Text_retrieval_with_recurrent_neural_methods_handout

December 8, 2023

1 LSTM in practice – NLP

1.1 Language modeling

A language model is a probability distribution over the sequence of words, modeling language (production), thus if the set of words is w, then for arbitrary $\mathbf{w} = \langle w_1, \dots, w_n \rangle$ $(w_i \in W)$ sequence it defines a $P(\mathbf{w})$ probability.

Probability with chain rule:

$$P(\mathbf{w}) = P(w_1) \cdot P(w_2|w_1) \cdot P(w_3|w_1, w_2) \cdot \dots \cdot P(w_n|w_1, \dots, w_{n-1})$$

so this means, that for the modeling we need only to give the conditional probability of the "continuation", the next word, thus for w word and $\langle w_1, \dots, w_n \rangle$ sequence the probability that the next word will be w

$$P(w \mid w_1, \dots, w_n)$$

There are character based models also, which take the individual characters as units, not the words, and model language as a distribution over sequences of characters (think T9…)

1.1.1 Measurement of performance: Perplexity

A language model $\mathcal{M}$'s perplexity over the word series $\mathbf{w} = \langle w_1, \dots, w_n \rangle$ is:

$$\mathbf{PP}_{\mathcal{M}}(\mathbf{w}) = \sqrt[n]{\frac{1}{P_{\mathcal{M}}(\mathbf{w})}}$$

With the chain rule can be rewritten as:

$$\mathbf{PP}_{\mathcal{M}}(\mathbf{w}) = \sqrt[n]{\frac{1}{P_{\mathcal{M}}(w_1)} \cdot \frac{1}{P_{\mathcal{M}}(w_2|w_1)} \cdot \frac{1}{P_{\mathcal{M}}(w_3|w_1, w_2)} \cdot \dots \cdot \frac{1}{P_{\mathcal{M}}(w_n|w_1, \dots, w_{n-1})}}$$

which is exactly the geometric mean of the reciprocals of the conditional probabilities of all words in the corpus.

In case of a bigram model this is further simplified to:

$$\mathbf{PP}_{\mathcal{M}}(\mathbf{w}) = \sqrt[n]{\frac{1}{P_{\mathcal{M}}(w_1)} \cdot \frac{1}{P_{\mathcal{M}}(w_2|w_1)} \cdot \frac{1}{P_{\mathcal{M}}(w_3|w_2)} \cdot \dots \cdot \frac{1}{P_{\mathcal{M}}(w_n|w_{n-1})}}$$

1.1.2 But what is it good for?

For example: - Predictive text input ("autocomplete") - Generating text - Spell checking - Language understanding - And most importantly representation learning - this we will be studiying in detail in a next lecture

1.1.3 Generating text with a language model

The language model produces a tree with probable continuations of the text:

Using this tree we can try different algorithms to search for the best "continuations". A full breadth-first search oi usually impossible, due to the high branching factor of the tree.

Alternatives: - "Greedy": we choose the continuation which has the highest direct probability, This will most probably be suboptimal, since the probability of the full sequence is tha product of the continuations, and if we would have chosen a different path, we might ahve been able to choose later words with hihg probabilities. - Beam-search: we always store a fixed k number of partial sequences, and we always try to expand these, always keeping the most probable k from the possible continuations.

Example (k=5):

1.1.4 The "old way": N-gram based solutions

With *gross* simplification we assume, that the distribution is only dependent on the prior $n-1$ words (where n is typically <= 4), thus we assume a Markov chain of the order n:

$$P(w \mid w_1, \dots, w_k) = P(w \mid w_{k-n+2}, \dots, w_k)$$

We simply compute these probabilities in a frequentist style by calculating the n-gram statistics of the corpus at hand:

$$P(w_2 \mid w_1) = \frac{c(\langle w_1, w_2 \rangle)}{c(w_1)}$$

$$P(w_{k+1} | \ w_1, \dots, w_k)_= \frac{c(\langle w_1, \dots, w_k, w_{k+1} \rangle)}{c(\langle w_1, \dots w_k \rangle)}$$

Please note, that in this case we are using "memorization", a form of database learning, with minimal compression - "counting".

But what do we do the given n-grams rarely or never occur? We have to employ some **smoothing** solutions, like:

Additive smoothing We pretend that we have seen the n-grams more times than we have actually did with a fixed δ number, in the simplest case with $n = 2$:

$$P(w_2 \mid w_1) = \frac{c(\langle w_1, w_2 \rangle) + \delta}{\sum_{w \in V}[c(\langle w_1, w \rangle) + \delta]}$$

Widespread solution for δ is 1.

The main problem with this kind of smoothing is that it does not take into account by "supplementing" the data the frequency of components of shorter n-grams, eg. if neither $\langle w_1, w_2 \rangle$ nor $\langle w_1, w_3 \rangle$ occurs in the corpus, it assumes the frequency of both bigrams to be δ, irrespective of the ratio of frequencies of w_2 and w_3. Most smoothing techniques are trying to accomodate this, eg: simple interpolation:

Interpolatcion In case of bigrams, we add - with a certain weight - the probabilities coming from the individual frequencies:

$$P(w_2 \mid w_1)_{\text{interp}} = \lambda_1 \frac{c(\langle w_1, w_2 \rangle)}{c(w_1)} + (1 - \lambda_1) \frac{c(w_1)}{\sum_{w \in V} c(w)}$$

Recursive solution for arbitrary k:

$$P(w_{k+1} | \, w_1, \dots, w_k)_{\text{interp}} = \lambda_k \frac{c(\langle w_1, \dots, w_k, w_{k+1} \rangle)}{c(\langle w_1, \dots w_k \rangle)} + (1 - \lambda_k) P_{\text{interp}}(\langle w_2, \dots, w_{k+1} \rangle)$$

λ_k is empirically set by examining the corpus, typically by Expectation Maximization algorithm, which - as we have mentioned - iteratively tunes the parameters to maximize the likelihood.

Good overview about the smoothing methods: MacCartney, NLP Lunch Tutorial: Smoothing

General problems

- Even the core assumption is not too realistic, since the probabilities are for sure influenced in a way by words further than n, but for practical reasons, it has to be limited (sparsity, computation capacity).
- On a large enough corpus, the memory footprint of the n-gram models is *huge*, eg. for the 1T n-gram corpus of Google (see here) containing 1,024,908,267,229 tokens the n-gram counts are as follows:
  - unigram: 13,588,391,
  - bigram: 314,843,401,
  - trigram: 977,069,902,
  - fourgrams: 1,313,818,354
  - fivegram: 1,176,470,663.

1.2 Language modeling with LSTMs

One way to circumvent the Markov assumption is to use RNN-s, which are capable of modeling the long-ter dependencies inside the sequence of words. The text is thus considered to be a time-series, and thus an appropriate architecture can be used (as we have already seen):

Notable features:

- Input is a "one-hot" encoded vector, wchic we on the spot transform into an "embedding vector"
- For each output step, we get a probability distribution over the whole vocabulary with softmax
- This above is a simple RNN, but LSTMs can be used without any problems

1.2.1 Teaching

In theory an RNN could be trained with full GD on the corpus in one go:

- The loss is generally the well-kown crossentropy, which is in this case (since the input is a one-hot vector):

  $$J^{(i)}(\Theta) = -\log(\hat{y}[x^{(i+1)}])$$

  the negative logarithm of the probability assigned by the network to the right word / next word.

- For the sake of more frequent updates, and since BPTT for long sequences is very expensive, teaching is done in smaller units with not necessarily the same length.

- The unit is typically one or more sentece, or if the length allows, and we have enough material, a paragraph can be a good candidate.

- Initial state in case of the time-series units: if the boundaries are inside a unit of text, it is important to *transfer the hidden state* from the previous unit, in other cases initialization can be done by some fixed value.

- (Somewhat misleading) terminology: the length of the "time" unit is *time step*, but sometimes certain implementations call it *minibatch*, though that would generally mean the number of units processed in one go for the sake of computaitonal efficiency.

1.2.2 LSTM as layers

- An LSTM - how ever strange that may sound - can be considered to be a complete layer. The most important parameter of it is the "number of (memory) units", which is the length of the hidden state vector, thus, the memory capacity. **Warning: this does not have any relationship to input size, thus can be considered a freely chosen parameter.**
- It is quite widespread to use multiple LSTM layers ("stacked LSTMs") – as in the case of ConvNets the hope is, that the layers learn a hierarchy of abstract representations:

(on the right side a network is shown with skip/residual connections!)

In this case it makes sense, that we do not only get on top of the LSTM a final prediction h (or even prediction + inner state vector c) for a sequence, but **we ask it to output the whole sequence of predictions**, so that the next layer can also operate on full sequences. Please bear this in mind during implementation, since this can be a common source of failure.

1.3 An LSTM language model in Keras

For this task the inspiration comes from the famous reference work of Andrej Karpathy.

Note, that in this case we will not use regularization, since we are willing to overfit - for the sake of play with the text. This is now an "overfitting competition", so *not* a generally good practice!

1.4 Reader

[1]:
```
import numpy as np
import tensorflow as tf
import nltk

from numpy.random import seed
seed(1212)

tf.random.set_seed(1234)

nltk.download("brown")

from nltk.corpus import brown

# This can be an important parameter, so be aware of it...
max_seq_length = 15
max_num_of_sents = 57200
# max_num_of_sents = 50 # How many sentences should we read from the corpus␣
  ↪(max=57200)

def generate_brown_word_to_id_map():
    """Return a dictionary mapping downcased Brown-words to their ids.
    Numbering starts from 1 since we use 0 for masking (!!!).
    """
    words = set()
    for word in brown.words():
        words.add(word.lower())
    return {word: idx + 1 for idx, word in enumerate(sorted(words))}

class BrownReader:
    """A reader class for the Brown corpus.
    """

    def __init__(self):
        self.word_to_id_map = generate_brown_word_to_id_map()
        self.id_to_word_map = {idx: word for word, idx in self.word_to_id_map.
  ↪items()}

    def n_words(self):
        return len(self.word_to_id_map)

    def sentence_to_ids(self, sentence):
        """Return the word ids of a sentence.
        """
        return [self.word_to_id_map[word.lower()] for word in sentence]
    def sentences(self):
        """Generator yielding features from the Brown corpus.
        """
        return (self.sentence_to_ids(sentence) for sentence in brown.sents())

    def sentence_matrixes(self):
        x = np.zeros((max_num_of_sents, max_seq_length-1))
        y = np.zeros((max_num_of_sents, max_seq_length-1))
        sents = self.sentences()
        for idx, sent in enumerate(sents):
            if idx == max_num_of_sents:
                break
            np_array = np.asarray(sent)
            length  = min(max_seq_length, len(np_array))
            x[idx, :length - 1] = np_array[:length - 1]
            y[idx, :length - 1] = np_array[1:length]
        return x, y
```

```
[nltk_data] Downloading package brown to
[nltk_data]     /Users/nilsmart96/nltk_data…
[nltk_data]   Package brown is already up-to-date!
```

1.5 Model

1.5.1 Parameters

[2]:
```
br = BrownReader()
n_words = br.n_words()

max_input_length = max_seq_length - 1 # since our x/y input does not contain␣
 ↪the last/first element of the sentences
```

[3]:
```
data_x, data_y = br.sentence_matrixes()
```

[4]:
```
data_y = np.expand_dims(data_y, -1) # It seems that Keras needs this for the␣
 ↪"one-cold" and softmax dims to match
```

2 Tasks

See below

[5]:
```
# Network parameters
lstm_size = 512
embedding_size = 100
```

2.0.1 Network

[6]:
```
# Import:
# Import the appropriate layers from tf.keras!
# Think about it, that one layer should map ("embed") tha input into a dense␣
 ↪vector!
# Don't forget the main model class - according to functional or sequential API
# And eventually the optimizer and backend
# Later one for resetting the graph - good practice!
# And think about it what loss function you will use. The output is a␣
 ↪classification
# (categorical) task, it is sparse, so...
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, LSTM, Dense
from tensorflow.keras import backend as K

# Please reset the graph here!
K.clear_session()

# Model
########
# Build the model!
# Start with the input layer, it receives a vector of the length of the maximal␣
 ↪text span (sentence)
# And: vector or not, shape is a tuple...
# After this, use the "mapping" layer.
# WARNING:
# 1. width = number of words +1
# 2. it's size is defined in a parameter, somewhere above
# 3. length of input: max input size -1
# 4. Zero values are to be masked in it!!!
#    the constructor has a named argument which has to be given with the True␣
 ↪value!!!
#
# TASK: Can you please verbally elaborate to the instructor, why the points␣
 ↪above are true?
# See markdown cell below.

# Input layer
inputs = Input(shape=(max_input_length,))

# Embedding layer
embedding_layer = Embedding(input_dim=n_words + 1,
                            output_dim=embedding_size,
                            input_length=max_input_length - 1,
                            mask_zero=True)(inputs)
```

```
# In the next TWO layers there should be LSTM-s
# For them being able to be stacked, they have to give back not only the␣
 ↪predictions at sequence end
# Somewhere there has to be a nice parameter for this... ;-)

# LSTM layers
lstm_1 = LSTM(units=lstm_size, return_sequences=True)(embedding_layer)
lstm_2 = LSTM(units=lstm_size, return_sequences=True, return_state=True)(lstm_1)
lstm_output = lstm_2[0]

# Important: for certain purposes (hint: search engine...) it's very useful to␣
 ↪have the last
# hidden state and cell state also included in the results, so for the 2nd LSTM␣
 ↪please turn
# on the return_state option. Be aware that thereby the 2nd LSTM cell will␣
 ↪return three
# tensors (the series of outputs, last hidden state, last cell state), and you␣
 ↪will need only
# the first of these as input for your next layer (as always, ask when in␣
 ↪doubt)

# Finally project the output with a fully connected layer and a softmax.
# What is it's width? (Help: If you have varbally elaborated well above, you␣
 ↪already know.)
# width = number of words + 1
output = Dense(units=n_words + 1, activation='softmax')(lstm_output)

# Finally, create a model instance!
model = Model(inputs=inputs, outputs=output)

model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 14)]              0

 embedding (Embedding)       (None, 14, 100)           4981600

 lstm (LSTM)                 (None, 14, 512)           1255424

 lstm_1 (LSTM)               [(None, 14, 512),         2099200
                              (None, 512),
                              (None, 512)]

 dense (Dense)               (None, 14, 49816)         25555608
```

```
=================================================================
Total params: 33891832 (129.29 MB)
Trainable params: 33891832 (129.29 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

2.0.2 TASK: Can you please verbally elaborate to the instructor, why the points above are true?

Certainly, let's address each point individually:

1. **Width = number of words + 1:**
   - The width of the embedding layer, which is essentially the number of neurons or nodes in that layer, is set to the total number of unique words in the vocabulary plus 1. This is because an additional index is needed to account for possible out-of-vocabulary words. These may appear during prediction, but were not present in the training vocabulary. By having this additional index, the model can handle unseen words by assigning them this special out-of-vocabulary index.
2. **Its size is defined in a parameter, somewhere above:**
   - It is defined in the parameter `n_words`, which captures the number of unique words in the brown dataset.
3. **Length of input: max input size - 1:**
   - The length of the input is set to the maximum input size minus 1. This is because the model is designed to predict the next word in a sequence, and for this task, the input consists of a sequence of words, with the last word being the target word to predict. To exclude the target from the input sequence, we have to set the input length to `max_input_length - 1`.
4. **Zero values are to be masked in it:**
   - The `Embedding` layer has a parameter called `mask_zero`, which is set to `True`. This parameter is crucial when working with sequences of varying lengths, as is the case with the sentences in todays assignment. It indicates that the model should consider zero values in the input as a special "mask" value and not process them further. This is important to ensure that the model does not treat padding (zero) values as meaningful input.

2.0.3 Error, optimizer, compilation

[7]:
```
# Loss
loss = "sparse_categorical_crossentropy" # Our output is One-hot encoded. What␣
↪do we use?

# Optimizer
optimizer = 'adam' # According to taste...

# Compilation
#############
```

```
model.compile(optimizer=optimizer, loss=loss, metrics=['accuracy'])
```

2.0.4 Training

We generate the trainig data.

[8]:
```
data_y = np.expand_dims(data_y, -1) # It seems that Keras needs this for the␣
 ↪"one-cold" and softmax dims to match
```

And train!

[9]:
```
# Fit a language model to the data!
# Use 10% validation - not so important in case of language models.
# Use default alidation split of Keras.
# And try to guess a realistic batch size!
batch_size = 64
epochs = 10
history = model.fit(data_x, data_y, batch_size=batch_size, epochs=epochs,␣
 ↪validation_split=0.1)
```

```
Epoch 1/10
805/805 [==============================] - 468s 578ms/step - loss: 7.7125 -
accuracy: 0.0708 - val_loss: 7.1911 - val_accuracy: 0.0863
Epoch 2/10
805/805 [==============================] - 477s 592ms/step - loss: 7.0754 -
accuracy: 0.0982 - val_loss: 6.6721 - val_accuracy: 0.0993
Epoch 3/10
805/805 [==============================] - 478s 594ms/step - loss: 6.5240 -
accuracy: 0.1208 - val_loss: 6.2215 - val_accuracy: 0.1215
Epoch 4/10
805/805 [==============================] - 476s 591ms/step - loss: 6.1500 -
accuracy: 0.1438 - val_loss: 6.0365 - val_accuracy: 0.1432
Epoch 5/10
805/805 [==============================] - 462s 574ms/step - loss: 5.8829 -
accuracy: 0.1605 - val_loss: 5.9423 - val_accuracy: 0.1552
Epoch 6/10
805/805 [==============================] - 467s 580ms/step - loss: 5.6512 -
accuracy: 0.1742 - val_loss: 5.8851 - val_accuracy: 0.1585
Epoch 7/10
805/805 [==============================] - 470s 584ms/step - loss: 5.4366 -
accuracy: 0.1843 - val_loss: 5.8901 - val_accuracy: 0.1632
Epoch 8/10
805/805 [==============================] - 467s 581ms/step - loss: 5.2276 -
accuracy: 0.1928 - val_loss: 5.8988 - val_accuracy: 0.1652
Epoch 9/10
805/805 [==============================] - 478s 594ms/step - loss: 5.0215 -
accuracy: 0.2009 - val_loss: 5.9254 - val_accuracy: 0.1699
Epoch 10/10
```

```
805/805 [==============================] - 474s 589ms/step - loss: 4.8184 -
accuracy: 0.2094 - val_loss: 5.9675 - val_accuracy: 0.1676
```

2.1 Demo 1: Predict next word

[10]:
```
# Prediction
############

def str_to_input(s):
    """Convert a string to appropriate model input.
    """
    words = [x.lower() for x in s.split()[:max_input_length]]
    ids = [br.word_to_id_map[word] for word in words]
    ids_array = np.asarray(ids)
    length = min(max_input_length, len(ids_array))
    result = np.zeros((1, max_input_length))
    result[0, :length] = ids_array[:length]
    return result, length


while True:
    s = input("\nEnter a few starting words of a sentence or <return> to stop:␣
 ↪")
    if s == "":
        break
    else:
        try:
            x, length = str_to_input(s)
            predictions = model.predict(x)
            probs = predictions[0][length - 1]
            most_probable = np.argmax(probs)
            print("Predicted next word:", br.id_to_word_map[most_probable])
        except KeyError:
            print("Unknown words -- please try again!")
```

```
1/1 [==============================] - 1s 907ms/step
Predicted next word: to
1/1 [==============================] - 0s 25ms/step
Predicted next word: the
1/1 [==============================] - 0s 25ms/step
Predicted next word: the
1/1 [==============================] - 0s 25ms/step
Predicted next word: not
1/1 [==============================] - 0s 25ms/step
Predicted next word: a
1/1 [==============================] - 0s 26ms/step
Predicted next word: good
```

2.2 Demo 2: Similarity of sentences

First we define a function that generates the hidden state of the LSTM from an input sentence:

[11]:
```
input_layer = model.get_layer("input_1")
lstm_2_layer = model.get_layer("lstm_1")

cell_state_fun = K.function([input_layer.input],[lstm_2_layer.output[2]])

def get_embedding(x):
    """Return the final cell state associated with the input.
       Returns the last cell state as a vector.
    """
    return cell_state_fun([x])[0].flatten()
```

Then we use the vectors for calculating the cosine distance between sentences.

[12]:
```
def cos_sim(a, b):
        return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

while True:
    s1 = input("\nEnter the first sentence or <return> to quit: ")
    if s1 == "": break
    s2 = input("\nEnter the second sentence: ")
    try:
        x1, _ = str_to_input(s1)
        x2, _ = str_to_input(s2)
        e1 = get_embedding(x1)
        e2 = get_embedding(x2)
        print("The cosine similarity between the two sentences is", cos_sim(e1,␣
 ↪e2))
    except KeyError:
        print("Unknown words -- please try again!")
```

```
The cosine similarity between the two sentences is 0.981803
The cosine similarity between the two sentences is 0.91222143
```

2.3 Demo 3: Mini search engine

We use the library Annoy published by Spotify to create a vector space index of the Brown corpus from the LSTM's cell state. We assign a vector for each sentence, and then store it to be able to run nearest neighbor queries on it. With this we effectively created a **semantic search engine**.

(There are multiple solutions for approximate nearest neighbor search a scale which are worth looking into, one of them is FAISS from Facebook Research.)

[13]:
```
def brown_sent_to_input(ids):
  ids_array = np.asarray(ids)
  length = min(max_input_length, len(ids_array))
  result = np.zeros((1, max_input_length))
  result[0, :length] = ids_array[:length]
  return result, length
```

[14]:
```
sentlist = list(br.sentences())
```

[15]:
```
!pip install annoy
```

[16]:
```
INDEX_COVERAGE_PERCENT = 1.0 #How much of the corpus you want ot index? 1.0␣
 ↪means whole, 0.5 means half.
NEAREST_NEIGHBOR_NUM = 5
```

[17]:
```
from annoy import AnnoyIndex
from tqdm import tqdm

index = AnnoyIndex(lstm_size, metric="angular") # 512 hard-coded originally

for i in tqdm(range(int(len(sentlist)*INDEX_COVERAGE_PERCENT))):
  inputs,length = brown_sent_to_input(sentlist[i])
  vector = get_embedding(inputs)
  index.add_item(i,vector)

print("Building index...")
index.build(100)
print("Index done, ready to query!")
```

```
  0%|          | 0/57340 [00:00<?, ?it/s]100%|      | 57340/57340
[31:06<00:00, 30.72it/s]
```

```
Building index…
Index done, ready to query!
```

[18]:
```
def print_brown_index(sentences, indices):
  for i in indices:
    word_ids_list = sentences[i]
    for j in word_ids_list:
      print(br.id_to_word_map[j]+" ", end='')
    print()
```

[19]:
```
while True:
  query = input("\nEnter the query or <return> to quit: ")
  if query == "": break
  try:
    in_ids, length = str_to_input(query)
    in_vector = get_embedding(in_ids)
    nearest_sentence_indices = index.get_nns_by_vector(in_vector,␣
 ↪NEAREST_NEIGHBOR_NUM)
    #print("nearest indices:", nearest_sentence_indices)
    print_brown_index(sentlist, nearest_sentence_indices)
```

```python
except KeyError:
  print("Unknown words -- please try again!")
```

```
there are two reasons for this .
there are two reasons for this .
there shall be covered into the treasury to the credit of the proper special
fund all funds hereinafter specified .
it is not within the scope of this report to elaborate in any great detail upon
special districts in rhode island .
we can expect more of the same .
there are two reasons for this .
there are two reasons for this .
there is scarcely a scar showing .
we can expect more of the same .
there need be no squeamishness about admitting this .
i am not a philosopher .
i have not the heart .
i find this view amazing .
it was not a pet .
i think it's a good deal .
```