Homework 10

IANNWTF 20/21

Submission until **27 Jan** 23:59 via https://forms.gle/H31ckx5251Qg3Ndm6

Welcome back to the 10th homework for IANNWTF. This week we will apply our Deep Learning skills on Natural Language Processing. Working with texts from Shakespeare, you will first create a word embedding. Then afterwards you have the chance to unleash your inner poet and teach a NN to write novel Shakespearean verses. This time you will have 2 weeks time to submit, so until Jan 27th.

We are coming closer to the end of this course and after this homework, there will only be one more homework. As a reminder, you need 9 homeworks (N-2) to pass and be admitted to the exam. So if you already completed all 9 homeworks with a Done or Outstanding, you technically don't need to submit this homework, but we nonetheless recommend it, because it's fun, a good recap of the content in preparation for the exam and a chance to snatch another bonus point.

1 Data set

We will work with the 'tiny_shakespeare' dataset from tensorflow_datasets.. It contains 40.000 lines of Shakespeare's texts. Load the data set.[1] In the hints we have some additional tips for processing the data set #spoilerwarning [2]

2 Word Embeddings

2.1 Preprocessing

In order to analyse the words from the Shakespeare text and create word embeddings from it, we need to translate the words to IDs. E.g. 'castle' → 1450, 'wine' → 8630. So assign an ID to every word (or more practically, to the 10.000 most common words). Keep in mind to also transform the text to lowercase and remove punctuation and new-line-characters. Then create two dictionaries from this assignment where you can translate a) the words to ids and b) translate back the ids into words. [3] [4]

Then 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.[5]

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.[6] 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.[7]

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 Shakespeare corpus that you want to keep track of, e.g. *queen, throne, wine, poison, love, strong, day* ect. Calculate and print their nearest neighbours according to the (cosine similarity) 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 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 Text Generation

3.1 Preprocessing

For the text generation, we will work with single characters 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. [8]

Create 2 dictionaries, one having the words as keys and IDs as values and one the other way around. Then convert the whole text into IDs. 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: Input sequence: "First Citizen: Befor" → Target sequence: "irst Citizen: Before"

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: Input sequence: "First Citizen: Befor" → Target sequence: "irst 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. [9] 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[10]. For an **outstanding**, 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,[13] translate to the actual character and add it to the sample string. Repeat this for the desired sequence length, by deleting the first character of the old sequence and adding the new character to create a sequence of length k again.

Additional bonus points for your theatrical performance of your generated Shakespeare play. :)

Notes

[1]Load via tfds.load(). The 'train' partition will be sufficient.

[2] The data set comes as a Features Dict and we are only interested in the text, which is marked as 'text'. Extract the text from the dictionary over the key 'text'. Inside the dictionary, the text comes as a string tensor, which can be quite annoying to work with. Our advise it to read out the whole text from the data set and save it as a string. '.numpy()' combined with an iteration will do the trick, but there might also be many other ways.

[3]So one dictionary that has the ids as its key and words as values and one the other way around.

[4]For handling out of vocabulary words u can assign the key 0 to unknown words. For accessing the words2ids with an unknown word you can use dictionary.get('word', default value), which will return the default value for non-existing keys

[5]Usually s=0.001 is used

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

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

[8]Unlike for the word embeddings you should also keep punctuation if you want your personal NN-written Shakespeare drama to contain punctuation. This time you don't need a tokenizer to do it, check out python sets if in doubt.

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

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

[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()