Word Embeddings and Recurrent NNs

Chapter 4 – (Introduction to Deep Learning)
Word Embedding for Language Models

- Language model is the probability distribution over all strings.
  - Language translation programs need to identify differences in sentence from one language to the other.
  - Language model helps with the above idea.
- Sentences can be broken into words and probabilities of each word following the previous can be counted.
- E.g. “We live in a small world”.
  - $P(\text{We live in a small world}) = P(\text{We})P(\text{live} \mid \text{We})P(\text{in} \mid \text{We live}) \ldots$
Language Models

\[ P(\text{We live in a small world}) = P(\text{We})P(\text{live} | \text{We})P(\text{in} | \text{We live}) \]

- Each word probability is calculated given all previous words are present in a sentence.
- Not a practical approach, as sentences can be very long.

Bigram language model:
- Probability for each word in a sentence is calculated based on just the previous word.
Word Embeddings

• Given a word in a vocabulary, a probability distribution of all other words following the previous one can be created in a table.

• Using deep network for a word $W_i$, a reasonable probability distribution can be calculated over possible next word.

• As deep network only work with floating numbers, each word can be mapped to a vector of float which is called Word Embeddings.
  • Each embedding is initialized as a vector of e floats.
  • Here, e is the system hyperparameter

• For the number of words are $|V|$ then an array $E$ is initialized to be $|V|$ by e to hold all word embeddings.
Feed Forward Network for Language Model

- Small square represents the input to the network, the integer index of current word $e_i$.
- Output of the network is the probability assignment for possible next word.
- The layer $E$, converts the word index to the embedding and all operations after that point are done on these embeddings.
Cosine similarity

• The cosine similarity of two vectors is a standard measure of how close the vectors are.

• For a two dimensional vectors,
  • if both vectors are pointing in same direction then the cosine similarity would be 1.0.
  • If both are pointing in opposite direction then it would be -1.0.
  • If both vectors are orthogonal then cosine similarity would be 0.

• For arbitrary dimensions, cosine similarity can be calculated using,

\[
\cos(x, y) = \frac{x \cdot y}{(\sqrt{\sum_{i=1}^{n} x_i^2})(\sqrt{\sum_{i=1}^{n} y_i^2})}
\]
Feed Forward Language Model

- First input for the NN is the word index. It is used to get the word embedding E.

```python
inpt=tf.placeholder(tf.int32, shape=[batchSz])
answr=tf.placeholder(tf.int32, shape=[batchSz])
E = tf.Variable(tf.random_normal([vocabSz, embedSz],
                               std_dev = 0.1))
embed = tf.nn.embedding_lookup(E, inpt)
```

- In the above code, inpt points to the word indices.
- And, answr points to the correct similarity index for each word.
- E is the embedding lookup array of the size |V| by e.
- All future operations will be done on embed.
Feed Forward Language Model Loss

• In a language model, each training example is a word probability.
• The loss in a language model is calculated per word.

\[ f(d) = e^{-\frac{x_d}{|d|}} \]

• Here if the corpus d of total words |d| has loss of \( x_d \) then \( f(d) \) represents the perplexity of the corpus d.
• As the training moves forward, the perplexity decreases.
Improving Language Model Efficiency

• Moving from a bigram language model to a trigram language model can help improve the efficiency.

• The previous model used two words (bigram) to create the model. That is each word is assigned probability based on the previous word.

• Trigram model calculates probability based on two previous words rather than one.

• Below is the code to support trigram model along with bigram model,
  
  ```
  embed2 = tf.nn.embedding_lookup(E, inpt2)
  both = tf.concat([embed, embed2], 1)
  ```
Overfitting

- An ideal training data set covers all possible test dataset samples.
  - Improving the model efficiency on test dataset.
- Training samples do not always include all possible examples for test dataset.
  - Hence there is a chance of a good performance on training dataset by the model
  - The same model fails with high loss/perplexity on test dataset.
  - This scenario is classified as overfitting of training dataset.
- Mnist dataset can be characterized as very ideal while PTB contains the combination of handwritten words which has potential for overfitting the training dataset.
Regularization

• Regularization is the modification to fix overfitting.
• Early stopping is a type of regularization.
  • The model stops training when the development perplexity is the lowest.
  • Not the best technique to fix overfitting.
  • Dropout and L2 regularization are much better solutions.
Dropout Regularization

- Here pieces of computation is dropped randomly from one layer of the network to the other.
- Next layer sees more zeros in random locations. This makes training data different for each epoch.
- Classifier cannot depend on the coincidence of a lot of features of the data lining up in a particular way so the generalization is better.
- Preferred method of regularization.

```python
keepP = tf.placeholder(tf.float32)
w1Out = tf.nn.dropout(w1Out, keepP)
```
- KeepP is set to 0.5 for 50% dropout in training phase and 1.0 for testing phase.
L2 Regularization

• In many machine learning overfitting problems are accompanied by model parameters getting too large or too small.
• Seeing the same data again and again contributes to probabilities being overestimated.
• This overestimate is achieved by large absolute weight values.
• L2 Regularization adds a quantity proportional to the sum of squared weights to the loss function.
• Following is added to the loss function in Tensorflow for L2 regularization.

\[ 0.1 \times \text{tf.nn.loss}(W1) \]
Recurrent Networks

- Recurrent Neural Networks are opposite of Feed forward NNs.
- If feed forward NNs are directed acyclic graphs then recurrent neural networks would be directed cyclic graphs.
- A part of the network’s output is feed as its own input.
Recurrent Neural Network – cont’d

• The previous figure can be explained with below equations

\[ S_0 = 0 \]
\[ s_{t+1} = \text{relu}(e_{t+1} \cdot s_t) W_r + b_r \]
\[ o = s_{t+1} W_o + b_o \]

• \( S_0 \) represents state vector and its dimension is a hyperparameter.
• By concatenating the next \( s_t \) and feeding it to linear unit.
• The output is passed through relu activation function.
• Finally output \( o \) is obtained by feeding current state through second linear unit.
• Loss is calculated on \( o \).
RNN

- Recurrent Neural Networks are used when previous input requires to influence arbitrarily far into the future.
- Language models are one of the cases where a word in a sentence can have effects on other word choices.
- Current explanation makes the recurrent neural network approach impractical.
  - Not practical to change all words’ weights and biases for the last word in the corpus in a backward pass.
  - Brute force method can be used to cut off backward pass calculation after certain iterations.
RNN Back propagation and Window size

• The number of iterations used to stop backward pass is called a window size.
  • It is a system hyperparameter.

• Above figure shows back propagation through time with window size set to three.
RNN Batch size and Window size

<table>
<thead>
<tr>
<th>STOP</th>
<th>It</th>
<th>is</th>
<th>a</th>
<th>small</th>
<th>world</th>
</tr>
</thead>
<tbody>
<tr>
<td>but</td>
<td>I</td>
<td>like</td>
<td>it</td>
<td>that</td>
<td>way</td>
</tr>
</tbody>
</table>

- If the corpus is “It is a small world but I like it that way”,
  - The batch size of two divides the sentence in half including STOP padding.
  - Window size 3 divides the batch size so input would be batchSz by window size.
RNN Tensorflow

```python
rnn = tf.contrib.rnn.BasicRNNCell(rnnSz)
initialState = rnn.zero_state(batchSz, tf.float32)
Outputs, nextState = tf.nn.dynamic_rnn(rnn, embeddings, initial_state = initialState)
```

• First line adds recurrent network to the computation.
  • RNN’s weight array is rnnSz.

• The last line calls RNN. It takes in three parameters and outputs two.
  • The first parameter is rnn, the second is words divided in batchSz by
    windowSz and the last is the initial state which comes from the previous run.
  • When the first call to RNN happens, there is no previous state so initial_state
    is set to a dummy value.
The two outputs represent outputs and nextState.

In previously shown RNN figure, outputs are represented as O1, O2 and O3.
  - Outputs has a shape of [batch-size, window-size, hidden-size].
  - The first dimension represents the batch-size of words.
  - The second dimension consists of O1, O2 and O3 for each word.
  - The last is the vector of size rnn-size floats.

nextState consists the last output from the current pass. The next pass will have initial_state set to nextState from the current pass.
RNN Tensorflow – cont’d

• The loss calculation can be done with little modification.
• As we know the output of RNN is a three dimensional array of [batch-size, window-size, hidden-size].
  • The output has to be reshaped for the next layer.

```python
output2 = tf.reshape(output, [batchSz*windowSz, rnnSz])
logits = matmul(output2, W)
```

• The logits can be handed to

```python
tf.nn.sparse_softmax_cross_entropy_with_logits to get a column vector of loss values which then can be passed to tf.reduce_mean to get the loss value. This loss value can be exponentiated to get perplexity.
```
Long Short Term Memory NN

• A type of recurrent neural network which almost always outperforms simple recurrent neural network.
• RNN’s goal is to remember things from far back while simple RNN forgets things quickly.
• LSTM NN’s goal is to improve RNN’s memory of past by training it to remember important things and forget everything else.

`tf.contrib.rnn.LSTMCell(runSz)`

• LSTM takes longer to train than simple RNN.
LSTM RNN Diagram
LSTM RNN Diagram – cont’d

• On the left, we have information coming from previous word with two tensors.

• At bottom left, the next word is coming in and at top right, the information about next word probability and loss is coming out.

• Memories are removed at times units in the diagram and added back at the plus units.

• Current word embedding goes through a layer of linear unit followed by sigmoid activation function.
LSTM RNN Diagram – cont’d

\[ h' = h_t \cdot e \]
\[ f = S(h'W_f + b_f) \]

- Center \( \cdot \) presents the concatenation of vectors. Previous word line \( h_t \) and current word embedding \( e \) are concatenated to create \( h' \) which is fed to forgetting unit to produce \( f \).

- The output of the sigmoid function is multiplied element-wise with memory line \( c \).

- As sigmoid output varies between 0 and 1, the multiplication must reduce the incoming absolute values.
• The next stop is the plus unit where the word embedding has gone through two linear units prior to reaching here.
  • One is sigmoid function and the second is hyperbolic tangent (tanh) function.

\[
a_1 = S(h'W_{a1} + b_{a1})
\]
\[
a_2 = \tanh \left((h_t . e)W_{a2} + b_{a2}\right)
\]

• The result of this is added to the + unit.

\[
c_{t+1} = c'_t + (a_1 \cdot a_2)
\]

• After this, one copy goes out and the other goes to tanh function followed by linear transformation of the more local history to become h line.

\[
\hat{h}'' = h'W_h + b_h
\]
\[
h_{t+1} = \hat{h}'' + a_2
\]