Natural Language Understanding, Generation, and Machine Translation (2023–2024)

Yifu Qiu, Shay Cohen and Alexandra Birch-Mayne
School of Informatics, University of Edinburgh

Coursework 2: Neural Machine Translation

> **This assignment is due on Friday, 22th March 2024, at 12:00 noon, GMT.**

Prerequisites   You should finish Lab 3: Tensor Computation in PyTorch BEFORE starting this coursework. Lab 3 will teach you tensor computation which is heavily used in this coursework. You can find Lab 3 materials on LEARN.

Executive Summary   Your task will be to work with a simple baseline NMT model for German to English, analysing its code and evaluating its performance. To improve over the baseline NMT model, you will implement the lexical attention model as described in Nguyen and Chiang (2017). Finally, you will analyse a basic implementation of the Transformer architecture and implement the multi-head attention mechanism according to Vaswani *et al.* (2017) to complete the model.

IMPORTANT: While modifying the baseline code may only take you a few minutes or hours, training the extended models will take you **A LOT OF TIME**. You might implement something in thirty minutes and leave it to train overnight. Imagine that you return the next morning to find it has a bug! If the next morning is the due date, then you'll be in a pickle, but if it's a week before the due date, you have time to recover. So, if you want to complete this coursework on time, **start early**.

Using ChatGPT   School policy requires you to complete coursework yourself, using your own words, code, figures, etc. and to acknowledge any sources of text, code, figures etc. that are not your own. This policy does not prevent you from using ChatGPT, but regularises your usage of ChatGPT. Using such an assistant without acknowledgement is a form of academic misconduct.

Overview of the assignment   There are seven questions in total, divided into four areas of interest. Part 1 asks you to analyse the code and train an improvement to the baseline without modifying the code. This part should not be too time consuming.

However, (re-)training the model can take up to 10 hours, so make sure to plan accordingly. Part 2 asks you to consider extensions to the baseline already supported in the code. Part 3 considers the lexical attention model and you will have to implement this in code, train a new model and discuss your results. Part 4 asks you to consider the Transformer model and requires you to add the Multi-Head attention code to complete the model. Make sure to allocate sufficient time to implement, train, and evaluate your model extensions.

Submission   You will submit **two** items for assessment for this coursework. You will deliver a PDF document detailing your answers to all questions, including code sections where appropriate, and also a ZIP archive containing the files specified below. [**IMPORTANT:** Your PDF document should be uploaded via Gradescope and your code files should be uploaded via Blackboard Learn. We will mark mainly based on PDF document and code screenshots on Gradescope, and the code uploaded to blackboard learn will be used to verify reproducibility.] **Do not include any names in either the code or the write-up.** The coursework will be marked anonymously since this has been empirically shown to reduce bias. For your writeup you must do the following:

- **Write your answers to all the questions in a single file titled `<UUN>.pdf`.** For example, if your UUN is `S1234567`, your corresponding PDF should be named `S1234567.pdf`.

- The answers should be clearly numbered and can contain text, diagrams, graphs, formulas, code snippets, where appropriate. Do not repeat the question text. If you are not comfortable with writing maths on LaTeX/Word you are allowed to include scanned handwritten answers in your submitted PDF. You will lose marks if your handwritten answers are illegible.

- For questions that require modifying the code or adding explanations to the code (i.e. Question 1, 5, 6 and 7), please put the screenshots of your comments or modified code snippet into the answers for corresponding questions in the PDF document.

- On Gradescope, select "NLU Coursework 2". Upload your `<UUN>.pdf` to this assignment. When you submit your report on Gradescope, please indicate the page of each question's answer accordingly.

- Please make sure you have submitted the right file. We cannot make concessions for students who turn in incomplete or incorrect files by accident.

For your code and parameter files:

- Compress your code for `lstm.py`, `train.py`, `transformer.py` and

`transformer_helper.py` into a ZIP file named `<UUN>.zip`. For example, if your UUN is `S1234567`, your corresponding ZIP should be named `S1234567.zip`.

- On Blackboard Learn, select the Turnitin Assignment "Coursework 2 CODE". Upload your `<UUN>.zip` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `S123456`.

Good Scholarly Practice Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put your work in a public repository then you must restrict access only to yourself and your partner. **You are not permitted to publish your code solution online.**

For your write-up, and particularly on the final questions, you should pay close attention to the guidance on plagiarism. Your instructors are **very good** at detecting plagiarism that even Turnitin can't spot. In short: the litmus test for plagiarism is not the Turnitin check—that is simply an automated assistant. If you have borrowed or lightly edited someone else's words, you have plagiarised. We are fully aware of what code examples and tutorials are on the Internet. Write your report in your own words. Your score does not reply on your writing skill. As long as you can express clearly, that is fine.

Part 0: Setting up Environment

Python Virtual Environment For this assignment you will be using Python 3.8 along with a few open-source packages, with PyTorch being the key library.

The instructions below are for DICE and is for the CPU version of PyTorch. You are free to use your own machine. We have tested the instruction on DICE and MacOS. If you are working on the Windows system there might be differences (but should not be very difficult to adjust). The key point here is to install PyTorch. You can also use lab sessions and TA office hours asking for help for setting up the environment.

Now we assume you have opened a terminal on a DICE machine. Run the following commands **one-by-one** (not all at once). Waiting for each command to complete will help catch any unexpected warnings and errors. The total installation is about 4.33GB, please ensure you have sufficient space using the `freespace` command on DICE.

First install Miniconda from the home directory of your DICE user space (respond yes to all prompts). *You can skip this stage if you already have Miniconda installed.*

```
$> wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
$> bash ./Miniconda3-latest-Linux-x86_64.sh
$> rm ./Miniconda3-latest-Linux-x86_64.sh
$> source ~/.bashrc
```

Now, your default Python version should be Python 3.8 to 3.10. Confirm with `python3 --version`. Then create a new environment called `nlu`.

1. Clone the GitHub repository to an appropriate location in your workspace
   You must do this even if you have the environment set up:
   ```
   $> git clone https://git.ecdf.ed.ac.uk/nlu_public/course_materials.git
   $> cd course_materials/2024/coursework/nlu-cw2
   $> git init #In example.sh we use git to find ROOT path.
   ```

2. Create an environment:
   ```
   conda create -n nlu python=3.10
   ```

3. Activate the `nlu` virtual environment:
   ```
   $> conda activate nlu
   ```

4. Install Pytorch and others: `$> pip install torch==1.13.1 tqdm numpy`

5. **Optional** Clean your workspace to free up space:
   ```
   $> conda clean --all
   ```

You should now have all the required packages installed. You only need to create the virtual environment and perform the package installations (step 1-5) **once**. However, make

sure you activate your virtual environment (step 3) **every time** you open a new terminal to work on your assignment. Remember to use the `conda deactivate` command to disable the virtual environment when you don't need it.

1. Activate the environment:
   ```
   $> conda activate nlu # Ready for working
   ```

2. Deactivate the environment (if you want to work on something else):
   ```
   $> conda deactivate nlu
   ```

Additionally, learning to use UNIX tools such as `screen` and DICE tools like `longjob` will make running code for this assignment much easier. Run `man screen` or `man longjob` for guidance with this.

Baseline NMT model   The baseline code is already in `nlu-cw2` directory which you have just cloned.

You'll find several directories inside the downloaded `nlu-cw2` folder, including `europarl_raw` containing raw English and German parallel data, `europarl_prepared` containing the pre-processed data your models will be trained on, and `seq2seq` containing the code you will be asked to extend. Moreover, you will find several python files of importance to the assignment (**DO NOT MODIFY FILES MARKED WITH ***):

- `train.py`* is used to train the translation models.

- `translate.py`* translates the test-set greedily using model parameters restored from the best checkpoint file and saves the output to `model_translations.txt`.

- `example.sh`. This is a suggested outline of a single experiment run to train a model, generate translations and then find the test-set BLEU score.

To train a baseline model, follow `example.sh` without modifying any lines. This script includes training and inference and is designed to help you get started, but you can modify it for later parts of the coursework. Additionally, instead of directly modifying `example.sh`, we recommend you to modify a duplicate:

```
$> cp example.sh example_dev.sh
```

You can specify the hyper-parameters for the training using the appropriate argument flags, but **we strongly recommend training with the default settings**. Run this script directly by running `bash example.sh` in the directory downloaded from GitHub. You can also just train a model by running: `python train.py`.

After calling the training script, you should see a progress bar denoting the training progress for the current epoch. Training will continue until no improvement can be observed on the development set for 10 consecutive epochs. After each epoch, the latest

model file is saved to disk as `checkpoint_last.pt`. If the model achieved a lower dev-set perplexity in the concluded epoch than in the previous epochs, a 'best' model file is saved to disk, as well, as `checkpoint_best.pt`. You can find the checkpoint files in the `checkpoints` directory or the location you specify using the `--save-dir` argument to the training script. After your model has finished training, use it to translate the test set by running: `python translate.py`.

The translations will be output to the file `model_translations.txt`. Next, use the `multi-bleu.perl` script to calculate the test-BLEU score of the baseline model:

```
perl multi-bleu.perl -lc europarl_raw/test.en < model_translations.txt
```

Report: (1). the BLEU score, (2). the validation-set perplexity and, (3). training loss from the final epoch. Then back up the checkpoints directory (e.g. by renaming it to `checkpoints_baseline` but `example.sh` does this automatically). This model is still quite basic and trained on a small dataset, so the quality of translations will be (very) poor. Your goal will be to see if you can improve it.

The current translation model implementation in `seq2seq/models/lstm.py` encodes the sentence using a bidirectional LSTM: one LSTM passing over the input sentence from left-to-right, the other from right-to-left. The final states of these LSTMs are concatenated and attended over by the decoder, using global attention with the general scoring function as described in Luong *et al.* (2015). While the encoder is implemented as a single-layer bidirectional RNN equipped with the LSTM cell, the decoder is a single-layer unidirectional RNN, also equipped with the LSTM cell. The file `seq2seq/models/transformer.py` defines an implementation of the Transformer architecture from Vaswani *et al.* (2017). The layers, positional embeddings and attention mechanism (that you must complete) are contained in `seq2seq/models/transformer_helper.py`.

Part 1: Getting Started

Question 1: Understanding the Baseline Model [10 marks]

Before we go deeply into modifications to the translation model, it is important to understand the baseline implementation, the data we run it on, and some of the techniques that are used to make the model run on this data.

The file `seq2seq/models/lstm.py` contains explanatory comments to step you through the code. Five of these comments (A-D) are missing, but they are easy to find: search for the string `__QUESTION` in the file. A fifth comment (E) is missing from `train.py`. There are also questions listed these comments for you to answer. For each of these cases:

1. Add comments in the code to answer the associated questions

2. Copy your comments to your report or take screenshots accordingly (we will mark the comments in your report, not the code, so it is vital that they appear there)

If you aren't certain what a particular function does, refer to the PyTorch documentation: `https://pytorch.org/docs/stable/index.html`. (However, explain the code in terms of its effect on the MT model; don't simply copy and paste function descriptions from the documentation. If you use ChatGPT to help you understand a Pytorch function, it may "hallucinate" (Bang *et al.*, 2023) and not always give you reliable answers.).

Before you continue to improve the model, validate that you can train the baseline model by training the LSTM with default arguments (given in `lstm.py`). The script `example.sh` shows you how to do this. Confirm that you can train this model and your results look similar to these metrics (your baseline performance may vary slightly due to the random nature of model parameter initialisation):

- `training loss during last epoch: 2.145`

  Your own training loss should be around 2.1 ± 0.3 depending on your random seeds. Since learning neural network is highly stochastic, everytime you run it with a different random seed, you should expect different (but close) numbers.

- `validation set perplexity during last epoch: 26.8`

  Your own perplexity should be around 26.8 ± 3.0

- `test set BLEU: 11.03`

  Your own BLEU should be around 11.03 ± 1.50

If your model performs similarly to the baseline, proceed with the rest of the assignment. If not then discuss with your partner, lab demonstrator or TA. Training the model may take between 4-6 hours depending on your CPU capability.

Question 2: Understanding the Data [10 marks]

The dataset we provide is a small sample of the Europarl Corpus (Koehn, 2005), which is a transcription of proceedings from the European Parliament. We will focus on parallel German and English data, providing 10,000 sentence pairs for training, and 500 pairs for validation and testing. In preparing the training data, word types that appear only once are replaced by a special token, `<UNK>`. This prevents the vocabulary from growing out of hand, and enables the model to handle unknown words in new test sentences (which may be addressed by post-processing). **Note that the data has already**

been tokenised for you, so you do not need to use further tokenisation software. "Tokenize" means splitting sentences into words/ subwords.

Examine the parallel training data located in the `europarl_raw` directory (`train.en` and `train.de`) and answer the following questions in your report.

1. How many word tokens are in the English data? In the German data? Give both the total count and the number of word types in each language.

2. How many word tokens will be replaced by `<UNK>` in English? In German? Subsequently, what will the total vocabulary size be before and after `<UNK>`?

3. Inspect the words which will be replaced by `<UNK>`. To do so, firstly extract the replaced words, then **sort them into alphabetic order**, then write them into a file, then open this file and look at these sorted words with your bare eyes. Is there any linguistic phenomenon associated with the replaced words? Are these words common or rare (for example, "parameters" is a common word and "reparameterization" is a rare word)? Based on the linguistic phenomenon you observed, how would you suggest to improve the tokenization process?

4. How many unique vocabulary tokens are the same between both languages? How could we exploit this similarity in our model? You don't have to consider false friends such as the English verb 'die' and German article 'die', just treat them as the same.

5. Given the observations above, how do you think the NMT system will be influenced by sentence length, tokenization process, and unknown words of the two languages?

Part 2: Exploring the Model

Let's explore the decoder. It makes predictions one word at a time from left-to-right, as you can see by examining the decoder module in the file `seq2seq/models/lstm.py` and the greedy decoding script in `translate.py`. Prediction works by first computing a distribution over all possible tokens conditioned on the input sentence. We then choose the most probable token, output it, add it to the conditioning context, and repeat until the end-of-sentence token (`<EOS>`) is predicted.

Question 3: Improved Decoding [10 marks]

1. Currently, the model uses greedy decoding, which simply chooses the maximum-probability token at each time step. Can you explain why this might be problematic? Give language specific examples as part of your answer.

2. How would you modify this decoder to do beam search - that is, to consider multiple possible translations at each time step. Your answer should be formulates as **pseudo codes**. You don't need to actually implement beam search – pseudo codes will be enough. The purpose of this question is simply for you to think through and list the important steps that is requires for beam search. Use math equations and textual explanations together to illustrate your idea.

   Below is an example of pseudo code for greedy decoding

   i current prob, current state = decoder(previous word, previous state)   # use the decode to generate the probability of all word at current step

   ii current word = argmax(current prob)   # find out the word with largest local probability

   iii previous word = current word; previous state = current state   # prepare for the next step decoding

3. Often with beam search (and greedy decoding), the decoder will output translations which are shorter than one would expect. How would you modify the decoding process to encourage longer sentence generation?

Question 4: Adding Layers [5 marks]

1. Change the number of layers in the encoder = 2, decoder = 3. You don't need to modify the codebase to train a deeper model - this is already supported by the provided code. Inspect the source code to find out how you can control the number of encoder and decoder layers via command line arguments. Train a system with this deeper architecture, and report the command that you used in your write up.

2. What effect does this change have on dev-set perplexity, test BLEU score and the training loss (all in comparison to the baseline metrics given in Q1)? Can you explain why it does worse/better on the training, dev, and test sets than the baseline single layer model? Is there a difference between the training set, dev set, and test set performance? Why is this the case?

Part 3: Lexical Attention

Question 5: Implementing the Lexical Model [15 marks]

In this part of the assignment, we ask you to augment the encoder-decoder with the lexical model defined in Section 4 of Nguyen and Chiang (2017). For this task, your primary guidance should be the descriptions provided in the paper. Moreover, we

have marked the different points in the encoder-decoder implementation where you must insert your code and take a screenshot to your submitted document (marked as `__QUESTION-5`).

Implementing the lexical model can be roughly subdivided into three steps:

1. Compute the weighted sum of source embeddings using weights extracted from the decoder-to-encoder attention mechanism.

2. Define the feed-forward layers used to project the weighted sum of source language embeddings.

3. Incorporate the lexical context tensor into the calculation of the predictive distribution over output words.

To accomplish this, **you only need to modify** `lstm.py` and **nothing else**. Implementing the modifications should not take you very long, but retraining the model will. Paste your code snippet for this question into the writeup.

NOTE: We recommend that test your modifications by retraining on a small subset of the data (e.g. a thousand sentences). To do that, you should add the flag `--train-on-tiny` to the set of arguments when executing `train.py`, i.e.:

```
python train.py --train-on-tiny
```

The results will not be very good; your goal is simply to confirm that the change does not break the code and that it appears to behave sensibly. This is simply a sanity check, and a useful time-saving engineering test when you're working with computationally expensive models like neural MT. For your final models, you should train on the entire training set.

Implement lexical model as described above, all changes to the baseline implementation **must be done in the decoder**. You should be able to easily access both source embeddings (assigned to the `src_embeddings` variable) as well as attention weights specific to each decoding step (assigned to the `step_attn_weights` variable). Adding your code to the specified positions within the decoder architecture will help ensure that everything works correctly.

When you have completed your implementation and you are sure that it doesn't break your model: retrain your translation model after augmenting it with the lexical model by running the following command:

```
python train.py --decoder-use-lexical-model True
```

Again, explain how the change affects results compared to the baseline in terms training set loss, dev perplexity, and test BLEU scores. Consider whether the addition of lexical

translation is beneficial or detrimental to performance on these automatic metrics.

Optionally, you can also examine the output translations – using translations that differ between models as motivating examples in your explanation of the effects of lexical attention. You do not need to exhaustively examine every output – but consider if you can find any trends in improvement between models (there may be none). In your report, you can discuss a trend you identify with a maximum of **five example output pairs**. Do not include all your model outputs in the report.

Part 4: Transformers

Modern NMT systems rely heavily on the Transformer architecture (Vaswani *et al.*, 2017), which has emerged in recent years as a viable competitor to the more established LSTM-based approach to sequence transduction. Transformers are a non-recurrent architecture which has set state-of-the-art performance in many areas of MT. We recommend you start this section by reading Vaswani *et al.* (2017) and the blog post `http://peterbloem.nl/blog/transformers`.

Question 6: Understanding the Transformer Model [10 marks]

This question asks you to similarly complete five explanatory comments (6A-6E) in `seq2seq/models/transformer.py` (6A-6C) and `seq2seq/models/transformer_helper.py` (6D, 6E). Find these again by searching for the string `_QUESTION-6` in the file.

1. Add comments in the code to answer the associated questions

2. Copy or take a screenshot for your comments to your report (we will mark the comments in your report, not the code, so it is vital that they appear there).

Again, you can use the PyTorch documentation if you don't understand a function. You must still explain each function in the model in your own words.

Question 7: Implementing Multi-Head Attention [40 marks]

In the final section of the coursework, we ask you to implement Multi-Head attention from Section 3.2 of Vaswani *et al.* (2017). As before, your main guidance should be the equations in the paper itself. If you use external resources, you **must** cite these.

Implementing multi-head attention can be roughly subdivided into three steps:

1. Linear projection of Query, Key and Value.

2. Computing scaled dot-product attention for h attention heads.

3. Concatenation of heads and output projection.

To accomplish this, you need to modify the `forward` method in the `MultiHeadAttention` class in `transformer_helper.py` and nothing else (marked as `_QUESTION-7`). Please also copy or take a screenshot of your implemented code to the report, because we will mark on your report only. This is a larger task than the lexical model and may take you some time to develop and test this function. There are some comments and checks to guide you about the required shape of the output tensors. When writing down your implementation, instead of compressing every tensor operations into one single line, like

$z = (x*y.mean(-1) + x*mask.expand(-1)).sum(-1).mean()$ # DO NOT DO THIS

You should do it step-by-step and explain the shape of every output tensor, like:

$prod = x*y.mean(-1)$ # prod.size = [...]
$x\_masked = x*mask.expand(-1)$ # x_masked.size = [...]
$z\_elementwise = prod + x\_masked$ # z_elementwise.size = [...]
$z\_average = z\_elementwise.sum(-1).mean()$ # z_average.size = [...]

Copy **only** your `forward` function from the `MultiHeadAttention` into your report, as we will mark this first.

As before, we recommend that you test the modifications by retraining on a small subset of the data. To do this for the Transformer, add the following arguments to `train.py`, i.e.:

```
python train.py --train-on-tiny --arch transformer
```

When you have completed multi-head attention, train a transformer model with the default arguments on the full dataset. Report your test-set BLEU and the final epoch training loss and validation-set perplexity as you did for the baseline models. Similar to before, this might take **a long time (5 hours)**. Can you explain why it does worse/better on the development and test sets than the previous LSTM-based models? Is there a difference between the training set, dev set, and test set performance? You should compare to the previous models you have trained.

You might notice that the quality of outputs is poor and the model converges quickly. Considering the dataset and the model size, give **two reasons** why this may be the case? How could we possibly improve performance? Note you do not have to retrain this model. You can answer without having a functioning model if you compare between the literature and the provided code.

Acknowledgements

References

Yejin Bang, Samuel Cahyawijaya, Nayeon Lee, Wenliang Dai, Dan Su, Bryan Wilie, Holy Lovenia, Ziwei Ji, Tiezheng Yu, Willy Chung, Quyet V. Do, Yan Xu, and Pascale Fung. A multitask, multilingual, multimodal evaluation of ChatGPT on reasoning, hallucination, and interactivity. In Jong C. Park, Yuki Arase, Baotian Hu, Wei Lu, Derry Wijaya, Ayu Purwarianti, and Adila Alfa Krisnadhi, editors, *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 675–718, Nusa Dua, Bali, November 2023. Association for Computational Linguistics.

Philipp Koehn. Europarl: A Parallel Corpus for Statistical Machine Translation. In *Conference Proceedings: the tenth Machine Translation Summit*, pages 79–86, Phuket, Thailand, 2005. AAMT, AAMT.

Minh-Thang Luong, Hieu Pham, and Christopher D Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

Toan Q Nguyen and David Chiang. Improving lexical choice in neural machine translation. *arXiv preprint arXiv:1710.01329*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, 2017.