Giff Gaff

Q1. When was the last time you picked up a new technical skill? What was it? How did you approach learning?

The last time I picked up a new technical skill was two days ago. It was learning about how to set up systemd timers: a specific on-premise deployment for a client required certain services to run at specific times. To learn these I followed a basic tutorial. After reading through the documentation, I tried to impelment the examples by my self.

In the last two months I have also learnt dagster, which required a lot more effort to learn and use. In this case, I first looked over the docs and followed the basic tutorials. Secondly I moved on to the more advanced examples provided by dagster. I then re-implemented one of them myself and slightly expanded on it by adding an extra example feature: https://github.com/TBourton/nyt-feed-dagster-example. Afterwards, I tried to apply what I had learnt to my specific problem, and I learnt a lot by doing, reading the docs, and trying things out.

Q2. How do you tackle a new codebase?

My first objective when tackling a new codebase would be to find the documentation for it, firstly looking in the obvious places e.g. from the repo README or by asking a colleague. I would be looking to extract the main purpose of the code, how to set up the repo and how to run the unit tests.

Secondly, I would also schedule a short meeting in the future with a colleague who knows the codebase fairly well. Before that meeting I would make sure to create a local development environment, e.g. python virtual environment, where I can run sucessfully run the unit tests. With that set up, I find it very useful to step through the 'happy' paths of the unit tests for the main entrypoints of the code using an interactive debugger, e.g. `pdb`. I would be aiming to get as much context and understanding of the codebase before the meeting in order to focus and bring any questions along.

Q3. Can you explain what GIL in Python is? What are some implications for parallel processing?

In short, the Python GIL is a lock that allows only a single thread to hold execution of the Python interpreter at any given time.

Python uses reference counting for memory management. It keeps track of the number of references to each object. A reference count for a given object is a positive integer or zero. Once the number of references drops to zero, Python's garbage collector "frees" that block of memory held by the given object.

However, the operations of modifying the reference counts is not thread safe. For example, it would be possible for two threads to increase and decrease a

reference count simultaneously, leading to either memory leaks (i.e. memory that is occupied by not referenced) or released memory for objects that are still referenced.

Python solves the above problem by introducing a lock on the interpreter.

A lock is a general concept in thread-based computing and they provide mechanisms to syncronise different processing threads. For example, one common use of a lock might be: if you have multiple threads trying to write to the same file, this can lead to a race condition where two or more threads try to perform a write at the same time, resulting in loss or corruption of data. This can be solved by introducing a binary lock, shared by all threads. In order for a thread to perform the write it must 'acquire' this lock, and after it has finished it 'releases' the lock. If a thread tries to acquire the lock that has already been acquired, it is forced to block until the lock has been released.

The Python GIL has many similarities to the above, however the role of the file write operation is played by executing python bytecode. Placing a lock that must be acquired by any thread wanting to perform execution ensures that reference counting is safe.

Implications for parallel processing

The GIL effectively means that any CPU-bound python programs are single threaded. This is in contrast to languages without a GIL, e.g. Java. that do support multiple threads executing in parallel.

The solution to this is to use multi-processing, for CPU-bound tasks that can be parallelised. Multi-processing uses processes instead of threads. Each process gets its own separate python interpreter, i.e. essentially creating many copies of single threaded runtimes. Python does have a version of multi-threading, however it is mostly only useful for I/O-bound tasks, due to the GIL.

Q4. When it comes to handling large models, where do you store parameters during training and how do you ship/deploy parameters?

For this question, the answer depends on the definition of 'large'. I think there's two such definitions that can fit the category: 1. A model whose parameters cannot fit on a single device. E.g. large-language-models such as GPT-X style models. 2. A model whose parameters can fit on a single device, but training takes a long time due to many parameters and a lot of data required for training.

Before coming up with any training solution one should investigate exactly which category the model belongs to, in particular the former is more complicated than the latter, and we should always solve the correct problem.

In the first case we require *model parallelism*, in this scenario the model parameters are distributed over multiple devices, i.e. GPUs. For example, Mesh

Tensorflow or SageMaker distributed could be used.

In the second case it is more suited to use *data parallelism*. In this case, many copies of the entire model are distributed over multiple devices. A single model is declared as a leader. Then, the data can be batched, and each batch sent to different devices. At the end the different parameters from each node are aggregated, and the leader models parameters updated with the aggregate. In order for this method to be efficient, the cost of a processing a batch of the data must be more expensive than the cost associated with transferring the parameters between devices.

Deploying

In both cases, a decent pattern is to store the model parameters in S3 storage. The code around the deployed model (in whatever form) can download at startup time - ideally with some level of caching in place due to the very large file sizes - and used to initilise the model for inference. Due to the large sizes of the weights, one might also consider using something like TensorRT to perform weight quantisation. Training would happen in full FP32 precision, but often for inference it's possible to quantise the model to use only INT8 or INT16. The benefit is much smaller model parameters memory footprint, at the cost of degraded performance. Another option is weight pruning - again with the same sort of tradeoffs.

In terms of the actual deployment strategy for serving the model predictions, we would have to gather information about how the model would be used in inference. Such as whether it needs to do live computing of events or could it do simpler batch processing? Generally, for the former: In case 2 deployment is more straightforward as inference can be performed on a single device allowing for more flexibility. In this case we might consider serving model behind either a custom REST API, or setting it up as a message queue consumer. Both of which could then be containerised and deployed, e.g. to a k8s cluster. In case 1 We require multiple GPUs for inference, as well as training. This might lend more towards using a pre-made tools such AWS sagemaker, in order to limit managing multiple GPUs ourselves.

Q5. How many models have you operationalised and deployed to a production environment?

I have deployed a variety of tensorflow computer-vision models (e.g. ResNets, RNNs, etc). Firstly, these were served via Flask API endpoints, dockerised and deployed on a k8s cluster for scalability. At some point, we began to hit issues of scaling due to serving large models over HTTP. I was involved, along with another colleague, in implementing a message queue system for those services.

I have also worked on automatic retraining pipelines using dagster. The pipelines would train and evaluate new models on a cron schedule, if the model evaluation

metrics were better than the old model, it would be automatically deployed. These models were also deployed on dagster, as they were used for batch processing.

Q6. What attracts you the most to Machine Learning Engineering?

My favorite part about machine learning engineering is the intersection between lots of different disciplines. For example the job requires learning about many different domains, such as data science, machine learning, dev-ops, software engineering, data engineering, etc. Therefore, the job has a large variety of tasks meaning it's easy to always be learning about new technologies and concepts. Machine learning engineering is also a relatively young field, with many companies only recently starting to understand that there's a lot of additional and specialised work that goes from taking a model from a notebook to an deployed piece of software that can provide value or function to the company. This means it's easy to play a large role in taking ownership and shaping the data culture of a company.