Homework 10

IANNWTF 21/22

Submission until 23 Jan 23:59 via https://forms.gle/n6ERdhYx3uBPzuGn9

Welcome back to the 10th homework for IANNWTF. This week, we will apply our Deep Learning skills on Natural Language Processing. We will work on one of the most important and widespread texts of the world: the bible. You will first create a word embedding. As a bonus task, you afterwards have the chance to unleash your inner priest and teach a NN to come up with new bible extracts
We know, taking the bible is a bit daring, so there is a last more unofficial task to this week: Be sensible about what you do with your results and maybe think about ethics and AI.

1 Data set

We will work with a raw .txt file containing an English version of the bible. You can find the respective file in the folder labelled 'Homework 10' on StudIP.

If you want to work on Goolge Colab, you can upload this 'bible.txt' file to your drive and mount the drive on Colab so you can access it. You can use the following code snippet to do so:

```
# bash code to mount the drive
import os
from google.colab import drive
drive.mount('/content/drive')
os.chdir('drive/MyDrive')
```

You can then go on and read in your file. [1]

2 Word Embeddings

2.1 Preprocessing

You should consider the following points to handle your data:

- Convert to lower case [2], remove new-line characters and special characters.
- Tokenize the string into word-tokens by using one of the word-level tokenizers from tensorflow-text for example this one

- For performance purposes, we advise you to take a small subset of all the words in the bible. We recommend starting using only the 10000 most common ones. [3]

Next, you need to create the input-target pairs, including a word (input) and a context word (target). We suggest a context window of 4 (but if you want to go bigger and have too many computational resources, go for it). Let's take the sentence "The cat climbed the tree" for instance. For the input word "climbed" we want to predict "the, cat, the, tree". The resulting input-target pairs to feed to the network will be (climbed, the), (climbed, cat), (climbed, the), (climbed, tree). Note that you leave out the index 0 when creating the pairs, so there is no pair (climbed, climbed). Create a data set from these pairs and batch and shuffle it. For an **outstanding**, also apply subsampling, i.e. discard words in the text when they appear very often. The probability for keeping a word is $P(w_i) = (\sqrt{\frac{z(w_i)}{s}} + 1) \cdot \frac{s}{z(w_i)}$, where s is a constant that controls how likely it is to keep a word.[4]

2.2 Model

Implement a SkipGram model to create the word embeddings. There are multiple ways of implementing a Skip Gram in tensorflow. We describe one here but feel free to use other implementations. We suggest subclassing from tf.keras.layers.Layer for your model.

- You can overwrite the **build function** and initialize the embedding and score matrices with self.add_weight. Why? The loss function you want to use, tf.nn.nce_loss, gets the weights and biases of the score matrix as input. This is computationally more efficient, since now only the outputs for the target and the sampled words can be computed.
- In the **init function**, where you normally define the layers, you can initialize the vocabulary and embedding size and use these in the build function to create weight matrices of the correct shape.
- In the **call function**, get the embeddings using tf.nn.embedding_lookup().

Instead of calculating the scores, we will directly calculate and return the loss using tf.nn.nce_loss.[5] Note that you do not need to compute the scores in the call function. The loss function does that with the weights and biases and returns the nce loss. Using this loss function, you need to average over the batch manually. For an **outstanding**, sample the negative samples based on word frequency.[6]

2.3 Training

Implement a training loop where input-target-pairs are presented to the network and the loss is calculated. We recommend to start with an embedding size of 64, but feel free to experiment.

To keep track of the training, we want to see which words are close to each other in the embedding space. Therefore, choose some words from the bible corpus that you want to keep track of, e.g. *holy, father, wine, poison, love, strong, day* etc. Calculate and print their nearest neighbours according to the (cosine similarity) of each epoch. The cosine similarity is the inner product of the normed vectors. You can either implement it yourself or use one of many functions provided by different python libraries. For the calculation of the **nearest neighbours**, you need the following steps:

1. Calculate the cosine similarities between the whole embedding and the embedding of the words you want to investigate
2. For each selected word, sort the neighbours by their distance and return the k-nearest ones.

3 Bonus: Text Generation

This task is definitely a Bonus, so don't stress about it if your schedule is full. However, considering it might make it even easier to solve next week's homework on attention (but that will be just doable without this bonus task - so again, do not worry about it if you have no capacities right now). If you however have the time, go ahead! This is not necessary for an outstanding, but it can get you an *additional bonus point*. Yay! Just drop a hint if you solved this task successfully.

3.1 Preprocessing

For the text generation, we will work with single characters (or sub-words if you want better results - see the tokenization section on courseware) instead of words as tokens. (Think about why this might be beneficial.) We need to again extract all relevant characters from the text and assign them to IDs. [7]

To generate text, we will give the network a subsequence, e.g. of length k = 20 and have it predict the next character. We then compare the predicted char to the actual next character and compute the loss for each timestep. Example (for character tokenization): Input sequence: "First Citizen: Befor" → Target sequence: "irst Citizen: Before". Or for sub-word tokenization: Input sequence: "First Citizen: Be" → Target sequence: " Citizen: Before".

To create these pairs of sequence, we chunk the dataset into subsequences of length k+1. You can use .batch() for this. And make sure that all subsequences in the resulting dataset have length k+1. [8] Once you have sequences of length k+1, split them into input and target sequences like in the example above and map them together to a data set, which you can then shuffle and batch again.

3.2 Model

For the text generation task, build an RNN model containing an RNN layer with 1 or more RNN cells and a Dense readout layer[9]. Note that you will need an embedding layer to transform your token-indices based input into a vector. [10] You might want to build your own RNN_Cell class[11], otherwise you can simply use the inbuilt SimpleRNNCell from keras.[12]

3.3 Generating Text

Write a function to generate text. It should take in a phrase to be continued that has the same length as the sequences you trained on (e.g. 20) and how long the sequence to be generated should be. Then translate the sample string into a tensor of shape (1,seq_length,vocab_size). Feed the sample sequence into the RNN and get the probabilities of next character. Sample the index for the new character/sub-word,[13] translate to the actual character/sub-word and add it to the sample string. Repeat this for the desired sequence length, by deleting the first character/sub-word of the old sequence and adding the new character/sub-word to create a sequence of length k again.

4 Outstanding Requirements

As always, the usual rule applies: Design your implementation such that it is readable and usable for others. You may refer to past instructions in the 'How to Outstanding' sections of the homework instructions, or take past sample solutions as guidance.

Also, for an outstanding, make use of subsampling and sample negative samples based on word-frequency for learning the embeddings.

To achieve *even another additional bonus point* (think about it as a double outstanding) solve the text generation task - make use of sub-word tokenization if you really want to flex.

But most importantly - have fun!

Notes

[1] you can use python's inbuilt open() and read() methods.

[2] You can just call .lower() on the full string

[3] You can scale up later if you know everything works to get more interesting results.

[4] Usually s=0.001 is used

[5] We did not talk about NCE(Noise Contrastive Estimation in the courseware. It is basically an improvement upon negative sampling. The core idea of only updating the weights for a few negative words is kept

[6] Have a look at tf.random.fixed_unigram_candidate_sampler

[7] Unlike for the word embeddings you should also keep punctuation if you want your personal ANN-written bible to contain punctuation.

[8] (understand the parameter 'drop_remainder' in .batch())

[9] Output layer with softmax activation. Make sure you have return_sequences=True in the RNN layer

[10] You may want to check out Tensorflow's Embedding Layer implementation.

[11] Define the weights w_in, w_h and b_h inside the build function (more explanation in the embedding part), in the call function define the pass through the RNN cell which outputs the new hidden state.

[12] Orientations for the state size can be something around 128, 256.

[13] use tf.random.categorical()