Case Study: Accelerating Transformer Classifiers

The transformer ML architecture is ubiquitious in text-based machine learning. It's also sometimes used in image-based machine learning.

In this blog post, I demonstrate how you can use `moco` to make a `TinyBERT` classification model run faster in inference.

We started with pulling a dataset and model from Huggingface. The model's base model is `TinyBERT` and is fine-tuned on the IMDB Dataset.

Dataset

Dataset: Stanford NLP -- IMDB

This dataset contains movie reviews and the task is to predict whether each movie review is a positive or negative review (a sentiment analysis task). Both the train and the test set contain 25000 samples.

Model

Model: Pretrained Model (fine-tuned BERT tiny on IMDB dataset)

Initially, this model gets 91.5% accuracy on this dataset, and takes 31.283 seconds to evaluate over the entire dataset of 25000 samples.

Accelerate it

Methods

We extract intermediate representations from the `TinyBERT` model.

The architecture of this model is

- Embedding
- Encoder (contains 2 BERTBlocks)
- Pooler
- Classifier

Each BERTBlock is made up of an attention layer, an intermediate layer and then an output layer.

Specifically, we extract representations from after the embedding layer, the first BERTBlock and the second BertBlock. Within each BERT block, we extract embeddings from after the attention layer and after the output layer.

The following visual is of the k-nearest-neighbor graph of the flattened version of the output of the attention representation extracted from the first layer. We see that there is *clear separation* between the two classes, leading us to believe that there is the opportunity to spot easy-to-classify data and early-exit the model at this point.

Following this approach, we programmatically build a rule out of this embedding. The way the model now works is the following:

- it runs the embedding step
- it runs the *first* attention layer step
- now given the output of the attention layer, it has a decision to make: the easy-data-classifier makes a prediction. If the prediction is 1 then by design, the predicted class of the model is the value associated with that rule, so the model early exits and predicts that value. If the predicton is 0, the rest of the model is run as normal.

```
from moco.ml_frame import MLFrame
from moco.torch_early_exit_models import EarlyExitTextClassificationModel

tiny_bert = pipeline("text-classification", "arnabdhar/tinybert-imdb")

# Wrap the model.
eetcm = EarlyExitTextClassificationModel(tiny_bert.model)

# Load an MLFrame -- a version of a pd.DataFrame with support for multi-
dimensional arrays.
frame = MLFrame.load_frame('imdb/out.frame')

# Pre-computed -> it's N X S X D shape.
```

```
embedding = frame['bert.encoder.layer.0.attention']

engine = AnalyticsEngine.from_2d_embedding(embedding, {'predictions':
frame['predictions']})
rule = engine.compute_linear_sufficient_rule('predictions', 1)
eetcm.add_rule(rule.condition, rule.metadata['class'], 'encoder', 0,
'attention')

# Now, run the model as you would a standard Pytorch nn.Module
inputs = tokenizer("This is a test", return_tensors = 'pt')
eetcm(**inputs)
```

Results

With a batch size of 64, we observe total time of running the entire dataset through the mdoel with acceleration of 24.561 seconds. The baseline is 31.283 seconds. This is a **-21.5% reduction** in latency.

Given 25000 samples, we observe 786 QPS at baseline and an improved 1018 QPS in the accelerated version or a **29.5% increase** in throughput.

On the test set, the initial accuracy was 91.52%, and now the accuracy is 91.524%.

Conclusion

- We see clearly that `moco` can immediately accelerate **TinyBERT classifiers** for edge and at-scale applications, *without* risk of accuracy loss.
- We are removing unnecessary computations, which has additional implications for energy savings, implicating running BERT on an edge device.
- This is an exciting result because it suggests there is a relatively clear path to LLMs and the text generation task (thinking of next word prediction as a many-class classification task) and applying this same methodology.