Assignment3: Fine-tuning Pre-Trained Transformers

COMP542-442: Natural Language Processing

May 17, 2023

1 Overview

This assignment has two parts.

- In the first part, you will fine-tune a pre-trained model on a specialized task.
- In the second part, you will fine-tune a model on multiple tasks

1.1 (45 points) Part 1: Pretrained Transformer models and knowledge access

You'll train a Transformer to perform a task that involves accessing knowledge about the world—knowledge which isn't provided via the task's training data (at least if you want to generalize outside the training set). You'll find that it more or less fails entirely at the task. You'll then learn how to pretrain that Transformer on Wikipedia text that contains world knowledge, and find that finetuning that Transformer on the same knowledge-intensive task enables the model to access some of the knowledge learned at pretraining time. You'll find that this enables models to perform considerably above chance on a held out development set.

The code you're provided with is a fork of Andrej Karpathy's minGPT. The "GPT" in minGPT refers to the Transformer language model of OpenAI, originally described in this paper [2].

You'll need around 3 hours for training, so budget your time accordingly! We have provided a sample Colab with the the commands that require GPU training. Note that dataset multi-processing can fail on local machines without GPU, so to debug locally, you might have to change num workers to 0.

Your work with this code base is as follows:

- (0 points) **Check out the demo.**

  In the `mingpt-demo/` folder is a Jupyter notebook `play_char.ipynb` that trains and samples from a Transformer language model. Take a look at it (locally on your computer) to get somewhat familiar with how it defines and trains models. Some of the code you're writing below will be inspired by what you see in this notebook.

  Note that you do not have to write any code or submit written answers for this part.

- (0 points) **Read through `NameDataset` in `src/dataset.py`, our dataset for reading namebirthplace pairs.**

  The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. We'll think of this as a particularly simple form of question answering:

  Q: Where was [person] born?

  A: [place]

From now on, you'll be working with the src/ folder. The code in mingpt-demo/ won't be changed or evaluated for this assignment. In dataset.py, you'll find the the class NameDataset, which reads a TSV (tab-separated values) file of name/place pairs and produces examples of the above form that we can feed to our Transformer model.

To get a sense of the examples we'll be working with, if you run the following code, it'll load your NameDataset on the training set birth places train.tsv and print out a few examples.

```
python src/dataset.py namedata
```

Note that you do not have to write any code or submit written answers for this part.

- (0 points) **Implement finetuning (without pretraining).**

  Take a look at `run.py`. It has some skeleton code specifying flags you'll eventually need to handle as command line arguments. In particular, you might want to pretrain, finetune, or evaluate a model with this code. For now, we'll focus on the finetuning function, in the case without pretraining.

  Taking inspiration from the training code in the play char.ipynb file, write code to finetune a Transformer model on the name/birthplace dataset, via examples from the `NameDataset` class. For now, implement the case without pretraining (i.e. create a model from scratch and train it on the birthplace prediction task from part (b)). You'll have to modify two sections, marked [part c] in the code: one to initialize the model, and one to finetune it.

  Use the hyperparameters for the Trainer specified in the `run.py` code.

  Also take a look at the evaluation code which has been implemented for you. It samples predictions from the trained model and calls `evaluate_places()` to get the total percentage of correct place predictions. You will run this code in part (d) to evaluate your trained models. This is an intermediate step for later portions, including Part d, which contains commands you can run to check your implementation. No written answer is required for this part.

- (10 points) **Make predictions (without pretraining).**

  Train your model on birth places `train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

  Train on the names dataset
  ```
  python src/run.py finetune wiki.txt \
  -writing_params_path model.params \
  -finetune_corpus_path birth_places_train.tsv
  ```
  Evaluate on the dev set, writing out predictions
  ```
  python src/run.py evaluate wiki.txt \
  -reading_params_path model.params \
  -eval_corpus_path birth_dev.tsv \
  -outputs_path nopretrain.dev.predictions
  ```
  Evaluate on the test set, writing out predictions
  ```
  python src/run.py evaluate wiki.txt \
  -reading_params_path model.params \
  -eval_corpus_path birth_test_inputs.tsv \
  -outputs_path nopretrain.test.predictions
  ```

  Training will take less than x minutes. Report your model's accuracy on the dev set (as printed by the second command above). We also have Tensorboard logging for debugging. It can be launched using `tensorboard -logdir expt/`. Don't be surprised if it is well below 10%; we will be digging into why in Part 3. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birth place for everyone in the dev set. Fill in `london_baseline.py` to calculate the accuracy of that approach and report your result in your write-up. You should be able to leverage existing code such that the file is only a few lines long.

- (20 points) **Define a span corruption function for pretraining.**

  In the file `src/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption is explored in the T5 paper [3]. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you'll implement a simplification that only masks out a single sequence of characters.

  This question will be graded via autograder based on whether your span corruption function implements some basic properties of our spec. We'll instantiate the `CharCorruptionDataset` with our own data, and draw examples from it.

  To help you debug, if you run the following code, it'll sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

  ```
  python src/dataset.py charcorruption
  ```

  No written answer is required for this part.

- (15 points) **Pretrain, finetune, and make predictions. Budget 2 hours for training.**

  Now fill in the pretrain portion of run.py, which will pretrain a model on the span corruption task. Additionally, modify your finetune portion to handle finetuning in the case with pretraining. In particular, if a path to a pretrained model is provided in the bash command, load this model before finetuning it on the birthplace prediction task. Pretrain your model on wiki.txt (which should take approximately two hours), finetune it on NameDataset and evaluate it. Specifically, you should be able to run the following four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

  Pretrain the model
  ```
  python src/run.py pretrain wiki.txt \
  -writing_params_path pretrain.params
  ```
  Finetune the model
  ```
  python src/run.py finetune wiki.txt \
  -reading_params_path pretrain.params \
  -writing_params_path finetune.params \
  -finetune_corpus_path birth_places_train.tsv
  ```
  Evaluate on the dev set; write to disk
  ```
  python src/run.py evaluate wiki.txt \
  -reading_params_path finetune.params \
  -eval_corpus_path birth_dev.tsv \
  -outputs_path pretrain.dev.predictions
  ```
  Evaluate on the test set; write to disk
  ```
  python src/run.py evaluate wiki.txt \
  -reading_params_path finetune.params \
  -eval_corpus_path birth_test_inputs.tsv \
  -outputs_path pretrain.test.predictions
  ```

  Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10 precent, and will expect a similar accuracy on the held out test set.

1.2 (55 points) Part 2: Multi-Task Learning with Transformers: Transformers with Multiple Prediction Heads

Hugging Face has been building a lot of exciting new NLP functionality lately. The newly released NLP provides a wide coverage of task data sets and metrics, as well as a simple interface for processing and caching the inputs

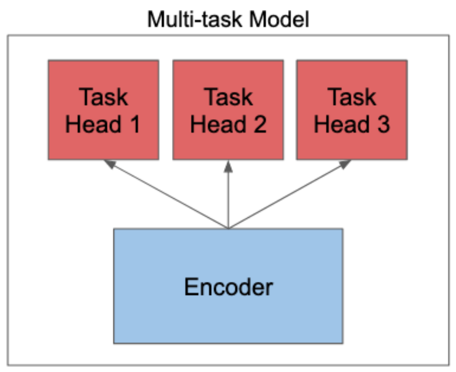

Figure 1: Single BERT model with multiple prediction heads for each task

extremely efficiently. Huggingface also has a Trainer class to the Transformers library that handles all of the training and validation logic.

To showcase our multi-task functionality, we will choose tasks of different formats:

- STS-B: A two-sentece textual similarity scoring task. (Prediction is a real number between 1 and 5)
- RTE: A two-sentence natural language entailment task. (Prediction is one of two classes)
- Commonsense QA: A multiple-choice question-answering task. (Each example consists of 5 seperate text inputs, prediction is which one of the 5 choices is correct)

In particular, notice that unlike STS-B and RTE, Commonsense QA consists of feeding multiple inputs into the transformer model. Many other tasks have weirder formats too, so our setup needs to be flexible enough to accomodate very different kinds of tasks.
We can simply call the datasets.load_dataset method, which automatically downloads the data and prepares it for use.

Please install the requirements from the `requirements.txt` file using `pip install -r requirements.txt`. Then run `pip install datasets "dill<0.3.5"`. You should be good to go after this.

Your work with this code base is as follows:

- (15 points) **Implementing the Multi-task Model in the file multitask_model.py**

  In this part, we will focus on building a multi-task training scheme. We will use a pre-trained transformer model and fine-tune it on multiple tasks. Typically, a multi-task model in the age of BERT works by having a shared BERT-style encoder transformer, and different task heads for each task.

  We could try to implement this directly in code, but there are two downsides to this approach:

  1. Hugging Face's Transformers has implementations for single-task models, but not modular task heads. This means we will need to do a lot of our own leg work to write our own task heads.
  2. This format assumes that the input is processed the same way in the encoder for every task. Already, Commonsense QA is problematic for this approach, since it requires the encoder to process multiple input sequences for a single example. Other tasks may similarly break this abstraction.

  Instead, we are going to do something very different. We are going to create separate models for each task, but we are going make them share the same encoder as in figure 2

  This will serve the same goal as having the encoder be jointly trained across multiple tasks, but still retain the independent implementations of each model. As such, we can use the existing task-model implementations in Transformers, such as `RobertaForSequenceClassification` and `RobertaForMultipleChoice`.

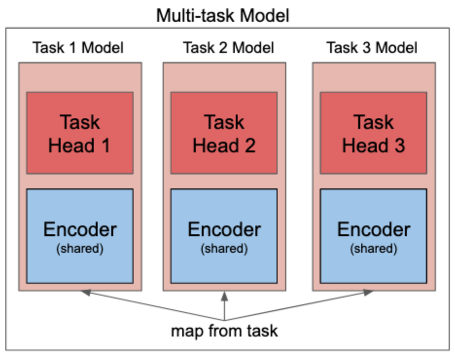

Figure 2: Separate BERT model for each task with shared encoder

Importantly, the shared encoder ensures that during training, all updates will update the same encoder weighs, and also does not consume any additional GPU memory.

- (20 points) **Processing the Data in the file preprocess.py**

  We created a dictionary of our task-specific datasets in the beginning, but we need to do a little more work to convert the respective task data into model inputs.

  We'll start by first getting the tokenizer corresponding to our model.

  - Both STS-B and RTE and two-sentence input tasks, so we will concatenate them with the corresponding special tokens. (The tokenizer's `batch_encode_plus` method handles this for us.)

    The input looking like this:

    So, the input might look like the following after processing:

    $\boxed{['<s>','This','is','my','premise','.','</s>','</s>','This','is','my','hypothesis','.','</s>']}$

  - CommonsenseQA, is a multiple choice task. A single example consists of a question, a five possible answer choices. We will feed the model inputs concatenated like QUESTION + CHOICE_1, QUESTION + CHOICE_2 and so on.

  For that we define functions specific to each dataset that convert a batch of data for that dataset into a format that the model can use. You may analyze the data present in each of the huggingface arrow datasets and accordingly complete the functions in `preprocess.py`

  Now that you have defined the above functions, you can use dataset.map method available in the `datasets` library to apply the functions over our entire datasets. The `datasets` library handles the mapping efficiently and caches the features.

  As a recap:

  - You have created our multi-task model by fusing several single-task Transformer models
  - We have created a dictionary of featurized inputs for each of our tasks, using datasets library

  Next up, you need to

  1. Set up the data loading
  2. Set up the Trainer
  3. Start training!

- (15 points) **Preparing a Multi-task Dataloader and Trainer**

  Setting up a multi-task data loader should be simple in principle - we simply need to sample from multiple single-task data loaders with some probability, and feed each batch to the multi-task model above. Of course, along with each batch, we also need to tell the model what task it is for, so `MultitaskModel` knows to use the right corresponding task-model.

  However, because we want to use the built-in `Trainer` class in Transformers, this gets a little tricky, since the `Trainer` expects a single data loader, and expects a very specific format of per-batch data. This slice of code is somewhat of a hack around that constraint.

  We need to define a `MultitaskDataloader` that combines several data loaders into a single "data loader" - not so different from our multi-task model above! This `MultitaskDataloader` should do what we described: sample from different single-task data loaders, and yield a task batch and the corresponding task name (we're going to add the `task_name` to the batch data).

  We will also need to override the `get_train_dataloader` method of the Trainer to play well with our `MultitaskDataloader`. We do this with a `MultitaskTrainer`.

  You may analyze `multitask_data_collator.py`. `DataLoaderWithTaskname`, `MultitaskDataloader` and `MultitaskTrainer` have already been implemented for you.

  For collating the data with the dataloader, you need to extend the functionality of the `DefaultDataCollator` class. You need to complete the `__call__` function in `NLPDataCollator` class in `multitask_data_collator.py`. It takes the lists of values from the arrow datasets and converts them to tensors to be returned by the dataloader (Uses the default data collator if the data is not an arrow dataset batch)

- (0 points) **Training**

  Now it is time to start training.

  Run the following command:

  Train the model
  ```
  python main.py \
  -model_name_or_path="roberta-base" \
  -train_batch_size=8 \
  -output_dir=output \
  -num_train_epochs=3
  ```

- (5 points) **Evaluate on all tasks**

  Now, you can evaluate your multi-task model on all three tasks. In this case, you can simply use single-task data loaders, since we are evaluating each task individually.

  You should expect scores of approximately:

  - RTE: 0.74
  - STS-B: 0.89/0.89
  - Commonsense QA: 0.60