Prepared By:
Sami & Ron

Deep Learning for NLP
Coding Assignment 1

CS 544
Due: Thu April 4

Introduction

The goal of this coding assignment to get you familiar with TensorFlow and walk you through some practical Deep Learning techniques. You will be starting with code that is similar to one taught in the first NLP Deep Learning class. Recall, the code we taught in class implemented a 3-layer neural network over the document vectors. The output layer classified each document into (positive/negative) and (truthful/deceptive). You will utilize the dataset from coding assignments 1 and 2. In this assignment, you will:

- Improve the Tokenization.
- Convert the first layer into an *Embedding Layer*, which makes the model somewhat more interpretable. Many recent Machine Learning efforts strive to make models more interpretable, but sometimes at the expense of prediction accuracy.
- Increase the generalization accuracy of the model by implementing *input sparse dropout* – TensorFlow's (*dense*) dropout layer does not work out-of-the-box, as explained later.
- Visualize the Learned Embeddings using t-SNE.

In order to start the assignment, please download the starter-code from:

- http://sami.haija.org/cs544/DL1/starter.py

You can run this code as:

```
python starter.py path/to/coding1/and/2/data/
```

Note: This assignment will automatically be graded by a script, which verifies the implementation of tasks one-by-one. It is important that you stick to these guidelines: Only implement your code in places marked by `** TASK`. Do not change the signatures of the methods tagged `** TASK`, or else the grading script will fail in finding them and you will get a zero for the corresponding parts. Otherwise, feel free to create as many helper functions as you wish!

Finally, you might find the first NLP Deep Learning lecture slides useful.

- This assignment is due Thursday April 4. We are working on Vocareum integration. Nonetheless, you are advised to start early (before we finish the Vocareum Integration). You can submit until April 7, but all submissions after April 4 will receive penalties.

[10 points] Task 1: Improve Tokenization

The current tokenization:

```
# ** TASK 1.
def Tokenize(comment):
  """Receives a string (comment) and returns array of tokens."""
  words = comment.split()
  return words
```

is crude. It splits on whitespaces only (spaces, tabs, new-lines). It leaves all other punctuations e.g. single- and double-quotes, exclamation marks, etc – there should be no reason to have both terms "house" and "house?" in the vocabulary. While a perfect tokenization can be quite involved, let us only slightly improve the existing one. Specifically, you should split on any non-letter. You might find the python standard re package useful.

- Update code of `Tokenize` to work as described. Correct implementation should reduce the number of tokens by about half.

[20 + 6.5 points] Task 2: Convert the 1st layer into an embedding layer

Our goal here is to replace the first layer with something equivalent to tf.nn.embedding_lookup, followed by averaging, but without using the function `tf.nn.embedding_lookup` as we aim to understand the underlying mathematics behind embeddings and we do not (yet) want to discuss variable-length representations in tensorflow[1].

The end-goal from this task is to make the output of this layer to represent every comment (document) by the average word embedding appearing in the comment. For example, if we represent the document by vector $\mathbf{x} \in \mathbb{R}^{|V|}$, with $|V|$ being the size of the vocabulary and entry x_i being the number of times word i appears in the document. Then, we would like the output of the embedding layer for document $\mathbf{x}$ to be:

$$\sigma\left(\frac{\mathbf{x}^\top Y}{||\mathbf{x}||}\right) \tag{1}$$

where σ is element wise activation function. We wish to train the embedding matrix $Y \in \mathbb{R}^{|V|\times d}$ which will embed each word in a d-dimensional space (each word embedding lives in one row of the matrix Y). The denominator $||\mathbf{x}||$ is to compute the average, which can be the L1 or the L2 norm of the vector. In this exercise, use the L2 norm. The above should make our model more interpretable. Note the following differences between the above embedding layer and a *traditional* fully-connected (FC) layer, with transformation: $\sigma\left(\mathbf{x}^\top W + \mathbf{b}\right)$.

1. FC layers have an additional bias-vector $\mathbf{b}$. We do not want the bias vector. Its presence makes the embeddings more tricky to be visualized or ported to other applications. Here, W corresponds to the embedding dictionary Y.

2. As mentioned, the input vector $\mathbf{x}$ to Equation 1 should be normalized. if $\mathbf{x}$ is a matrix, then normalization should be row-wise. (**Hint:** you can use tf.nn.l2_normalize).

3. Modern fully-connected layers are have σ = ReLu. Embeddings generally either have (1) no activation or (2) a squashing activation (e.g. tanh, or L2-norm). We will opt to use (2) specifically tanh activation, as option (1) might force us to choose an adaptive learning-rate[2] for the embedding layer.

4. The parameter W will be L2-regularized in standard FC i.e. by adding $\lambda||W||_2^2$ to the overall minimization objective function (where the scalar coefficient λ is generally set to a small value such as 0.0001 or 0.00001). When training embeddings, we only want to regularize the words that appear in the document rather than *all* embeddings at every optimization update step. Specifically, we want to regularize by replacing the standard L2 regularization with $\lambda \left|\left|\frac{\mathbf{x}^\top Y}{||\mathbf{x}||}\right|\right|_2^2$

[1] Variable-length representations will likely be on next coding assignment

[2] Adaptive learning rates are incorporated in training algorithms such as AdaGrad and ADAM.

In this task, you will represent the embedding transformation using the fully_connected functionality. must edit the code of function `FirstLayer`. Here are your sub-tasks:

3 points Replace the ReLu activation with tanh.

4 points Remove the Bias vector.

7 points Replace the L2-regularization of fully_connected with manual regularization. Specifically, tf.losses.add_loss on $R(Y)$, but choose the one in Bonus' Part 1. Unlike the bonus questions, here you will let TensorFlow determine the gradient and update rule. **Hint: tf.losses.add_loss and to the collection** `tf.GraphKeys.REGULARIZATION_LOSSES`.

4 points Preprocess the layer input by passing e.g. through l2_normalize $\mathbf{x}$ as in $\mathbf{x} := \sigma(\mathbf{x})$ where $\sigma(\mathbf{x}) = \frac{\mathbf{x}}{||\mathbf{x}||_2}$.

2 points Add Batch Normalization.

6.5 points **Bonus**: Work-out the analytical expression of the gradient of the regularization $R(Y)$ with respect to the parameters Y. Provide a TensorFlow operator that carries the update by-hand (without using automatic differentiation). The update should act as $Y := Y - \eta \frac{\partial R(Y)}{\partial Y}$, where $\eta \in \mathbb{R}_+$ is the learning rate. Zero credit will be given to all solutions utilizing `tf.gradients()`. However, you are allowed to "test it locally" by comparing your expression with the output of `tf.gradients()`, so long as you dont call the function (in)directly from `Embedding*Update` functions below.

3 points Part i. if $R(Y) = \lambda \left|\left| \frac{\mathbf{x}^\top Y}{||\mathbf{x}||} \right|\right|_2^2$. Implement it in `EmbeddingL2RegularizationUpdate`.

3.5 points Part ii. if $R(Y) = \lambda \left|\left| \frac{\mathbf{x}^\top Y}{||\mathbf{x}||} \right|\right|_1$. Implement it in `EmbeddingL1RegularizationUpdate`.

- **PLEASE** Do not discuss the bonus questions on Piazza, with the TAs, or amongst yourselvs. You must be the sole author of the implementation. However, you are allowed to discuss them after you submit but **with only those who submitted**.
- **Note:** The functions `Embedding*Update` are not invoked in the code. That is okay! Our grading script will invoke them to check for correctness.

The completion of the above tasks should successfully convert the first layer to an embedding layer (Equation 1). You might find the documentation useful: tf.contrib.layers.fully_connected

[25 points] Task 3: Sparse Input Dropout

At this point, after improving tokenization and converting the first layer to be an embedding layer, the model accuracy might have reduced... Do not worry! In fact, the model "train" accuracy have improved at this point: but we do not care about that! We always only care about the model generalization capability i.e. its performance on an unseen test examples, as we do not want it to *over-fit* (i.e. memorize) the training data while simultaneously performing bad on test data.

Thankfully, we have Deep Learning techniques to improve generalization. Specifically, we will be using Dropout.

Dropping-out document terms helps generalization. For example, if a document contains terms "A B C D", then in one training batch the document could look like "A B D", and in another, it could look like "A C D", and so on. This will essentially prevent our 3-layer neural network from "memorizing" how the document looks like, as it appears different every time (there are exponentially many different configurations a document can appear with dropout, and all configurations are equally likely).

The issue is that we cannot use TensorFlow dropout layer out-of-the-box as it is designed for dense Vectors and Matrices. Specifically, if we perform `tf.contrib.layers.dropout` on the input data using:

```
net = tf.contrib.layers.dropout(x, keep_prob=0.5, is_training=is_training)
```

Then, TensorFlow will be dropping half of the entries in `x`. But, this is almost useless because most entries of `x` are already zero (most words do not occur in most documents). We wish to be efficient and drop-out exactly words that appear in the documents rather than entries that are already zero.

Thankfully, we have students to implement sparse-dropout for us! There are many possible ways to implement sparse dropout. Your task is to:

- Please trace the usage of `SparseDropout` and fill its body. It currently reads as:

```
# ** TASK 3
def SparseDropout(slice_x, keep_prob=0.3):
  """Sets random (1 - keep_prob) non-zero elements of slice_x to zero.

  Args:
    slice_x: 2D numpy array (batch_size, vocab_size)

  Returns:
    2D numpy array (batch_size, vocab_size)
  """
  return slice_x
```

Use vectorized implementation with numpy's advanced indexing. Slow solutions (i.e. using for-loops in python) will receive at most 15/25 points.

[10 points] Task 4: Tracking Auto-Created TensorFlow Variables

TensorFlow is arguably the best abstraction for describing a graph of mathematical operations and programming Machine Learning models, but it is not perfect. One weakness of TensorFlow is that it does not provide easy access to variables that are automatically created by the layers (e.g. the fully-connected layer). Often times, one would like to grab a handle on specific variable in a specific layer e.g. to visualize the embeddings, as we will do in the next task.

To do this task, you will find the function `tf.trainable_variables()` helpful. **Hint:** you can print the contents of `tf.trainable_variables()` in `BuildInferenceNetwork`, before and after `FirstLayer`. Your taks is:

10 points Modify the code of `BuildInferenceNetwork`. In it, populate `EMBEDDING_VAR` to be a reference to the tf.Variable that holds the embedding dictionary Y. A snapshot of the code is here for your reference:

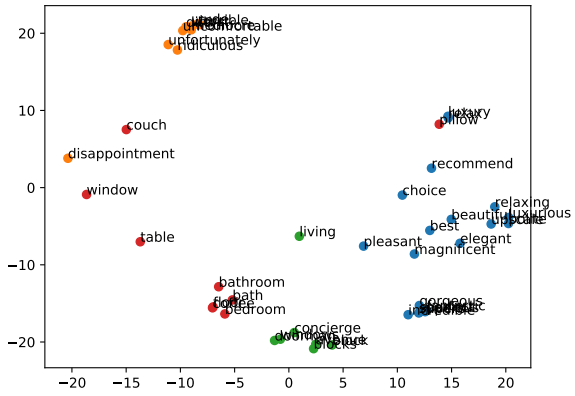
```
def BuildInferenceNetwork(x):
  """From a tensor x, runs the neural network forward to compute outputs.
  This essentially instantiates the network and all its parameters.

  Args:
    x: Tensor of shape (batch_size, vocab size) which contains a sparse matrix
       where each row is a training example and containing counts of words
       in the document that are known by the vocabulary.

  Returns:
    Tensor of shape (batch_size, 2) where the 2-columns represent class
    memberships: one column discriminates between (negative and positive) and
    the other discriminates between (deceptive and truthful).
  """
  global EMBEDDING_VAR
  EMBEDDING_VAR = None  # ** TASK 4: Move and set appropriately.

  ## Build layers starting from input.
  net = x

  # ... continues to construct 'net' layer-by-layer ...
```

Set `EMBEDDING_VAR` to a tf.Variable reference object. Keep first line: '`global EMBEDDING_VAR`'.

[25 points] Task 5: Visualizing the embedding layer

We want to visualize the embeddings learned by our Deep Network. The embedding layer learns Y, a 40-dimensional embedding for each word in the vocabulary. You will project the 40 dimensions onto 2 dimensions using sklearn tsne. Rather than visualizing **all** the words, we will choose 4 kinds of words: Words indicating positive class (shown in blue), negative class (shown in Orange), Words describing furniture (Red) and location (green). Notice that the words that are useful for this classification task occupy different parts of the embedding space: You can easily separate the orange and the Blue with a separating hyperplane. In contrast, words not indicative of the classes (e.g. furniture, location) are not as well clustered[3].

Successfully visualizing the embeddings using t-SNE should like this:



This is a fairly open-ended task, but there should be decent documentation in the `TASK 5` functions that you should implement: `ComputeTSNE` and `VisualizeTSNE`. **Note: you must separately upload the PDF produced by `VisualizeTSNE` onto Vocareum with name `tsne_embeds.pdf`.**

[3] In word2vec, which we will learn soon, the training is unsupervised as the document classes are not known. As a result, all symantically similar words should cluster around one another as there are no classes (not just ones indicative of any classes, as they are not present during training)