Naomi Bleiker (nbleiker, 17-648-981)
Alison Y. Kim (akim, 21-738-877)

Exercise 04

Maschinelle Übersetzung
16. Mai 2022

1 Instances of LayerNorm in `JoeyNMT`

1.1 Layer normalization: brief introduction

Normalization is the process of transforming data into a particular range, often $[0, 1]$. A common application of normalization is found in language modeling. When a language model outputs token scores at a given time step, normalization is typically used to "map" these scores to a probability distribution, where values are restricted to $[0, 1]$ by definition.

When training neural networks, it is often advantageous to apply normalization to datasets. *Batch normalization* has been shown to reduce training time and display faster convergence, as well as display robustness to hyperparameters [2]. However, relatively poor performance is noted when using small batch sizes/or and varying input lengths [1], the latter of which is feasible in NLP tasks in which inputs can vary in size (e.g. sentence length). In particular, when training RNNs with such data, it is not always clear how to calculate normalization statistics. Additionally, batch normalization is incompatible with online learning and large distributed models [1].

In *layer normalization*, normalization is performed over all hidden units in the same layer. For the lth hidden layer, the normalization statistics are given by

$$\mu^l = \frac{1}{H} \sum_{i=1}^{H} a_i^l \tag{1}$$

$$\sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^{H} \left(a_i^l - \mu^l\right)^2} \tag{2}$$

where H is the number of hidden units in layer l and $a^l = \langle a_1^l, \ldots, a_H^l \rangle$ is the vector representation of the summed inputs to the neurons in layer l [1]. Clearly, these values depend only on the *summed* inputs to a layer at the current time step, which enables training with inputs of varying lengths. Additionally, layer normalization can be used with small batch sizes, which is often required when training large distributed models. Layer-normalized networks are also robust to issues such as vanishing or exploding gradients.

1.2 Instances of layer normalization: `JoeyNMT`

We installed `JoeyNMT` per documentation instructions[1] and focused our search in the `joeynmt/joeynmt` directory.

After scanning [3], our first intuition was to look for instances of post-normalization in scripts referring to encoder and decoder architectures. In `decoders.py`, one indeed encounters an implementation of layer normalization within the initialization of class `TransformerDecoder`. It can be activated when instantiating this class by passing the keyword argument (kwarg) `layer_norm`. As seen in lines 534-535, it is performed via the PyTorch implementation `nn.LayerNorm` with arguments `hidden_size` (number of units in hidden layer) and `eps` (a term that facilitates numerical stability).

[1] https://joeynmt.readthedocs.io/en/latest/install.html

The final lines of the forward pass show that **pre-normalization** is performed, i.e. the input is normalized *before* it is passed through the sublayer.

In `encoder.py` layer normalization is implemented in each encoder layer of the `TransformerEncoder` class. It can be activated by passing the keyword argument `layer_norm` when initializing the class. Again, normalization is performed via `nn.LayerNorm`. In lines 252-254 we can see that **post-normalization** is implemented because the input is normalized **after** it is passed through the sublayer.

Further instances of layer normalization can be found in `transformer_layers.py` in the following classes: `PositionwiseFeedForward`, `TransformerEncoderLayer` and `TransformerDecoderLayer`. In each of the classes, either **pre- or post-normalization** is implemented. Post-normalization is the default (i.e. the argument `layer_norm` is per default set to "post") but pre-normalization can also be implemented by assigning "pre" to the parameter `layer_norm` when initializing the respective class.

2 Implementing pre- and post-normalization

2.1 Training procedure

We created two copies of the baseline model configuration file `deen_transformer_regular.yaml`: one for pre-normalization and the other for post-. They are named `deen_transformer_{pre, post}.yaml`, respectively, and can be found in the `joeynmt` repository. The only changes to these configuration files were the additions of a `layer_norm="{pre, post}"` argument under both `model/{encoder, decoder}`.

After making the aforementioned changes, we ran the training script `train.sh`, changing the `model_name` argument and the log and error filenames accordingly. This was carried out twice, once per model.

2.2 Submission

2.2.1 Unresolved path errors

To resolve path errors that we faced during training, we reshuffled some files between the exercise and `JoeyNMT` directories. We advise the user to keep everything as-is (i.e. as submitted) so that they can run the training script without facing path errors.

1. **Created** model configurations `joeynmt/configs/deen_transformer*`
2. **Copied** `mt-2023-ex04/data` to `joeynmt/data`
3. **Copied** `mt-2023-ex04/scripts/train.sh` to `joeynmt/scripts` (with appropriate modifications)

2.2.2 Training execution

After cloning the JoeyNMT directory, one can modify the variable `model_name` in the training script `joeynmt/scripts/train.sh` to select the model of choice. (See `joeynmt/configs` for a list of available models.) Then, **from the uppermost directory level** of `joeynmt`, one can run the following on the command line:

```
bash ./scripts/train.sh
```

2.2.3 Training submission

Ultimately, the reader should refer to the exercise directory for all submission materials:

1. Configuration files: `models/deen_transformer_{pre, post}/config.yaml`
2. Log files (error, out): `logs/deen_transformer_{pre, post}/*`
3. Training files: `models/deen_transformer_{pre, post}/{train.log, validation.txt}`
4. Plot, table: `scripts/plot_ppl.py`, `ppl_plot/ppl_{lineplot.png, table.csv}`
5. Documentation: `akim_nbleiker_mt_ex04.pdf`

The checkpoint, hypothesis, and `TensorBoard` files were not uploaded due to the large file sizes, which bottlenecked our initial attempt to push them to our repository.

2.3 Discussion

Given that there is a difference in the training progress for the three models, can you think of a reason for it?

Since the point of the assignment was to implement and observe two forms of layer normalization in action, our primary hypothesis is that the absence/presence thereof, and specific implementation if present, played a role. While all three models eventually converged to a stable perplexity (PPL) value, the pre- and post-norm models displayed the sharpest drops in model PPL at the start of training.

However, the post-norm model quickly diverges from its pre-norm counterpart between 1,500-2,500 steps, already exhibiting a slower convergence. This surprised us, as [4] found that the pre-norm model lags behind its post-norm counterpart with relatively shallow architectures, only beginning to surpass the latter when the number of layers increases (optimal at $L = 20$). Our trained transformer has 4 and 1 layers in the encoder and decoder, respectively — might this begin to approach the boundary between shallow and deep(er)?

Regardless of the answer, it is worth mentioning that [4] attribute the better performance of the pre-normed transformer (with a deep network) to the vanishing gradient problem, where the gradients approach 0 the deeper the network is. Perhaps the vanishing gradient problem becomes salient even with 4/1 encoder/decoder layers.

What also surprised us is that the baseline model begins to surpass the post-norm model at around 5,000 steps and is neck-to-neck with the pre-norm model at around 15,000 steps. We are not entirely sure of any model-intrinsic reasons for this, but we note that pre-processing factors such as tokenization, BPE encoding (or lack thereof), etc. could have played a role.

In what way does our setup differ from [4]? How could that have influenced our results?

Our training setup included a smaller batch size (2046 vs. 4096), fewer updates (100 vs. 500,000-3,000,000), smaller dataset (subset of 100,000 vs. 400,000+), absence of dropout (0 vs. 0.1), and fewer layers (5 vs. 30, their best-performing). They also had significantly increased computational power with 8 GPUs. In theory, these parameters and hyperparameters pose disadvantages compared to the setup of [4] in terms of quality of data, computational power, and model architecture.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. *Layer Normalization*. 2016. arXiv: 1607.06450 [stat.ML].

[2] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[3] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[4] Qiang Wang et al. *Learning Deep Transformer Models for Machine Translation*. 2019. arXiv: 1906.01787 [cs.CL].