NLP exam FS24

HSLU Hochschule Luzern

November 22, 2023
Time: 90 minutes

Full name: _____

Initials: _____

Instructions

- This is a *closed book* exam. No course material or other additional material is allowed.
- Fill in your full name as well as your initials in block letters at the top of this page.
- Write your initials on each page.
- This exam consists of **14 pages**. Make sure you have all pages.
- If you have questions, raise your hand to clarify any uncertainties.
- Use the designated space for your answers. You may use the back of the page as additional space. If you do, indicate that your answer continues on the back.
- Write clearly and legibly. Only readable answers give points.
- Sign the declaration of academic integrity below.
- Good luck with the exam!

Question:	1	2	3	4	5	6	7	8	9	Total
Points:	6	8	12	12	10	16	8	12	6	90
Score:										

Declaration of Academic Integrity

By signing below, I pledge that the answers of this exam are my own work without the assistance of others or the usage of unauthorized material or information.

Signature: .....

1. **Introduction to NLP**

   (a) [2 points] What is the idea of the *NLP pipeline*?

   Solution: The NLP pipeline is the sequence of steps performed in classical NLP to understand words and sentences by assigning labels. In the pipeline, the later stages build on the results of the earlier stages.
   (1 point for sequence of steps, 1 point for later steps build on earlier ones)

   (b) [2 points] Name as many stages as you know.

   Solution:

   - tokenization
   - stemming
   - lemmatization
   - part-of-speech (POS) tagging
   - parsing
   - named entity recognition/resolution/linking (NER)

   (0.5 points for each named stage)

   (c) [2 points] Which stages of the NLP pipeline are taken care of by pretrained neural networks (such as BERT), and which stages do still have to be done by our (or a library's) code?

   Solution: We still have to do tokenization ourselves or with libraries like NLTK or Hugging Face. The rest is done by the NN internally (stemming and lemmatization are not done, but their purpose of giving the same meaning to words with the same stem is handled by the NN).
   (1 point for tokenization manually, 1 point for rest by NN, 0 points for all done by NN)

2. **Embeddings**

(a) [4 points] Why is cosine similarity a good word similarity metric for word2vec embeddings but not for one-hot encoding?

Solution: Word2vec embeds similar words (appear in similar contexts) close together, so the angle between their word vectors will be small, resulting in high cosine similarity. In one-hot encoding, all word vectors are orthogonal, so their cosine similarity is always 0, no matter how similar they are.
(2 points each for why it's good for word2vec and why it's bad for one-hot encoding)

(b) [4 points] In topic modeling, we perform topic analysis by manually assigning topic names, for example by inspecting the words with the highest weight for a given topic. In class, we saw an example where we assigned the topic name *arts* to a topic with the words *new*, *film*, *show*, *music*, *movie*, *play*, *musical*, *best*, *actor*, and *opera*. A cheap automatic solution is to pick the word with the highest weight, but this may not be representative of the topic; in the previous example we would have picked *new*. Suggest a better fully automatic solution.

Solution:

- Use an LLM to create a topic name from the topic words.
- Average the word embeddings of the topic words, then name the topic after the word embedding with highest cosine similarity to that average. Optional: Weight by importance while averaging.

(4 points for any reasonable solution that is fully automatic, 1 bonus point for second approach including weighing by importance)

3. **Recurrent Neural Networks**

(a) You train a tri-gram language model on the following data: "the quick brown fox jumps over the lazy dog. the fox jumps across the field." N-grams do not cross sentence boundaries. Give all probabilities as fractions.

i. [2 points] What is the probability of the next word being "over" given the prefix "the quick red fox jumps"?

Solution: Trigrams: (fox jumps — over), (fox jumps — across) → 1/2

ii. [2 points] What is the probability of the next word being "down" given the prefix "the fox jumps"?

Solution: Trigrams: (fox jumps — over), (fox jumps — across) → 0/2

iii. [4 points] What is the probability of the next word being "down" given the prefix "the fox jumps" when you use a tri-gram LM with add-one smoothing?

Solution: All 11 [or 13 with punctuation] trigrams: (the quick brown), (quick brown fox), (brown fox jumps), (fox jumps over), (jumps over the), (over the lazy), (the lazy dog), [(lazy dog .)], (the fox jumps), (fox jump across), (jumps across the), (across the field), [(the field .)] → (0 + 1) / (2 + 11) = 1/13 [or (0 + 1) / (2 + 13) = 1/15]

(b) Exploding gradients in RNNs.

i. [2 points] Explain how gradients can explode in RNNs.

Solution: In backpropagation through time, gradients can propagate over many steps of an unrolled computation graph. If the gradient is larger than 1 and gets multiplied with weights also larger than 1 over many time steps, the gradient grows exponentially.
(1 point for many multiplication steps, 1 point for gradient large/larger than 1)

ii. [2 points] Name two techniques (not neural network architectures) to fix exploding gradients.

Solution:

- Gradient clipping

- Regularization (on weights or activations)
- Gating

4. **Attention**

(a) [12 points] The code below shows the forward pass of a decoder LSTM with attention. Unfortunately, a part of the loop over decoder time steps has been lost. Reconstruct it with the help of some of the functions below. Write Python code (exact syntax will not be graded). The number of dotted lines is no indication of how many code lines you need. If you use a different version of attention in RNNs than the one we have used in class, describe it.

- F.layer_norm
- F.relu
- F.softmax
- self.compute_context_vector
- self.project_to_higher_dim
- self.project_to_lower_dim
- self.project_to_same_dim
- torch.add
- torch.cat
- torch.matmul
- torch.mul
- torch.ones
- torch.stack
- torch.zeros

```
def forward(self, y, encoder_hidden_states):
    h = torch.zeros(self.hidden_dim)
    c = torch.zeros(self.hidden_dim)
    hidden_states = []

    # loop over the target sequence
    for y_i in y:
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        .....
        h, c = self.cell(y_i, (h, c))  # LSTM cell forward pass
        .....
        .....
        .....
        .....
        hidden_states.append(h)

    return torch.stack(hidden_states), (h, c)
```

Solution:

```
def forward(self, y, encoder_hidden_states):
    h = torch.zeros(self.hidden_dim)
    c = torch.zeros(self.hidden_dim)
    hidden_states = []

    # loop over the target sequence
    for y_i in y:
        context_vec = self.compute_context_vector(h,
        ↪  encoder_hidden_states)
        concatenated = torch.cat([h, context_vec],
        ↪  dim=-1)
        h = self.project_to_lower_dim(concatenated)
        h, c = self.cell(y_i, (h, c))  # LSTM cell
        ↪  forward pass
        hidden_states.append(h)

    return torch.stack(hidden_states), (h, c)
```

Points:

- 1 for compute context vector, 2 for correct arguments
- 2 for cat, 2 for correct arguments
- 2 for down-project
- 1 for all results saved and used correctly
- 1 for no additional steps before cell
- 1 for no additional steps after cell

5. **Transformer**

(a) [4 points] Name 2 fundamental reasons why the Transformer outperforms the RNN.

Solution:

- The self-attention mechanism allows direct attention to all other tokens in the sequence.
- Removing the recurrence gets rid of the temporal dependency and allows training on the entire sequence in parallel. This allows to train on more data/deeper networks in the same amount of time.

(b) [4 points] Why do we need position encoding in the Transformer, but not in the RNN?

Solution: The self-attention output is a weighted sum of all inputs, so the ordering gets lost. In the RNN, the elements are processed one after the other, so the ordering is always clear.
(2 points for sequential processing in RNNs, 2 points for ordering lost in Transformer due to sum in attention, only 1 point due to parallelization or independence of tokens)

(c) [2 points] What is the big weakness of the attention mechanism in the Transformer, and the main reason researchers have been looking at recurrent architectures again in the past year?

Solution: In self-attention, every token looks at all other tokens. This incurs a computation complexity quadratic in the sequence length. This is especially a problem for long sequences, and prevents large context windows. In the RNN, memory stays constant and does not grow with the sequence length.
(2 points for quadratic complexity of self-attention)

6. **Pretraining**

Here is an example from Winogrande: *Sarah was a much better surgeon than Maria so _ always got the easier cases.* The options are *Sarah* and *Maria*.

(a) [3 points] Explain why this example (and the entire dataset) is hard to solve for word embeddings, as opposed to RNNs and Transformers.

Solution: Word embeddings don't have ordering, which is a requirement to solve the task.

(b) [3 points] Explain why this example (and the entire dataset) is hard to solve for a model finetuned from a randomly initialized neural network, as opposed to a pretrained one.

Solution: The dataset is too small to learn the common sense relations that this task is about.
(Half points for *learning language*, which is only part of the task)

(c) [5 points] Create a one-shot prompt for which a good model should output the correct solution to the introductory example in this section (by actually solving the task, not just repeating/outputting the solution).

Solution: Fill in the blank in the following sentences.
Sentence: He never comes to my home, but I always go to his house because the _ is smaller.
Option 1: home
Option 2: house
Answer: home
Sentence: Sarah was a much better surgeon than Maria so _ always got the easier cases.
Option 1: Sarah
Option 2: Maria
Answer:
(1 point for instruction, 0.5 points for each part (sentence, options, answer), answer at the end should be left empty, -1 point for wrong answer in demonstration, -3 points for no one-shot example)

(d) [5 points] Create a one-shot chain-of-thought prompt for the above example.

Solution: Fill in the blank in the following sentences.
Sentence: He never comes to my home, but I always go to his house because the _ is smaller.
Option 1: home
Option 2: house
Answer: People prefer being in larger rather than smaller rooms. Since I always to his house, this means the house is larger and the home is smaller. The answer is: home
Sentence: Sarah was a much better surgeon than Maria so _ always got the easier cases.
Option 1: Sarah
Option 2: Maria
Answer:
(1 point for keeping rest the same, 3 points for good chain-of-thought explanation in example, 1 point for correct format and leaving final answer open)

7. **Text Generation**

(a) Nucleus sampling.

i. [1 point] What value do you have to set for p in Nucleus sampling to get the same results as greedy decoding? Round to 2 decimal points.

> **Solution:** 0 (any answer below 0.1 gives a point)

ii. [4 points] Apply Nucleus sampling with $p = 0.75$ to the model's following next word predictions and associated probabilities. Write down the updated distribution (word and updated probability, in fractions) we sample our next word from in Nucleus sampling.

- where – 0.4
- that – 0.2
- which – 0.1
- so – 0.1
- such – 0.05
- in – 0.04
- a – 0.02
- . – 0.01

> **Solution:** Selected words: where, that, which, so. Total probability: 0.8. Renormalize by 0.8.
>
> - where – 1/2
> - that – 1/4
> - which – 1/8
> - so – 1/8
>
> (1 point for correct words, 3 for correct fractions)

(b) [3 points] Will the following candidate translation achieve high BLEU score? Why? (Hint: 1 mile = 1.60934 kilometers)
Candidate: I ran a mile.
Reference: My running went for 1.61 kilometers.

> **Solution:** It will not achieve high BLEU score, as BLEU is a word-overlap based metric that does not use stemming/lemmatization. None of the words match between reference and candidate.

8. **Text Classification**

(a) [12 points] Your colleague AM has been working on a text classification task and created a pull request (see next page). You were assigned to peer review his code. Check the code for correctness and efficiency, and suggest improvements. State the line number, what needs to be changed and why. You do not have to provide the implementation for the changes. You can assume that all imports are present and the comments are not misleading.

Solution:

- 10, 11: lowercasing for a cased model
- 10, 11: stemming for a pretrained model not trained with stemmed words
- 10, 11: 2x option1 added to options
- 12: sort inputs alphabetically removes ordering
- 23: not putting model into ‘eval()‘ mode
- 25: tokenizer must use ‘return_tensors = ’pt’‘
- 26: not using with ‘torch.no_grad()‘
- 29: classify from padding tokens
- 33: softmax should use ‘F.softmax‘ or ‘torch.nn.Softmax(dim=-1)‘
- 33: softmax should state dimension
- 36: normalization after softmax means predictions are no longer probabilities
- 37: dropout on predictions zeros the predicted class

(2 points for each mistake found)

```python
def evaluate():
    # read data
    data = load_dataset('winogrande', 'winogrande_l',
    ↪  split='validation')

    # preprocess
    processed = []
    stemmer = nltk.stem.PorterStemmer()
    for example in data:
        options = []
        options.append(stemmer.stem(example['option1'].lower()))
        options.append(stemmer.stem(example['option1'].lower()))
        options = sorted(options)
        sentence = f"Sentence: {example['sentence']}, Option 1:
        ↪  {options[0]}, Option 2: {options[1]}"
        processed.append(sentence)

    # load model
    tokenizer = AutoTokenizer.from_pretrained(
        'google-bert/bert-base-cased'
    )
    model = AutoModelForSequenceClassification.from_pretrained(
        'google-bert/bert-base-cased'
    )

    # run evaluation
    inputs = tokenizer(processed, padding=True)
    outputs = model(**inputs)

    # outputs.logits shape: [batch_size, seq_len, hidden_dim]
    logits = outputs.logits[:, -1]
    if self.num_classes == 1:
        predictions = torch.sigmoid(logits)
    else:
        predictions = torch.nn.Softmax(logits)

    # normalize and regularize
    predictions = F.layer_norm(predictions)
    predictions = F.dropout(predictions)

    return predictions
```

Evaluation function for text classification.

9. **Research Topics and Guest Lectures**

(a) [3 points] In the argument mining guest lecture, we have seen that the labels that a large language model gives for the same question can flip for various reasons. Devise a strategy which can reduce the uncertainty in the output label.

Solution:

- Rerun the prompt with different random seeds
- Rerun with different prompts
- Rerun with higher temperature
- Use interpretability techniques
- Ask the model to output its confidence (unreliable)
- Chain-of-thought prompting (only for reasoning tasks)

(b) [3 points] In the NLP in health guest lecture, we have seen that not all models perform well on medical text data. Since the models are usually large (e.g. EPFL's Meditron model is built from Llama 3 70B), we need a lot of resources to evaluate them. In contrast, the tokenizer is cheap to download and run. Given a list of medical terms you care about and only the tokenizers of your candidate models, how can you select the most promising models for an in-depth evaluation?

Solution: Run the tokenizers on the medical terms and see which one tokenizes them into the fewest tokens. This means that the model has been pretrained on medical texts (since they have received their own token in BPE tokenization).