Recitation 05 – KI430

AUTHOR
Christian Osendorfer

AFFILIATION
Hochschule Landshut

Written Questions

Neural Networks for Named Entity Recognition

We aim to build a feedforward neural network to perform *Named Entity Recognition (NER)*, a common task in natural language processing. NER is a multiclass classification problem in which each word in a sentence is assigned a label from a predefined set of entity types. In our case, entities are *name*, *place*, *date*, *company* or *event*. For example, the sentence *franz celebrates his birthday in june* has *franz* as *name*, *birthday* as *event* and *june* as *date*.

Our neural network architecture includes a single hidden layer and and it classifies each word in a given sentence using a sliding window of size 5: For a given sentence $w_1, w_2, \dots, w_T$, your network classifies w_t when the sentence fragment $w_{t-2}, w_{t-1}, w_t, w_{t+1}, w_{t+2}$ is contained in that window. The vocabulary (of full words, not generic tokens) V contains $|V|$-many english words.

Demonstrate the working of that generic architecture using the example sentence *sissi and franz marry in june*. Are there any cases that you need to handle in a special way?

- How does your architecture work with words at the beginning/end of a sentence? Add additional tokens to the vocabulary, if helpful.
- All elements in the vocabulary should be represented by 300-dimensional embeddings. What form in terms of dimensions does the input to your network have, when the word *franz* is classified? Use the expression $e(\cdot)$ in order to describe word embeddings, e.g. $e(\text{franz})$ to describe the vector representation of the word *franz*.
- The hidden layer of the neural network is 500 dimensional. Describe the network architecture in terms of all learnable parameters and their exact sizes/shapes. Also describe the cost function for a sentence with n words. Your architecture should be structured in such a way that the **ordering of the input words** in a window matters for a correct use of the network.
- To reduce the number of learnable parameters, the input representation of all relevant word embeddings is processed as follows:
  - The center word in a window is processed with its *own weight matrix*.
  - The remaining words (in this setting: 4 words) are processed with the same matrix. In order to avoid that positional information of a word (*where* in the window is the word) is lost, use *learnable vectors with matching dimensions*, which are encoding position information. Describe two alternatives (with all relevant parameters and their sizes) which either **add** or **concatenate** these *positional embeddings* to word embeddings.

RNNs and Grammars

The Bounded Parenthesis Language (or Dyck Language, a context free grammar), consists of strings of '(' and ')' characters. A string is a valid word of this language if (i) the number of '(' characters equals the number of ')' characters **and** (ii) if *every* prefix of the string contains at least as many '(' characters as ')' characters. Examples for valid strings are "(()())" and "(()((()))())" while strings "(()()))" and "(()((())" are not. Note that the latter is a valid prefix!

- Define a recurrent Neural Network that classifies sequences of parenthesis accordingly to their membership in the Bounded Parenthesis Language.

  That is, at the end of a sequence, a classification function needs to implement a binary decision.

- Also determine a classification function that correctly decides in each time step if the prefix constraint is fulfilled.

Provide all the parameters and functions that make up your RNN. You can use scalar values to encode the input characters.

> **Note**
>
> Formal languages, often classified using frameworks like the Chomsky hierarchy, are a core interest in theoretical computer science. During the 1990s, when Recurrent Neural Networks (RNNs) were researched the first time by a wider audience, their theoretical ability to represent such languages was an important question. A well-known example explored in this context is the *embedded Reber grammar*, see e.g. the origianl LSTM paper

Implementation: Machine Translation with RNN and Attention

In this lab, you will implement a basic machine translation system using recurrent neural networks (RNNs) and attention mechanisms. While RNN-based models are no longer state-of-the-art for translation, they remain a fundamental architecture for understanding sequence-to-sequence modeling and attention.

More specifically, you will build an encoder-decoder architecture with attention, train it on a small parallel English–French dataset, and evaluate both its outputs and attention behavior.

> **Important**
>
> This is a substantial programming task, so it's important to get started as soon as possible, ideally, right away. Begin by reading through the entire assignment to get a sense of the scope and requirements.
>
> Exploring the dataset, inspecting intermediate results of your implementation and visualizing final results is probably best done in a notebook, so it is fine to submit a notebook that demonstrates your work, using the implemented code to be found in modules. Please make a remark in your README.md regarding this notebook, if necessary.

Platform & Setup

Begin development on your local machine using the CPU to prototype your code, debug, and iterate on ideas. For full model training, you are strongly encouraged to use Google Colab with GPU support. While it is possible to train on a CPU, this will be significantly slower. Structure your code to support both CPU and GPU with minimal changes required by the user.

You will work with a small English–French dataset, provided as a CSV file, `processed_en_fr.csv`. The file contains aligned sentence pairs in English and French. Before doing any coding with respect to the model, inspect the CSV file, look at random pairs and see if a professional translation service matches the provided data.

Dataset and Preprocessing

The first step is to properly load and prepare the parallel corpus. Use `pandas` to load in the CSV file, or maybe try `polars` as an alternative to expand your data processing toolkit.

Each language needs its own vocabulary (`class Vocab`) so you can convert text to a sequence of tokens. For now, use a simple word-level approach (cf. assignment 1). Include at least the following special tokens into your vocabularies:

- `<pad>` (Padding Token).
- `<sos>` (Start-of-Sentence).
- `<eos>` (End-of-Sentence).

Apart from the vocabulary size, `Vocab` needs attributes to map from tokens to indices, and vice versa, and a way to track the frequency of every token. Make sure every sentence processed is turned into a list of indices, padded or truncated to a fixed maximum length (`max_len`), and ends with the `<eos>` token. Also remember assignment 1 where `attention_id` was a binary mask indicating if tokens where representing real text or just used for padding.

For inspiration, explore HuggingFace's tokenizer library.

Your `Vocab` class will integrate with a proper PyTorch dataset (see `torch.utils.data.Dataset`) for the task. The dataset represents each sentence pair as a sequence of token indices and ensures proper handling of variable sequence length with either padding (see `<pad>`) or truncating (ensure that the last token is always `<eos>`). For training, return tensor pairs (`input_sentence` and `output_sentence`), i.e. implement a useful `__getitem__`.

In order to automatically create batches you need to rely on `torch.utils.data.DataLoader`. If you struggle to make batching work with the default settings for a `DataLoader`, read up on collate functions to customize the batching process.

Model Architecture

Implement an Encoder-Decoder architecture based on recurrent models, where the decoder is utilizing attention as discussed in the lecture. At least the Decoder can't just utilize the ready-made full RNN layers from pytorch, you need to do more manual work there.

- It should be possible to specify the various dimensions and number of layers for each model.
- Choose between a GRU-style or LSTM-style architecture.
- For the specific attention mechanism, choose between Bahdanau style attention or Luong style attention. You must be able to explain both, if necessary.
- The decoder should also have a possibility to switch on dropout (for those layers where it is reasonable).

Training

During training, apply **teacher forcing** to guide the decoder by providing the actual target token as the next input at each time step, rather than the token predicted by the model.

To calculate the training loss, use PyTorch's `CrossEntropyLoss` while ensuring that potential `<pad>` tokens at the end of a sentence are ignored, as they do not contribute meaningful information to the learning process (cf. `attention_id`).

Throughout the training loop, log the loss at regular intervals to monitor progress and help diagnose any potential issues with learning. Utilize tensorboard for logging.

Evaluation

Evaluate your model during training on a held-out test set to monitor for overfitting. Use greedy decoding for translation: continue generating until `<eos>` is produced or `max_len` is reached.

To evaluate translation quality, use the BLEU score, accessible via HuggingFace's `evaluate` package. While BLEU has limitations, it's a convenient scalar metric for early experimentation. Using the library and the metric as a black-box is acceptable for now.

In addition to automated evaluation, conduct manual inspections of model outputs. If you don't speak French, don't worry —- Google Translate is your friend. Pick some example sentences from your test and compare your model's result with suggestions from Google. Even better (even when you speak French) do the following: Ask Google for French sentences (maybe you let it translate some English/German sentences), then let Google translate these to English, and then compare your *backtranslations* with the starting points.

Additionally, for a few example sentence pairs from your test set, visualize the model's attention during inference. Save the attention weights at each decoding step (consider using a flag to toggle this behavior). Then, create a heatmap where the rows represent the target (French) words and the columns represent the source (English) words: each cell at position (i, j) should display the attention weight assigned to source word j when generating target word i.

Interpret the resulting attention patterns: does the model appear to align words correctly (e.g., "cat" with "chat")? Are there surprising or insightful alignment patterns?

Optional Work

If you have time, explore ablations and variations of your architecture and evaluate those using the BLEU score. These experiments will give you deeper insight into what makes translation systems work.

Here are some suggestions:

- Switch between LSTM and GRU architectures and compare results. Does a bidirectional encoder (which paper did propose this originally?) help?
- Try *the other attention* mechanism than the one you initially chose.
- Evaluate the impact of optimization choice: compare Adam/AdamW to vanilla SGD, or even vSGD.

- Use a more advanced tokenizer, such as BPE. Adapt your code to support HuggingFace's tokenizer library rather than implementing BPE from scratch.
- Experiment with dropout, deeper networks, or larger hidden dimensions. Does increased model capacity help?

:🎉 ✨ You did it! Congratulations.