CS 224D: Deep Learning for NLP[1]

Lecture Notes: Part I[2]

Spring 2015

[1] Course Instructor: Richard Socher

[2] Authors: Francois Chaubard, Rohit Mundra, Richard Socher

Keyphrases: Natural Language Processing. Word Vectors. Singular Value Decomposition. Skip-gram. Continuous Bag of Words (CBOW). Negative Sampling.

This set of notes begins by introducing the concept of Natural Language Processing (NLP) and the problems NLP faces today. We then move forward to discuss the concept of representing words as numeric vectors. Lastly, we discuss popular approaches to designing word vectors.

1 Introduction to Natural Language Processing

We begin with a general discussion of what is NLP. The goal of NLP is to be able to design algorithms to allow computers to "understand" natural language in order to perform some task. Example tasks come in varying level of difficulty:

Easy

- Spell Checking

- Keyword Search

- Finding Synonyms

Medium

- Parsing information from websites, documents, etc.

Hard

- Machine Translation (e.g. Translate Chinese text to English)

- Semantic Analysis (What is the meaning of query statement?)

- Coreference (e.g. What does "he" or "it" refer to given a document?)

- Question Answering (e.g. Answering Jeopardy questions).

The first and arguably most important common denominator across all NLP tasks is how we represent words as input to any and all of our models. Much of the earlier NLP work that we will not cover treats words as atomic symbols. To perform well on most NLP tasks we first need to have some notion of similarity and difference

Natural Language Processing tasks come in varying levels of difficulty:

Easy

- Spell Checking
- Keyword Search
- Finding Synonyms

Medium

- Parsing information from websites, documents, etc.

Hard

- Machine Translation
- Semantic Analysis
- Coreference
- Question Answering

between words. With word vectors, we can quite easily encode this ability in the vectors themselves (using distance measures such as Jaccard, Cosine, Euclidean, etc).

2   Word Vectors

There are an estimated 13 million tokens for the English language but are they all completely unrelated? Feline to cat, hotel to motel? I think not. Thus, we want to encode word tokens each into some vector that represents a point in some sort of "word" space. This is paramount for a number of reasons but the most intuitive reason is that perhaps there actually exists some N-dimensional space (such that $N \ll 13$ million) that is sufficient to encode all semantics of our language. Each dimension would encode some meaning that we transfer using speech. For instance, semantic dimensions might indicate tense (past vs. present vs. future), count (singular vs. plural), and gender (masculine vs. feminine).

So let's dive into our first word vector and arguably the most simple, the **one-hot vector**: Represent every word as an $\mathbb{R}^{|V| \times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language. In this notation, $|V|$ is the size of our vocabulary. Word vectors in this type of encoding would appear as the following:

$$w^{aardvark} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^a = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, w^{at} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \cdots w^{zebra} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

We represent each word as a completely independent entity. As we previously discussed, this word representation does not give us directly any notion of similarity. For instance,

$$(w^{hotel})^T w^{motel} = (w^{hotel})^T w^{cat} = 0$$

So maybe we can try to reduce the size of this space from $\mathbb{R}^{|V|}$ to something smaller and thus find a subspace that encodes the relationships between words.

3   SVD Based Methods

For this class of methods to find word embeddings (otherwise known as word vectors), we first loop over a massive dataset and accumulate word co-occurrence counts in some form of a matrix X, and then perform Singular Value Decomposition on X to get a USV^T decomposition. We then use the rows of U as the word embeddings for all

One-hot vector: Represent every word as an $\mathbb{R}^{|V| \times 1}$ vector with all 0s and one 1 at the index of that word in the sorted english language.

Fun fact: The term "one-hot" comes from digital circuit design, meaning "a group of bits among which the legal combinations of values are only those with a single high (1) bit and all the others low (0)".

words in our dictionary. Let us discuss a few choices of X.

3.1   Word-Document Matrix

As our first attempt, we make the bold conjecture that words that are related will often appear in the same documents. For instance, "banks", "bonds", "stocks", "money", etc. are probably likely to appear together. But "banks", "octopus", "banana", and "hockey" would probably not consistently appear together. We use this fact to build a word-document matrix, X in the following manner: Loop over billions of documents and for each time word i appears in document j, we add one to entry X_{ij}. This is obviously a very large matrix ($\mathbb{R}^{|V| \times M}$) and it scales with the number of documents (M). So perhaps we can try something better.

3.2   Word-Word Co-occurrence Matrix

The same kind of logic applies here however, the matrix X stores co-occurrences of words thereby becoming an affinity matrix. We display an example one below. Let our corpus contain just three sentences:

1. I enjoy flying.

2. I like NLP.

3. I like deep learning.

The resulting counts matrix will then be:

$$
X = \begin{array}{c} \\ I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \begin{array}{c} \begin{array}{cccccccc} I & like & enjoy & deep & learning & NLP & flying & . \end{array} \\ \left[\begin{array}{cccccccc} 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{array} \right] \end{array}
$$

We now perform SVD on X, observe the singular values (the diagonal entries in the resulting S matrix), and cut them off at some index k based on the desired percentage variance captured:

$$
\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}
$$

Using Word-Word Co-occurrence Matrix:

- Generate $|V| \times |V|$ co-occurrence matrix, X.

- Apply SVD on X to get $X = USV^T$.

- Select the first k columns of U to get a k-dimensional word vectors.

- $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{|V|} \sigma_i}$ indicates the amount of variance captured by the first k dimensions.

We then take the submatrix of $U_{1:|V|,1:k}$ to be our word embedding matrix. This would thus give us a k-dimensional representation of every word in the vocabulary.

Applying SVD to X:

$$
|V|\begin{bmatrix} & & \\ & X & \\ & & \end{bmatrix} = |V|\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix} |V|\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} |V|\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}
$$

Reducing dimensionality by selecting first k singular vectors:

$$
|V|\begin{bmatrix} & & \\ & \hat{X} & \\ & & \end{bmatrix} = |V|\begin{bmatrix} | & | & \\ u_1 & u_2 & \cdots \\ | & | & \end{bmatrix} k\begin{bmatrix} \sigma_1 & 0 & \cdots \\ 0 & \sigma_2 & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} k\begin{bmatrix} - & v_1 & - \\ - & v_2 & - \\ & \vdots & \end{bmatrix}
$$

Both of these methods give us word vectors that are more than sufficient to encode semantic and syntactic (part of speech) information but are associated with many other problems:

- The dimensions of the matrix change very often (new words are added very frequently and corpus changes in size).

- The matrix is extremely sparse since most words do not co-occur.

- The matrix is very high dimensional in general ($\approx 10^6 \times 10^6$)

- Quadratic cost to train (i.e. to perform SVD)

- Requires the incorporation of some hacks on X to account for the drastic imbalance in word frequency

Some solutions to exist to resolve some of the issues discussed above:

- Ignore function words such as "the", "he", "has", etc.

- Apply a ramp window – i.e. weight the co-occurrence count based on distance between the words in the document.

- Use Pearson correlation and set negative counts to 0 instead of using just raw count.

As we see in the next section, iteration based methods solve many of these issues in a far more elegant manner.

4   Iteration Based Methods

Let us step back and try a new approach. Instead of computing and storing global information about some huge dataset (which might be billions of sentences), we can try to create a model that will be able to learn one iteration at a time and eventually be able to encode the probability of a word given its context.

We can set up this probabilistic model of known and unknown parameters and take one training example at a time in order to learn just a little bit of information for the unknown parameters based on the input, the output of the model, and the desired output of the model.

At every iteration we run our model, evaluate the errors, and follow an update rule that has some notion of penalizing the model parameters that caused the error. This idea is a very old one dating back to 1986. We call this method "backpropagating" the errors (see LEARNING REPRESENTATIONS BY BACK-PROPAGATING ERRORS. DAVID E. RUMELHART, GEOFFREY E. HINTON, AND RONALD J. WILLIAMS (1988).)

Context of a word:
The context of a word is the set of C surrounding words. For instance, the $C = 2$ context of the word "fox" in the sentence "The quick brown fox jumped over the lazy dog" is {"quick", "brown", "jumped", "over"}.

4.1   Language Models (Unigrams, Bigrams, etc.)

First, we need to create such a model that will assign a probability to a sequence of tokens. Let us start with an example:

"The cat jumped over the puddle."

A good language model will give this sentence a high probability because this is a completely valid sentence, syntactically and semantically. Similarly, the sentence "stock boil fish is toy" should have a very low probability because it makes no sense. Mathematically, we can call this probability on any given sequence of n words:

$$P(w^{(1)}, w^{(2)}, \cdots, w^{(n)})$$

We can take the unary language model approach and break apart this probability by assuming the word occurrences are completely independent:

$$P(w^{(1)}, w^{(2)}, \cdots, w^{(n)}) = \prod_{i=1}^{n} P(w^{(i)})$$

However, we know this is a bit ludicrous because we know the next word is highly contingent upon the previous sequence of words. And the silly sentence example might actually score highly. So perhaps we let the probability of the sequence depend on the pairwise

Unigram model:

$$P(w^{(1)}, w^{(2)}, \cdots, w^{(n)}) = \prod_{i=1}^{n} P(w^{(i)})$$

probability of a word in the sequence and the word next to it. We call this the bigram model and represent it as:

$$P(w^{(1)}, w^{(2)}, \cdots, w^{(n)}) = \prod_{i=2}^{n} P(w^{(i)}|w^{(i-1)})$$

Again this is certainly a bit naive since we are only concerning ourselves with pairs of neighboring words rather than evaluating a whole sentence, but as we will see, this representation gets us pretty far along. Note in the Word-Word Matrix with a context of size 1, we basically can learn these pairwise probabilities. But again, this would require computing and storing global information about a massive dataset.

Now that we understand how we can think about a sequence of tokens having a probability, let us observe some example models that could learn these probabilities.

4.2 Continuous Bag of Words Model (CBOW)

One approach is to treat {"The", "cat", 'over", "the', "puddle"} as a context and from these words, be able to predict or generate the center word "jumped". This type of model we call a Continuous Bag of Words (CBOW) Model.

Let's discuss the CBOW Model above in greater detail. First, we set up our known parameters. Let the known parameters in our model be the sentence represented by one-hot word vectors. The input one hot vectors or context we will represent with an $x^{(i)}$. And the output as $y^{(i)}$ and in the CBOW model, since we only have one output, so we just call this y which is the one hot vector of the known center word. Now let's define our unknowns in our model.

We create two matrices, $W^{(1)} \in \mathbb{R}^{n \times |V|}$ and $W^{(2)} \in \mathbb{R}^{|V| \times n}$. Where n is an arbitrary size which defines the size of our embedding space. $W^{(1)}$ is the input word matrix such that the i-th column of $W^{(1)}$ is the n-dimensional embedded vector for word $w^{(i)}$ when it is an input to this model. We denote this $n \times 1$ vector as $u^{(i)}$. Similarly, $W^{(2)}$ is the output word matrix. The j-th row of $W^{(2)}$ is an n-dimensional embedded vector for word $w^{(j)}$ when it is an output of the model. We denote this row of $W^{(2)}$ as $v^{(j)}$. Note that we do in fact learn two vectors for every word $w^{(i)}$ (i.e. input word vector $u^{(i)}$ and output word vector $v^{(i)}$).

We breakdown the way this model works in these steps:

1. We generate our one hot word vectors $(x^{(i-C)}, \ldots, x^{(i-1)}, x^{(i+1)}, \ldots, x^{(i+C)})$ for the input context of size C.

Bigram model:

$$P(w^{(1)}, w^{(2)}, \cdots, w^{(n)}) = \prod_{i=2}^{n} P(w^{(i)}|w^{(i-1)})$$

CBOW Model:
  Predicting a center word form the surrounding context

Notation for CBOW Model:

- $w^{(i)}$: Word i from vocabulary V
- $W^{(1)} \in \mathbb{R}^{n \times |V|}$: Input word matrix
- $u^{(i)}$: i-th column of $W^{(1)}$, the input vector representation of word $w^{(i)}$
- $W^{(2)} \in \mathbb{R}^{n \times |V|}$: Output word matrix
- $v^{(i)}$: i-th row of $W^{(2)}$, the output vector representation of word $w^{(i)}$

2. We get our embedded word vectors for the context ($u^{(i-C)} = W^{(1)}x^{(i-C)}$, $u^{(i-C+1)} = W^{(1)}x^{(i-C+1)}$, ..., $u^{(i+C)} = W^{(1)}x^{(i+C)}$)

3. Average these vectors to get $h = \frac{u^{(i-C)}+u^{(i-C+1)}+...+u^{(i+C)}}{2C}$

4. Generate a score vector $z = W^{(2)}h$

5. Turn the scores into probabilities $\hat{y} = \text{softmax}(z)$

6. We desire our probabilities generated, $\hat{y}$, to match the true probabilities, y, which also happens to be the one hot vector of the actual word.

So now that we have an understanding of how our model would work if we had a $W^{(1)}$ and $W^{(2)}$, how would we learn these two matrices? Well, we need to create an objective function. Very often when we are trying to learn a probability from some true probability, we look to information theory to give us a measure of the distance between two distributions. Here, we use a popular choice of distance/loss measure, cross entropy $H(\hat{y}, y)$.

The intuition for the use of cross-entropy in the discrete case can be derived from the formulation of the loss function:

$$H(\hat{y}, y) = -\sum_{j=1}^{|V|} y_j \log(\hat{y}_j)$$

Let us concern ourselves with the case at hand, which is that y is a one-hot vector. Thus we know that the above loss simplifies to simply:

$$H(\hat{y}, y) = -y_i \log(\hat{y}_i)$$

In this formulation, i is the index where the correct word's one hot vector is 1. We can now consider the case where our prediction was perfect and thus $\hat{y}_i = 1$. We can then calculate $H(\hat{y}, y) = -1\log(1) = 0$. Thus, for a perfect prediction, we face no penalty or loss. Now let us consider the opposite case where our prediction was very bad and thus $\hat{y}_i = 0.01$. As before, we can calculate our loss to be $H(\hat{y}, y) = -1\log(0.01) \approx 4.605$. We can thus see that for probability distributions, cross entropy provides us with a good measure of
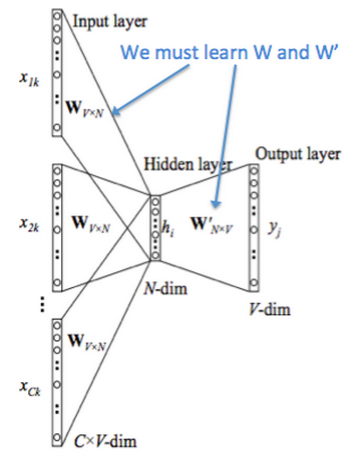

Figure 1: This image demonstrates how CBOW works and how we must learn the transfer matrices

distance. We thus formulate our optimization objective as:

$$
\begin{aligned}
\text{minimize } J &= -\log P(w^{(i)}|w^{(i-C)},\ldots,w^{(i-1)},w^{(i+1)},\ldots,w^{(i+C)}) \\
&= -\log P(v^{(i)}|h) \\
&= -\log \frac{\exp(v^{(i)T}h)}{\sum_{j=1}^{|V|}\exp(v^{(i)T}u^{(j)})} \\
&= -v^{(i)T}h + \log\sum_{j=1}^{|V|}\exp(v^{(i)T}u^{(j)})
\end{aligned}
$$

Since we use gradient descent to update word vectors all relevant word vectors $v^{(i)}$ and $u^{(j)}$, we calculate the gradients in the following manner:

[To be added after Assignment 1 is graded]

4.3   Skip-Gram Model

Another approach is to create a model such that given the center word "jumped", the model will be able to predict or generate the surrounding words "The", "cat", "over", "the", "puddle". Here we call the word "jumped" the context. We call this type of model a Skip-Gram model.

Let's discuss the Skip-Gram model above. The setup is largely the same but we essentially swap our x and y i.e. x in the CBOW are now y and vice-versa. The input one hot vector (center word) we will represent with an x (since there is only one). And the output vectors as $y^{(j)}$. We define $W^{(1)}$ and $W^{(2)}$ the same as in CBOW.

Notation for Skip-Gram Model:
- $w^{(i)}$: Word i from vocabulary V
- $W^{(1)} \in \mathbb{R}^{n \times |V|}$: Input word matrix
- $u^{(i)}$: i-th column of $W^{(1)}$, the input vector representation of word $w^{(i)}$
- $W^{(2)} \in \mathbb{R}^{n \times |V|}$: Output word matrix
- $v^{(i)}$: i-th row of $W^{(2)}$, the output vector representation of word $w^{(i)}$

We breakdown the way this model works in these 6 steps:

1. We generate our one hot input vector x

2. We get our embedded word vectors for the context $u^{(i)} = W^{(1)}x$

3. Since there is no averaging, just set $h = u^{(i)}$?

4. Generate $2C$ score vectors, $v^{(i-C)}, \ldots, v^{(i-1)}, v^{(i+1)}, \ldots, v^{(i+C)}$ using $v = W^{(2)}h$

5. Turn each of the scores into probabilities, $y = \text{softmax}(v)$

6. We desire our probability vector generated to match the true probabilities which is $y^{(i-C)}, \ldots, y^{(i-1)}, y^{(i+1)}, \ldots, y^{(i+C)}$, the one hot vectors of the actual output.

As in CBOW, we need to generate an objective function for us to evaluate the model. A key difference here is that we invoke a Naive Bayes assumption to break out the probabilities. If you have not seen this before, then simply put, it is a strong (naive) conditional independence assumption. In other words, given the center word, all output words are completely independent.

$$\text{minimize } J = -\log P(w^{(i-C)}, \ldots, w^{(i-1)}, w^{(i+1)}, \ldots, w^{(i+C)} | w^{(i)})$$

$$= -\log \prod_{j=0, j \neq C}^{2C} P(w^{(i-C+j)} | w^{(i)})$$

$$= -\log \prod_{j=0, j \neq C}^{2C} P(v^{(i-C+j)} | u^{(i)})$$

$$= -\log \prod_{j=0, j \neq C}^{2C} \frac{\exp(v^{(i-C+j)T} h)}{\sum_{k=1}^{|V|} \exp(v^{(k)T} h)}$$

$$= -\sum_{j=0, j \neq C}^{2C} v^{(i-C+j)T} h + 2C \log \sum_{k=1}^{|V|} \exp(v^{(k)T} h)$$

Figure 2: This image demonstrates how Skip-Gram works and how we must learn the transfer matrices

With this objective function, we can compute the gradients with respect to the unknown parameters and at each iteration update them via Stochastic Gradient Descent.

[To be added after Assignment 1 is graded]

4.4   Negative Sampling

Lets take a second to look at the objective function. Note that the summation over $|V|$ is computationally huge! Any update we do or evaluation of the objective function would take $O(|V|)$ time which if we recall is in the millions. A simple idea is we could instead just approximate it.

For every training step, instead of looping over the entire vocabulary, we can just sample several negative examples! We "sample" from a noise distribution $(P_n(w))$ whose probabilities match the ordering of the frequency of the vocabulary. To augment our formulation of the problem to incorporate Negative Sampling, all we need to do is update the:

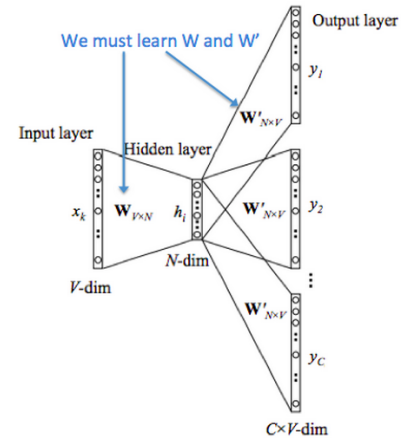• objective function

• gradients

• update rules

MIKOLOV ET AL. present **Negative Sampling** in DISTRIBUTED REPRESENTATIONS OF WORDS AND PHRASES AND THEIR COMPOSITIONALITY. While negative sampling is based on the Skip-Gram model, it is in fact optimizing a different objective. Consider a pair (w, c) of word and context. Did this pair come from the training data? Let's denote by $P(D = 1|w, c)$ the probability that (w, c) came from the corpus data. Correspondingly, $P(D = 0|w, c)$ will be the probability that (w, c) did not come from the corpus data. First, let's model $P(D = 1|w, c)$ with the sigmoid function:

$$P(D = 1|w, c, \theta) = \frac{1}{1 + e^{(-v_c^T v_w)}}$$

Now, we build a new objective function that tries to maximize the probability of a word and context being in the corpus data if it indeed is, and maximize the probability of a word and context not being in the corpus data if it indeed is not. We take a simple maximum likelihood approach of these two probabilities. (Here we take θ to be the parameters of the model, and in our case it is $W^{(1)}$ and $W^{(2)}$.)

$$\theta = \underset{\theta}{\text{argmax}} \prod_{(w,c) \in D} P(D = 1|w, c, \theta) \prod_{(w,c) \in \tilde{D}} P(D = 0|w, c, \theta)$$

$$= \underset{\theta}{\text{argmax}} \prod_{(w,c) \in D} P(D = 1|w, c, \theta) \prod_{(w,c) \in \tilde{D}} (1 - P(D = 1|w, c, \theta))$$

$$= \underset{\theta}{\text{argmax}} \sum_{(w,c) \in D} \log P(D = 1|w, c, \theta) + \sum_{(w,c) \in \tilde{D}} \log(1 - P(D = 1|w, c, \theta))$$

$$= \underset{\theta}{\text{argmax}} \sum_{(w,c) \in D} \log \frac{1}{1 + \exp(-v_c^T v_w)} + \sum_{(w,c) \in \tilde{D}} \log(1 - \frac{1}{1 + \exp(-v_c^T v_w)})$$

$$= \underset{\theta}{\text{argmax}} \sum_{(w,c) \in D} \log \frac{1}{1 + \exp(-v_c^T v_w)} + \sum_{(w,c) \in \tilde{D}} \log(\frac{1}{1 + \exp(v_c^T v_w)})$$

Note that $\tilde{D}$ is a "false" or "negative" corpus. Where we would have sentences like "stock boil fish is toy". Unnatural sentences that should get a low probability of ever occurring. We can generate $\tilde{D}$ on the fly by randomly sampling this negative from the word bank. Our new objective function would then be:

$$- \log \sigma(v^{(i-C+j)} \cdot h) + \sum_{k=1}^{K} \log \sigma(\tilde{v}^{(k)} \cdot h)$$

In the above formulation, $\{\tilde{v}^{(k)}|k = 1 \ldots K\}$ are sampled from $P_n(w)$. Let's discuss what $P_n(w)$ should be. While there is much discussion of what makes the best approximation, what seems to work best is the Unigram Model raised to the power of 3/4. Why 3/4? Here's an example that might help gain some intuition:

$$\text{is: } 0.9^{3/4} = 0.92$$
$$\text{Constitution: } 0.09^{3/4} = 0.16$$
$$\text{bombastic: } 0.01^{3/4} = 0.032$$

"Bombastic" is now 3x more likely to be sampled while "is" only went up marginally.

[Gradients and update rules to be added after Assignment 1 is graded]