sequence from the matrix
- Take the mean of these embeddings, and append this to the x_train or x_test list.

```
glove_embeddings = np.load('../preprocessed_data/glove_embeddings.npy')
vocab_size = 100000

x_train = []
with io.open('../preprocessed_data/imdb_train_glove.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0
    line = line[line!=0]

    line = np.mean(glove_embeddings[line],axis=0)

    x_train.append(line)
x_train = np.asarray(x_train)
x_train = x_train[0:25000]
y_train = np.zeros((25000,))
y_train[0:12500] = 1
```

Part 1 Results Analysis

- In part 1a, test accuracy achieves its max after the 3rd epoch and begins to decrease with more training while the training accuracy continues to increase well into 90+%. → **Overfitting**
- In part 1b, the training accuracy stops much earlier (around 86-88%) and doesn't seem to improve much more.
- Nearly 95% of the weights belong to the embedding layer in part 1a.
- We're training significantly less in part 1b and can't actually fine-tune the word embeddings at all.
- Using only 300 hidden weights for part 1b results in very little overfitting while still achieving decent accuracy.

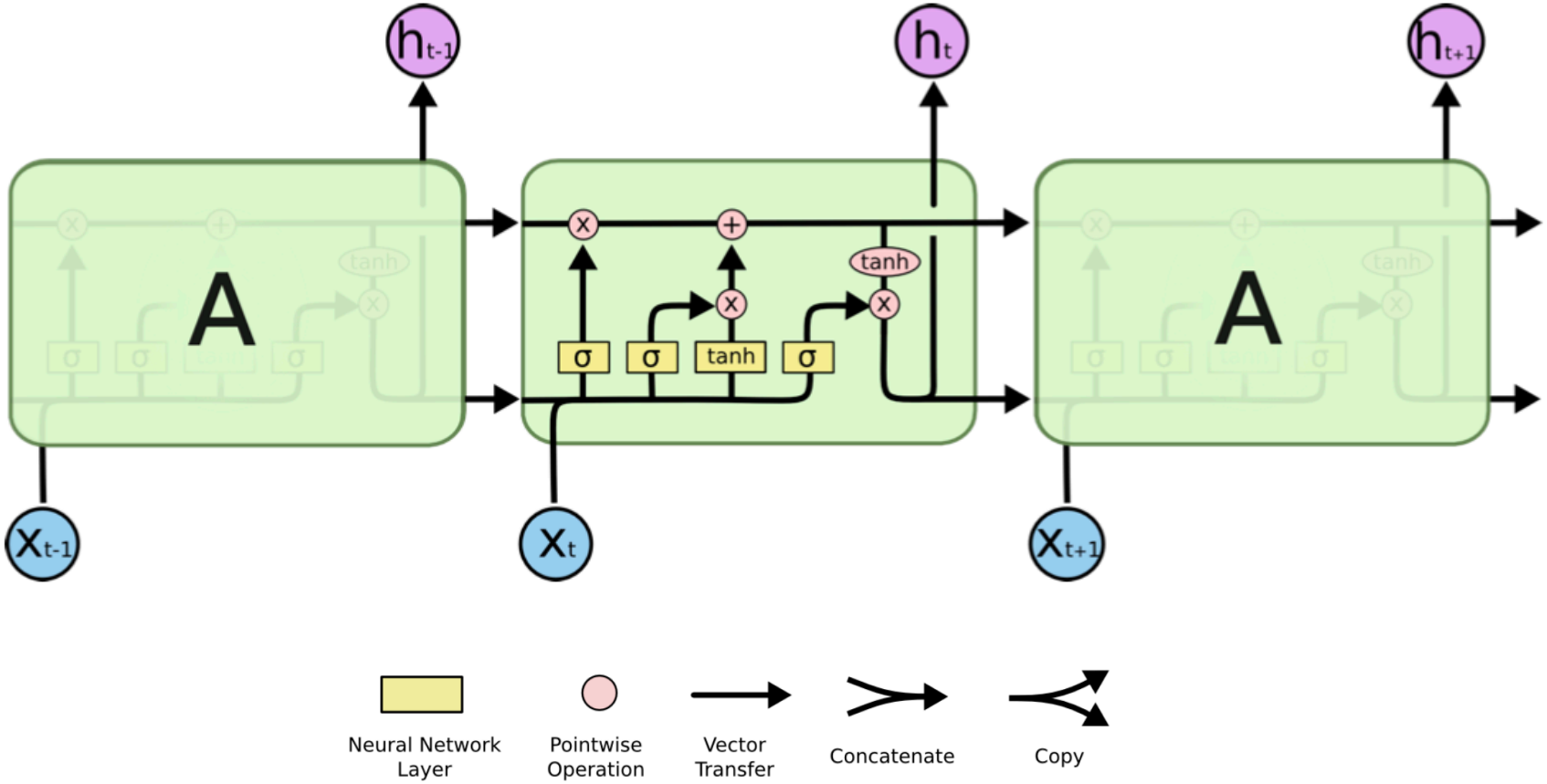
Part 2 - Recurrent Neural Network

- Temporal information matters

  "Although the movie had great visual effects, I thought it was terrible." → negative
  VS "Although the movie had terrible visual effects, I thought it was great." → positive

- Train a recurrent neural networks built with LSTM layers
  - 2a - Without GloVe Features
  - 2b - With GloVe Features

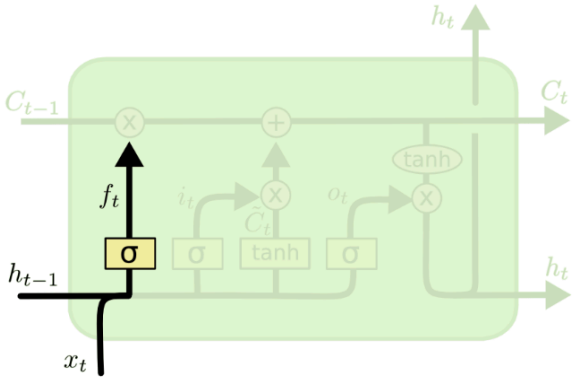
LSTM (Long short-term memory)



https://colah.github.io/posts/2015-08-Understanding-LSTMs/

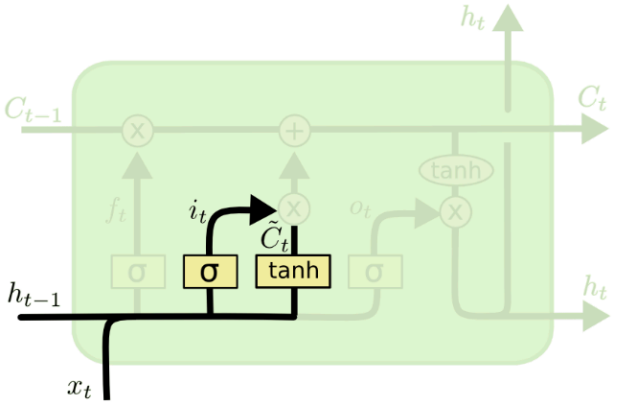
LSTM continue

forget gate layer



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

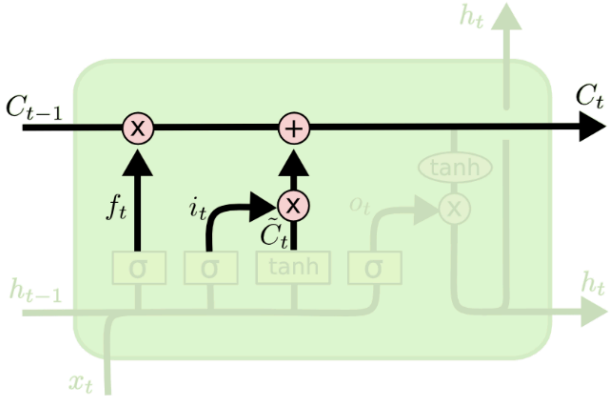
input gate layer



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

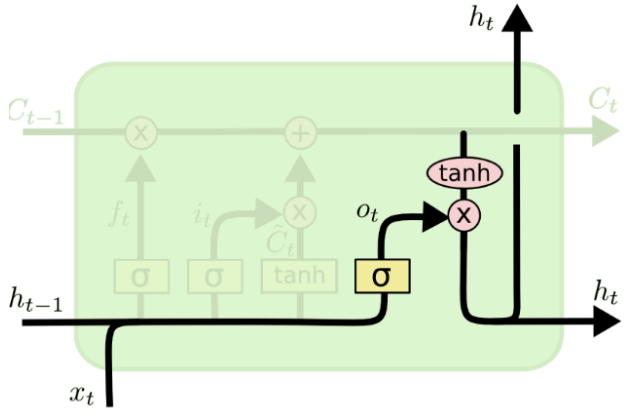
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

update the old cell state



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

output



$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

2a RNN_model.py

```
class StatefulLSTM(nn.Module):
    def __init__(self,in_size,out_size):
        super(StatefulLSTM,self).__init__()

        self.lstm = nn.LSTMCell(in_size,out_size)
        self.out_size = out_size

        self.h = None
        self.c = None

    def reset_state(self):
        self.h = None
        self.c = None

    def forward(self,x):

        batch_size = x.data.size()[0]
        if self.h is None:
            state_size = [batch_size, self.out_size]
            self.c = Variable(torch.zeros(state_size)).cuda()
            self.h = Variable(torch.zeros(state_size)).cuda()
        self.h, self.c = self.lstm(x,(self.h,self.c))

        return self.h
```

```
class LockedDropout(nn.Module):
    def __init__(self):
        super(LockedDropout,self).__init__()
        self.m = None

    def reset_state(self):
        self.m = None

    def forward(self, x, dropout=0.5, train=True):
        if train==False:
            return x
        if(self.m is None):
            self.m = x.data.new(x.size()).bernoulli_(1 - dropout)
        mask = Variable(self.m, requires_grad=False) / (1 - dropout)

        return mask * x
```

- Apply the same dropout mask after every timestep
- It is equivalent to drop word types at random rather than word tokens
- For example, the sentence “the dog and the cat” might become “— dog and — cat” or “the — and the cat”, but never “— dog and the cat”
- It has been shown in here that this technique makes training more efficient, i.e. converges faster.

2a RNN_model.py continue

- Additional input variable **train** to deal with LockedDropout() layer
- Each subsequent output has the **context** of all words preceding it.
- To the sentiment for the entire sequence, we use a max pooling operation **after** processing the **temporal information**.

```
def forward(self, x, t, train=True):

    embed = self.embedding(x) # batch_size, time_steps, features

    no_of_timesteps = embed.shape[1]

    self.reset_state()

    outputs = []
    for i in range(no_of_timesteps):

        h = self.lstm1(embed[:,i,:])
        h = self.bn_lstm1(h)
        h = self.dropout1(h,dropout=0.5,train=train)

        # h = self.lstm2(h)
        # h = self.bn_lstm2(h)
        # h = self.dropout2(h,dropout=0.3,train=train)

        outputs.append(h)

    outputs = torch.stack(outputs) # (time_steps,batch_size,features)
    outputs = outputs.permute(1,2,0) # (batch_size,features,time_steps)

    pool = nn.MaxPool1d(no_of_timesteps)
    h = pool(outputs)
    h = h.view(h.size(0),-1)
    #h = self.dropout(h)

    h = self.fc_output(h)

    return self.loss(h[:,0],t), h[:,0]#F.softmax(h, dim=1)
```

2a RNN_sentiment_analysis.py

- We train on a fixed sequence length even though the reviews have varying length
- x_input2 is a list with each element being a list of token ids for a single review
- x_input has our appropriate dimensions of **batch_size by sequence_length** (e.g. 100)

```
x_input2 = [x_train[j] for j in I_permutation[i:i+batch_size]]
sequence_length = 100
x_input = np.zeros((batch_size,sequence_length),dtype=np.int)
for j in range(batch_size):
    x = np.asarray(x_input2[j])
    sl = x.shape[0]
    if(sl < sequence_length):
        x_input[j,0:sl] = x
    else:
        start_index = np.random.randint(sl-sequence_length+1)
        x_input[j,:] = x[start_index:(start_index+sequence_length)]
y_input = y_train[I_permutation[i:i+batch_size]]

data = Variable(torch.LongTensor(x_input)).cuda()
target = Variable(torch.FloatTensor(y_input)).cuda()

optimizer.zero_grad()
loss, pred = model(data,target,train=True)
loss.backward()
```

2a Sample Results

Epoch / 50, test accuracy, test loss, time

```
50  76.14 0.7071 17.5213
100  81.74 0.5704 35.1576
150  84.51 0.4760 57.9063
200  86.10 0.4200 84.7308
250  86.69 0.3985 115.8944
300  86.98 0.3866 156.6962
350  87.00 0.3783 203.2236
400  87.25 0.3734 257.9246
450  87.22 0.3726 317.1263
```

- Performs worse than the bag of words model from part 1a
- Why? Because it is already overfitting before and we vastly increased the size of the model

Solution:

- Train on shorter sequences
- Apply enough regularization/dropout

2b - With GloVe Features

- Same RNN_model.py except embedding layer is removed from before.
- Modify the **RNN_sentiment_analysis.py** from part 2a to work with GloVe features
- Load the data
- Make sure to extract the glove_embeddings as shown above before sending it into the model.

```
glove_embeddings = np.load('../preprocessed_data/glove_embeddings.npy')
vocab_size = 100000

x_train = []
with io.open('../preprocessed_data/imdb_train_glove.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    x_train.append(line)
x_train = x_train[0:25000]
y_train = np.zeros((25000,))
y_train[0:12500] = 1

x_test = []
with io.open('../preprocessed_data/imdb_test_glove.txt','r',encoding='utf-8') as f:
    lines = f.readlines()
for line in lines:
    line = line.strip()
    line = line.split(' ')
    line = np.asarray(line,dtype=np.int)

    line[line>vocab_size] = 0

    x_test.append(line)
y_test = np.zeros((25000,))
y_test[0:12500] = 1

vocab_size += 1

model = RNN_model(500)
model.cuda()
```

2b Sample Results

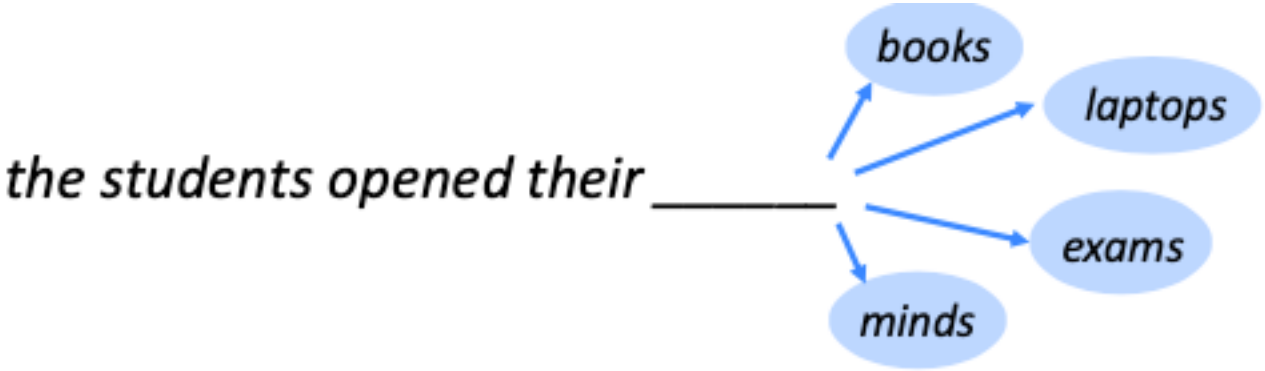
Epoch / 50, test accuracy, test loss, time

```
50 78.94 0.5010 24.4657
100 84.63 0.3956 48.4858
150 87.62 0.3240 75.0972
200 89.18 0.2932 109.8104
250 89.90 0.2750 149.9544
300 90.50 0.2618 193.7136
350 90.83 0.2530 245.5717
400 90.90 0.2511 299.2605
450 91.08 0.2477 360.4909
500 91.19 0.2448 428.9970
```

- GloVe features are much better utilized when allowing the model to handle the temporal information
- Overfitting seems to be less of a problem

Part 3 Language Model

- Language Modeling is the task of **predicting what word comes next**.
- Feed sequences into a recurrent neural network
- Then train the model to predict the following word
- No need for additional data labeling. The words themselves are the labels.



3a RNN_language_model.py

```
def forward(self, x, train=True):


    embed = self.embedding(x) # batch_size, time_steps, features

    no_of_timesteps = embed.shape[1]-1

    self.reset_state()

    outputs = []
    for i in range(no_of_timesteps):

        h = self.lstm1(embed[:,i,:])
        h = self.bn_lstm1(h)
        h = self.dropout1(h,dropout=0.3,train=train)

        h = self.lstm2(h)
        h = self.bn_lstm2(h)
        h = self.dropout2(h,dropout=0.3,train=train)

        h = self.lstm3(h)
        h = self.bn_lstm3(h)
        h = self.dropout3(h,dropout=0.3,train=train)

        h = self.decoder(h)

        outputs.append(h)
```

```
    outputs = torch.stack(outputs) # (time_steps,batch_size,vocab_size)
    target_prediction = outputs.permute(1,0,2) # batch, time, vocab
    outputs = outputs.permute(1,2,0) # (batch_size,vocab_size,time_steps)

    if(train==True):

        target_prediction = target_prediction.contiguous().view(-1,self.vocab_size)
        target = x[:,1:].contiguous().view(-1)
        loss = self.loss(target_prediction,target)

        return loss, outputs
    else:
        return outputs
```

- Three stacked LSTM layers
- The final layer has more outputs (called decoder) self.decoder = nn.Linear(no_of_hidden_units, vocab_size)
- After every timestep, we have an output for a **probability distribution over the entire vocabulary**.

3a train_language_model.py

- Copy **RNN_sentiment_analysis.py** from part 2a.
- Make sure to import the **RNN_language_model** you made above.
- Recurrent neural networks can sometimes experience extremely large gradients for a single batch
  → Use Gradient clipping $\tilde{G} = \max\left(-c, \min(G, c)\right)$

```
norm = nn.utils.clip_grad_norm_(model.parameters(),2.0)
```

- Takes about a day to train (**start early!)**
- **Low** accuracy

3b Generating Fake Reviews

- Predict P(w_n|w_0, …, w_n-1)
- Pick some starting words, e.g. “a” , “i”, or “I love this move”
- Use the trained language model from part 3a

```
# tokens = [['i', 'love', 'this', 'movie', '.'],['i', 'hate', 'this', 'movie', '.']]
tokens = [['a'],['i']]

token_ids = np.asarray([[word_to_id.get(token,-1)+1 for token in x] for x in tokens])

## preload phrase
x = Variable(torch.LongTensor(token_ids)).cuda()

embed = model.embedding(x) # batch_size, time_steps, features

state_size = [embed.shape[0],embed.shape[2]] # batch_size, features
no_of_timesteps = embed.shape[1]

model.reset_state()

outputs = []
for i in range(no_of_timesteps):

    h = model.lstm1(embed[:,i,:])
    h = model.bn_lstm1(h)
    h = model.dropout1(h,dropout=0.3,train=False)

    h = model.lstm2(h)
    h = model.bn_lstm2(h)
    h = model.dropout2(h,dropout=0.3,train=False)

    h = model.lstm3(h)
    h = model.bn_lstm3(h)
    h = model.dropout3(h,dropout=0.3,train=False)

    h = model.decoder(h)

    outputs.append(h)

outputs = torch.stack(outputs)
outputs = outputs.permute(1,2,0)
output = outputs[:,:,-1]
```

3b Generating Fake Reviews - continue

```
temperature = 1.0 # float(sys.argv[1])
length_of_review = 150

review = []
####
for j in range(length_of_review):

    ## sample a word from the previous output
    output = output/temperature
    probs = torch.exp(output)
    probs[:,0] = 0.0
    probs = probs/(torch.sum(probs,dim=1).unsqueeze(1))
    x = torch.multinomial(probs,1)
    review.append(x.cpu().data.numpy()[:,0])

    ## predict the next word
    embed = model.embedding(x)

    h = model.lstm1(embed)
    h = model.bn_lstm1(h)
    h = model.dropout1(h,dropout=0.3,train=False)

    h = model.lstm2(h)
    h = model.bn_lstm2(h)
    h = model.dropout2(h,dropout=0.3,train=False)

    h = model.lstm3(h)
    h = model.bn_lstm3(h)
    h = model.dropout3(h,dropout=0.3,train=False)

    output = model.decoder(h)
```

1. Sample a word from the previous output (apply softmax on output)
2. Predict next word
3. Generate review

Remark:

- Token id 0 ← unknown token
- Temperature

```
review = np.asarray(review)
review = review.T
review = np.concatenate((token_ids,review),axis=
review = review - 1
review[review<0] = vocab_size - 1
review_words = imdb_dictionary[review]
for review in review_words:
    prnt_str = ''
    for word in review:
        prnt_str += word
        prnt_str += ' '
    print(prnt_str)
```

3b examples # temperature 1.0

a hugely influential , very strong , nuanced stand up comedy part all too much . this is a film that keeps you laughing and cheering for your own reason to watch it . the same has to be done with actors , which is surely `` the best movie '' in recent history because at the time of the vietnam war , ca n't know who to argue they claim they have no choice ... out of human way or out of touch with personal history . even during the idea of technology they are not just up to you . there is a balance between the central characters and even the environment and all of that . the book is beautifully balanced , which is n't since the book . perhaps the ending should have had all the weaknesses of a great book but the essential flaw of the

i found it fascinating and never being even lower . spent nothing on the spanish 's particularly good filming style style . as is the songs , there 's actually a line the film moves so on ; a sequence and a couple that begins almost the same exact same so early style of lot of time , so the indians theme has not been added to the on screen . well was , the movie has to be the worst by the cast . i did however say - the overall feel of the film was superb , and for those that just do n't understand it , the movie deserves very little to go and lets you see how it takes 3 minutes to chilling . i must admit the acting was adequate , but `` jean reno '' was a pretty good job , he was very subtle

3b examples # temperature 0.25

a little slow and i found myself laughing out loud at the end . the story is about a young girl who is trying to find a way to get her to go to the house and meets a young girl who is also a very good actress . she is a great actress and is very good . she is a great actress and she is awesome in this movie . she is a great actress and i think she is a great actress . i think she is a great actress and i hope she will get more recognition . i think she has a great voice . i think she is a great actress . i think she is one of the best actresses in the world . she is a great actress and i think she is a great actress . she is a great actress and

i i was a fan of the original , but i was very disappointed . the plot was very weak , the acting was terrible , the plot was weak , the acting was terrible , the story was weak , the acting was horrible , the story was weak , the acting was bad , the plot was bad , the story was worse , the acting was bad , the plot was stupid , the acting was bad , the plot was stupid and the acting was horrible . i do n't know how anyone could have made this movie a 1 . i hope that it will be released on dvd . i hope it gets released on dvd soon . i hope that it will be released on dvd . i hope it gets released soon because it is so bad it 's good . i hope that

3c Learning Sentiment

- Copy all your code from part 2a.
- Modify RNN_model.py to have three lstm layers
- Create a new RNN_model for sentiment analysis but **copy all of the weights for the embedding and LSTM layers** from the language model.
- Define the optimizer, and choose to **just fine-tune the last LSTM layer** and the output layer (to avoid overfitting)

```
params = []
# for param in model.embedding.parameters():
#     params.append(param)
# for param in model.lstm1.parameters():
#     params.append(param)
# for param in model.bn_lstm1.parameters():
#     params.append(param)
# for param in model.lstm2.parameters():
#     params.append(param)
# for param in model.bn_lstm2.parameters():
#     params.append(param)
for param in model.lstm3.parameters():
    params.append(param)
for param in model.bn_lstm3.parameters():
    params.append(param)
for param in model.fc_output.parameters():
    params.append(param)

opt = 'adam'
LR = 0.001
if(opt=='adam'):
    optimizer = optim.Adam(params, lr=LR)
elif(opt=='sgd'):
    optimizer = optim.SGD(params, lr=LR, momentum=0.9)
```

Sample Results of 3c

Epoch / 50, test accuracy, test loss, time

```
50 80.98 0.4255 17.0083
100 87.17 0.3043 30.2916
150 90.18 0.2453 45.0554
200 91.15 0.2188 59.9038
250 91.96 0.2022 74.8118
300 92.34 0.1960 89.7251
350 92.64 0.1901 104.7904
400 92.83 0.1863 119.8761
450 92.95 0.1842 134.8656
500 93.01 0.1828 150.3047
```

The state of the art on the IMDB sentiment analysis is around 97.4 %.

https://paperswithcode.com/sota/sentiment-analysis-on-imdb

HW5 Tasks

- For each of 1a, 1b, 2a, 2b, train with the desired model and two other trials

- No need to write your own codes, just take the pieces of code provided on the website, combine them, and edit them for different tasks
- Tips: make sure you create a new directory when you run a new task

HW6 Tasks

- For 3a, only train the model with the given hyperparameters
- For 3b, generate fake movie results
- For 3c, train with the desired model and two other trials

What to submit

- Make a copy of the google sheet here
- Fill in the sheet
- Zip all of your code (excluding the dataset/preprocessed data/trained PyTorch models).
- Save your completed google sheet as a PDF.
- Submit the code and PDF on compass.

	A	B	C	D	E	F	G	H
	Natural Language Processing - IMDB Movie Review							
		Description	Hyperparameters	Number of Epochs	Training Loss	Training Accuracy	Test Accuracy	Comments
	Part 1a	Given model - Word Embedding Layer + Mean Pooling + Fully Connected Layer + Relu + Output Layer	ADAM optimizer with LR=0.001, BatchSize=200, VocabularySize=8000, HiddenUnits=500	6			should be around 86-88%	Describe more about the model/results such as why certain hyperparamters were chosen or the effect it had on the accuracy/training time/overfitting/etc.
		Custom 1						
		Custom 2						
		Custom 3						
	Part 1b	Given Model - (write description)					Should be around ~81-87%	
		Custom 1						
		Custom 2						
		Custom 3						
	Part 2a	Given Model - (write					~87%	

Q&A

References

Stanford NLP course slides

The Illustrated Word2vec

Explanation of Word Embedding