ECSE 484: Optimizing Chinese Translation Deep Learning Models in Production

Joseph Chen (jxc1598@case.edu)

Benson Jin (bxj155@case.edu)

https://github.com/jchen42703/chinese-translation-api

Introduction

Neural networks are computationally expensive to run. Not everyone can afford to rent expensive AWS GPU or Google Cloud TPU instances to run their freshly trained models in production to promise low latency. This especially holds true when you're trying to use your model to serve millions of users.

CPU instances are much, much cheaper to host and consequently, are more scalable.

The goal of this project is to test the effect of different techniques on model inference speed and performance on AWS CPU instances and calculate the most cost efficient approach.

Experimental Design

Our base model for chinese to english translation was `BERT`, which is a widely used model in production for natural language processing applications.

We decided to test if dynamic quantization would make BERT "perform better" in production. We quantified that "performance" in terms of:

- inference speed
- BLEU (how well the models translate the chinese text)
    - We chose BLEU over "better" metrics, such as SacreBLEU, because BLEU one of the most widely used metrics and it was used by the original pretrained BERT.
- speed in production

The dynamic quantization was done through pure PyTorch. Originally, we also planned on experimenting with operation fusion quantization, but due to time constraints, we could not get a working version out.

Evaluation Process

Inference speed and BLEU were fairly easy to calculate. Since we were using a pretrained BERT for chinese to english translation (https://huggingface.co/Helsinki-NLP/opus-mt-zh-en), we wanted to avoid data leakage. Hence, we used the same test set that the pretrained model used (https://github.com/jchen42703/chinese-translation-api/tree/main/server/test_data). We predicted all of the example chinese strings and compared them to their labeled reference to calculate the BLEU (both sentence-wise BLEU and corpus-wise BLEU). We also benchmarked the inference speeds to give us a sneak peek to production performance.

To test the performance in production, we built an API with FastAPI and PyTorch. We built a CLI to automatically deploy the api to an AWS EC2 instance as a Docker container. The docker container was

then served to the public behind an nginx reverse proxy (http only for simplicity) We then load tested the api with Locust.

We also employed specific performance boosting techniques such as:

- Increasing the number of worker connections in nginx to allow for increased traffic
- Running `uvicorn` with 2 workers instead of 1
- Setting the number of Pytorch threads to 1 (This has been shown to be better to prevent conflicts with `uvicorn` by the Roblox engineering team).

We tested with a `t2.large` AWS EC2 instance.

Results: Evaluation (BLEU)

Sentence BLEU

At first, we decided to evaluate the BLEU of each sentence and predicted translation of said sentence by each model and then average the BLEUs.

The following evaluation pipelines were run locally on the same machine with 4 workers and 10,000 test samples.

For the base pretrained BERT:

```
(base) joseph@joseph-ideapad-330:~/Coding/ml_projects/chinese-translation-api/server$ poetry run python chinese_trans
lation_api/scripts/evaluate.py
Loaded default model
40000
Removed whitespace:  30000
Num test samples:  10000 10000
Example batch:
 汤姆不会同意的 Tom isn't going to give me that.
load_dset: memory before: 955,633,664, after: 960,888,832, consumed: 5,255,168; exec time: 00:00:00
Current Time = 11:31:08
10000it [52:34,  3.17it/s]
Average BLEU:
 0.2281195551728802
evaluate: memory before: 960,888,832, after: 871,190,528, consumed: -89,698,304; exec time: 00:52:34
After Evaluation = 12:23:43
Duration:  0:52:34.390733
```

For the quantized BERT:

```
(base) joseph@joseph-ideapad-330:~/Coding/ml_projects/chinese-translation-api/server$ poetry run python chinese_trans
lation_api/scripts/evaluate_quantized_dynamic.py
Loaded default model
Loaded quantized model
40000
Removed whitespace:  30000
Num test samples:  10000 10000
Example batch:
 汤姆不会同意的 Tom isn't going to give me that.
load_dset: memory before: 830,078,976, after: 834,265,088, consumed: 4,186,112; exec time: 00:00:00
Current Time = 12:32:11
10000it [35:39,  4.67it/s]
Average BLEU:
 0.22147732021415314
evaluate: memory before: 834,265,088, after: 1,112,158,208, consumed: 277,893,120; exec time: 00:35:39
After Evaluation = 13:07:50
Duration:  0:35:39.356969
```

The average sentence BLEU of the quantized model was only slightly less (0.221), but the model evaluation was almost 15 minutes faster! That's approximately a 32% increase speed for only a 3%

decrease in performance. This was a very good sign that the quantized model would be better suited for production purposes.

Corpus BLEU

You can immediately tell that the BLEU is quite low (< 0.3). The official BLEU for our pretrained model is 0.356 (https://github.com/Helsinki-NLP/Tatoeba-Challenge/tree/master/models/zho-eng). Hence, we felt that our evaluation strategy was incorrect. Upon further research, we decided to try a corpus wide BLEU evaluation. That means we accumulate all of the references and predictions and evaluate the BLEU all at once. This approach is actually the recommended way for calculating BLEU (since it was designed to be a corpus-wide evaluation metric).

(Ignore the intermediate BLEU scores, those were only corpus BLEUs calculated for sanity purposes)

For the base pretrained BERT:

```
(base) joseph@joseph-ideapad-330:~/Coding/ml_projects/chinese-translation-api/server$ poetry run python chinese_trans
lation_api/scripts/evaluate.py
Loaded default model
40000
Removed whitespace:  30000
Num test samples:  10000 10000
Example batch:
 汤姆不会同意的 Tom isn't going to give me that.
load_dset: memory before: 958,013,440, after: 962,363,392, consumed: 4,349,952; exec time: 00:00:00
Current Time = 15:30:10
1999it [09:09,  3.60it/s]
BLEU at 2000: 0.319857674918382
3999it [17:50,  4.76it/s]
BLEU at 4000: 0.3159453222138197
5999it [26:13,  2.91it/s]
BLEU at 6000: 0.3235646312706778
7999it [34:11,  3.70it/s]
BLEU at 8000: 0.33045977428070483
10000it [43:44,  3.81it/s]
Results:
 {'bleu': 0.31157213663675326, 'precisions': [0.579178024427343, 0.3853655077037641, 0.2745629897528632, 0.1988208764
6371613], 'brevity_penalty': 0.9378025057709101, 'length_ratio': 0.9396589542206184, 'translation_length': 66401, 're
ference_length': 70665}
evaluate: memory before: 962,363,392, after: 789,798,912, consumed: -172,564,480; exec time: 00:43:46
After Evaluation = 16:13:56
Duration:  0:43:46.159350
```

For the quantized BERT:

```
lation_api/scripts/evaluate_quantized_dynamic.py
Loaded default model
Loaded quantized model
40000
Removed whitespace:  30000
Num test samples:  10000 10000
Example batch:
 汤姆不会同意的 Tom isn't going to give me that.
load_dset: memory before: 826,585,088, after: 830,763,008, consumed: 4,177,920; exec time: 00:00:00
Current Time = 14:43:58
999it [03:09,  5.13it/s]
BLEU at 1000: 0.30869673388001173
1999it [06:02,  5.45it/s]
BLEU at 2000: 0.31777635591719344
2999it [08:59,  5.41it/s]
BLEU at 3000: 0.31390909250008153
3999it [12:57,  6.39it/s]
BLEU at 4000: 0.3106519734708974
4998it [15:51,  6.22it/s]
BLEU at 5000: 0.3134550341904854
5999it [18:36,  4.04it/s]
BLEU at 6000: 0.3192151622639447
6999it [21:29,  4.96it/s]
BLEU at 7000: 0.32367012534320816
7999it [24:25,  5.68it/s]
BLEU at 8000: 0.32609569241313785
8999it [27:49,  3.56it/s]
BLEU at 9000: 0.3207901436054642
10000it [31:26,  5.30it/s]
Results:
 {'bleu': 0.3053781150114456, 'precisions': [0.5693735637329513, 0.3752587083874136, 0.26554200484426127, 0.190604780
49302176], 'brevity_penalty': 0.9469763561713301, 'length_ratio': 0.9483336871152621, 'translation_length': 67014, 'r
eference_length': 70665}
evaluate: memory before: 830,763,008, after: 1,123,422,208, consumed: 292,659,200; exec time: 00:31:28
After Evaluation = 15:15:26
Duration:  0:31:28.064072
```

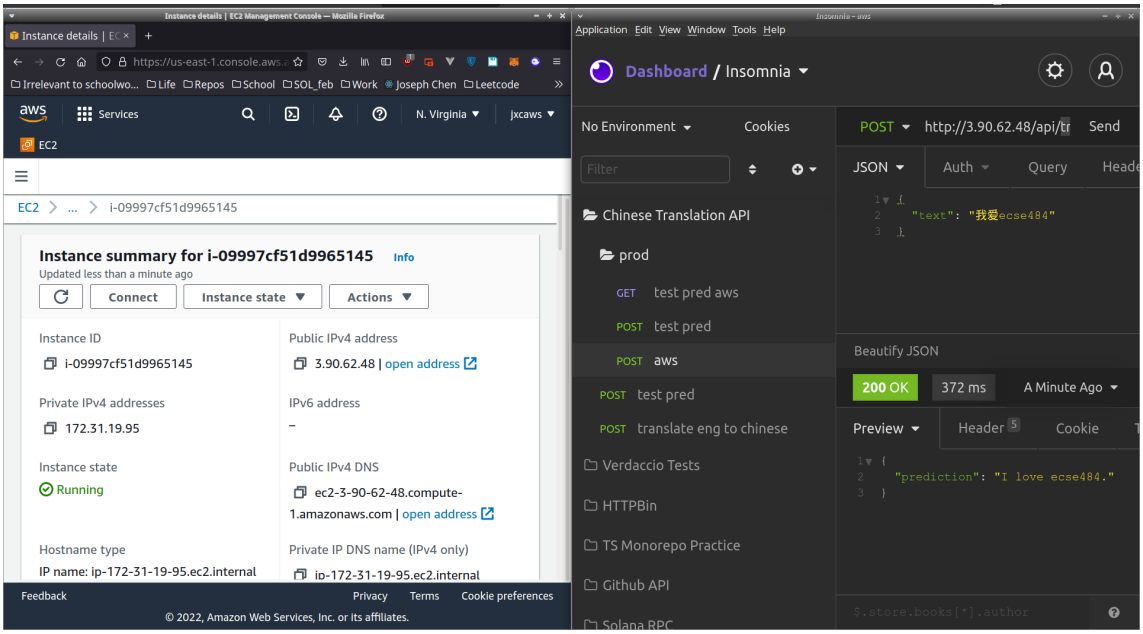
Again, you can see that the quantized model only marginally worse (~3% worse) than the base model, but was much quicker (more than 25% faster than the regular BERT).

A more detailed report of our evaluation blunders and eventually fixes are documented here: https://github.com/jchen42703/chinese-translation-api/blob/main/server/chinese_translation_api/scripts/EVALUATION_REG.md.

Results: Load Testing an API in Production

We deployed our FastAPI api to AWS EC2 instances through CLI and Docker. Requests were served through an nginx reverse proxy. The specific aws and deploy scripts are located in the `/server/aws` directory and nginx configurations are located in the `/server/nginx` directory.

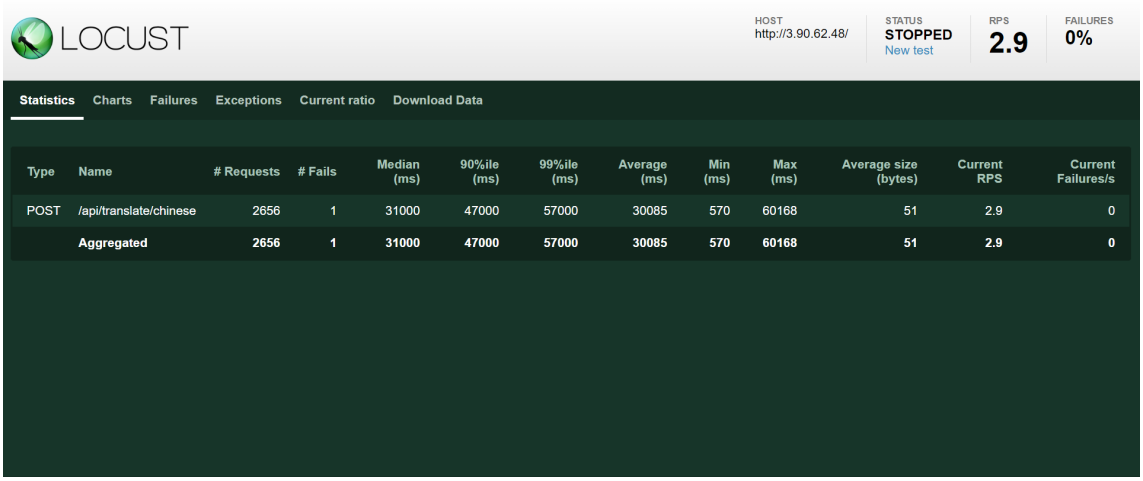
The API serves POST requests like:

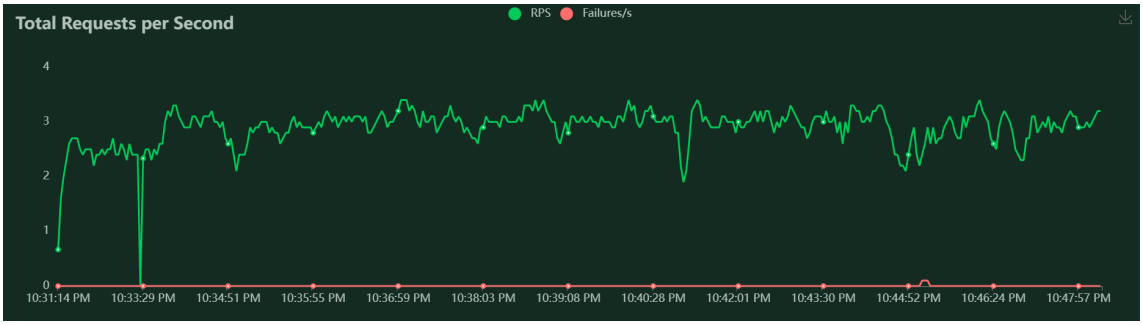


We load tested two versions of the API separately (one with the base BERT and one with the quantized model) with Locust.

Our locust configuration is located in the `server/loadtest` directory. For the experiments below, we mimiced the traffic of 100 users over the course of 15 minutes. The statistics and metrics from the Locust dashboards are shown below.

No Quantization



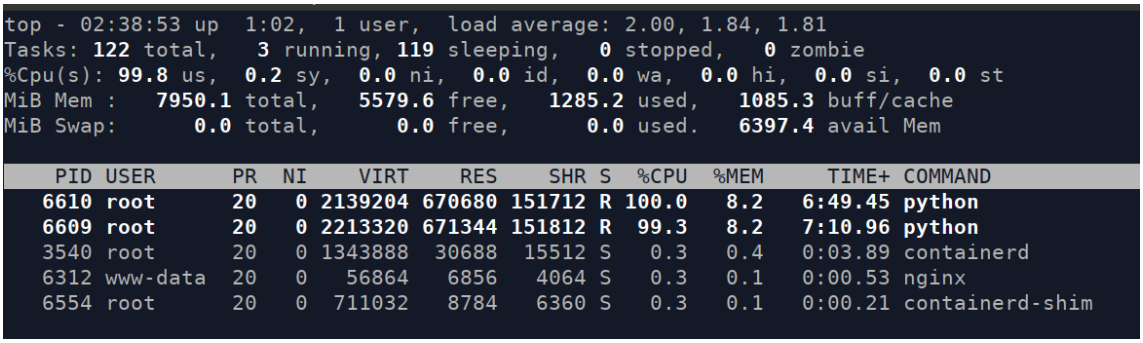


With Quantization

Observations

The base `BERT` only served `2656 requests /(15 minutes * 60 seconds) = 2.95` requests per second on average, while the quantized model served `5406 requests /(15 minutes * 60 seconds) = 6` requests per second on average. This was a more than 100% increase in performance by using the quantized model. This makes sense since the quantized model is much smaller than the base model (200mb vs 297 mb and 33 million parameters vs 119 million parameters).

Conclusions and Further Research



Although our experimental results were successful, the overall performance of our API was subpar. With only one user, the latency of the API was 500ms. With over 100 users, the latency was around 30 seconds for the base model and 13 seconds for the quantized model. In production, this is really slow (even without any caching).

We suspect that a reason for this lack of performance is due to us using a fairly low spec'd server (t3.large only has 8 GB of RAM with 2 cores). You can even tell by the screenshot above that both workers are blasting the CPUs at 100%, but the actual total memory consumed is only 16%. We could look towards load testing on a heavier server that has more cores to leverage the extra RAM.

Nevertheless, we were still able to build a quantized model that was able to perform quicker than the base model for a relatively negligible loss in performance. This performance held up both in evaluation scripts and in production.

P.S. Feel free to play around with our models @ https://chinesetranslationapi.com/.

- It's a much smaller server because we're broke college students, but it works.