ChatGPT

Performance Improvement Suggestions for the Current Project

Environment and PyTorch Settings

- **GPU Driver and Power Settings:** Ensure the NVIDIA driver is up to date and GPU power management is set to maximize performance. If you have root access, enable persistence mode and set application clocks via `nvidia-smi` (this prevents throttling). For example: `sudo nvidia-smi -pm 1` and check that the GPU isn't power-capped in `nvidia-smi --query-gpu=pstate,power.limit --format=csv`. This guarantees the GPU runs at full speed during training.

- **CUDNN & TF32 Optimization:** The project already enables important PyTorch speed knobs. In the logs, we see `torch.backends.cudnn.benchmark = True` and TensorFloat-32 (TF32) is allowed on the A100 [1]. This is good – it lets cuDNN autotune for your data shapes and uses TF32 math on Ampere for faster matrix multiplies. Make sure **deterministic mode** is off (it is off by default unless you set `DETERMINISTIC=1`). The code shows non-deterministic mode is used (cudnn.deterministic = False) [2]. These settings maximize throughput. Additionally, PyTorch 2.x introduced `torch.set_float32_matmul_precision("high")` which the code is calling [1] – this ensures high precision TF32. All these are already in place.

- **PyTorch 2.x Compiler (Optional):** You might try using the PyTorch 2.x compile feature for further speed. For example, after constructing the model, do `model = torch.compile(model, mode="max-autotune")`. This can JIT-compile and optimize the model graph. It's not guaranteed to always help, but for some workloads it can improve GPU utilization. Since your environment is already PyTorch 2.x, this could be an easy win (the code attempts something similar for matmul precision, so compile is the next step).

Data Loading Pipeline

- **Enable Multi-Worker DataLoader:** A **major issue** noted in the logs is that the DataLoader is forced into single-worker mode: `[loader-guard] Forcing DataLoader into single-process mode (num_workers=0) to avoid worker aborts` [3]. This means data loading is happening on the main thread, potentially bottlenecking the GPU. By default, the config tried to use 8 workers (as seen in the command), but an environment safety guard set `num_workers=0` [4]. The likely reason was "sporadic worker aborts with multi-process parquet loading" [5]. If those issues have been resolved or can be mitigated, **try enabling multi-process data loading**. You can do this by setting `ALLOW_UNSAFE_DATALOADER=1` in the environment, which the code checks to bypass the single-process guard [4]. With 8 workers, each pre-fetching batches, you can keep the GPU fed much more consistently. Monitor CPU utilization – if enabling workers pegs the CPU or causes crashes, you might tune the number of workers (e.g. start with 4 or 8 per GPU and adjust). Since you have a powerful CPU and fast NVMe, using multiple loader workers should significantly boost throughput.

- **Persistent Workers and Prefetch:** The training command already sets `persistent_workers=True` and `prefetch_factor=4`. Once multi-workers are enabled, these help hide data loading latency.

Persistent workers keep the worker processes alive across epochs (avoiding startup overhead each epoch), and a prefetch of 4 means each worker will pre-load 4 batches ahead. These settings are in line with best practices (the snippet suggested 2–4 prefetch, you chose 4). Make sure pinning memory is also enabled (it was set to True in the logs, which is good for faster host-to-GPU transfers). With these in place, the data pipeline can better overlap with GPU training.

- **Optimize Data Decoding:** If reading Parquet with Polars/pyarrow is the source of crashes, consider alternatives to speed up data reads safely. One idea is to **convert datasets to a simpler format** that is faster for streaming. For example, if possible, writing the dataset to a single contiguous file (like **Apache Arrow IPC/Feather** or a memory-mapped Numpy array) could allow safer multi-threaded reads. The current format (multiple Parquet files per stock) might be causing GIL contention or fork issues. If conversion is feasible, a format like TFRecord or a WebDataset (tar of samples) could also be used, but those may require significant refactoring. At the very least, ensure that data is stored on local NVMe SSD (which it appears to be) rather than network storage, to maximize I/O throughput.

- **NVIDIA DALI or GPU Preprocessing:** Since your data is primarily tabular/time-series, the CPU is likely handling it fine, but if any heavy augmentations or transformations are done per batch, consider offloading them to the GPU. NVIDIA DALI is more geared towards image pipelines, but libraries like **Kornia** or **torchvision** (for image augmentation) on GPU won't apply here unless you have image-like data. For now, the main bottleneck is reading and slicing the data; focus on multi-threaded loading as above.

Mixed Precision and Batch Size

- **Automatic Mixed Precision (AMP):** You are correctly using mixed precision. The log shows `train.trainer.precision = bf16-mixed` and the environment sets `AMP_DTYPE=bf16` [6] [7]. On A100 GPUs, **bfloat16** is a great choice – it gives almost FP32-level stability with performance close to FP16. The code appropriately **disables GradScaler for BF16** (since scaling isn't needed for bfloat16) [8]. This is in line with the checklist's suggestion to use BF16 on A100/H100. So AMP is well-configured. Just ensure that anywhere you do `.to(device)` for tensors, use `non_blocking=True` when pin_memory is True (the code does this in places). That allows overlapping data transfer with computation.

- **Batch Size and Gradient Accumulation:** The effective batch size is already quite large (4096) through gradient accumulation [9]. This is good for throughput on the A100. The base per-GPU batch was 1024 with `accumulate_grad_batches=4`, yielding 4096 effective [9]. If GPU memory isn't fully utilized (and it looks like there was headroom), you could push this further. For example, you might try a per-GPU batch of 2048 with grad accumulation 2 for a similar effective 4096 but fewer steps, or even $2048 \times 4 = 8192$ effective (if memory allows). In one run, the logs show an attempt at effective batch 8192 [10]. Larger batch can improve utilization and learning stability up to a point, but watch out for diminishing returns or gradients getting too smooth. Alternatively, if you suspect the model is too stable and not learning (performance plateaus), you could **try a smaller batch** to inject a bit more gradient noise. This might help escape trivial solutions. It's a trade-off – since you have lots of data, the large batch is more for speed. Given training isn't improving, a smaller batch (e.g. 512 with more updates per epoch) might help the model find a signal, at the cost of longer training time.

- **Gradient Accumulation Efficiency:** With accumulation, ensure that any batchnorm or learning rate scheduler steps are handled correctly (they likely are, since you're using PyTorch Lightning or

a custom loop that accounts for it). The code prints the accumulation steps clearly, so it's aware of it. One thing: monitor memory usage during accumulation – the peak memory occurs after accumulating 4 mini-batches. If memory is still nowhere near the 80GB, that suggests you can increase batch or model size safely.

- **Gradient Checkpointing:** If you run into memory limits (perhaps after increasing model size), you can implement **gradient checkpointing** for certain parts of the model. The checklist snippet shows an example for sequential checkpoints. In PyTorch, `torch.utils.checkpoint` can save memory by not storing intermediate activations. You would then re-compute them on the backward pass. For instance, wrapping parts of the model with `checkpoint_sequential` could let you increase sequence length or batch size further without OOM. Currently, memory doesn't seem to be the blocker (an 80GB GPU for a 2.7M param model is plenty), so this is optional. But for future larger models, keep this in mind.

Multi-GPU Utilization

- **Distributed Data Parallel (DDP):** If you have access to multiple GPUs on this node (e.g. 4× A100), you should leverage them. The launch command `torchrun --nproc_per_node=$(nvidia-smi --query-gpu=name ... | wc -l)` suggests it will launch one process per GPU. However, I did not see an explicit `torch.distributed.init_process_group` or `DistributedSampler` in the code, which means true DDP might not be set up. **Verify that multi-GPU training is correctly implemented.** Ideally, each process should get a subset of data (via a DistributedSampler) and gradients should be all-reduced. If this isn't done, multiple processes could be redundantly training on the same data. To fix this, you can integrate DDP as follows:
- Initialize the process group at start (`dist.init_process_group(backend="nccl")` and set the GPU device per rank).
- Wrap the model with `DistributedDataParallel(model, device_ids=[local_rank])`.
- Use `DistributedSampler(dataset, shuffle=True)` for train (and val if needed) so each GPU sees a unique portion of data each epoch.
- Ensure `torchrun` is launched with the appropriate rendezvous (the script's `--standalone` flag suggests it's using c10d backend which is good for single node).

Following these steps (which align with the checklist's DDP snippet) will let you scale linearly with more GPUs. Also, set `gradient_as_bucket_view=True` in DDP to save memory (the checklist noted this). Finally, be mindful of **NCCL parameters**: if you have a high-speed network or multiple NICs, env vars like `NCCL_SOCKET_IFNAME` and `NCCL_NSOCKS_PERTHREAD` can improve throughput. On a single node with NVLink or PCIe, usually the defaults are fine, but tuning can help a bit.

- **Avoid DataParallel:** Ensure you're using DDP as described, not the old `nn.DataParallel`. DDP is more efficient and the only way to scale to multi-node if needed. The code does appear to aim for DDP (using torchrun), so just fill in any gaps in initialization as mentioned. This will maximize GPU utilization across all devices.

- **Multi-Node (Future):** If you ever need to go multi-node, you'll want to use a proper rendezvous backend (e.g., `--rdzv_backend=c10d` with a host:port) and ensure time synchronization across nodes. But for now, focus on using all GPUs in the single node effectively.

Model Architecture and Training Enhancements

- **Increase Model Capacity:** It appears the model's hidden size might be much smaller than intended. The logs show `Found hidden_size=64 at path: model.hidden_size` [11], even though the config file had hidden_size=512. This indicates that somewhere the model dimension was overridden to 64 (possibly via Hydra config or a mistake). A hidden size of 64 for the ATFT-GAT-FAN (with 189 features input) is quite low, resulting in only ~2.7 million parameters [2] [12]. This could severely underfit the data. Consider **increasing the hidden dimensions** and number of layers if possible. For example, using hidden_size 256 or 512 (and correspondingly larger internal dims for the TFT and GAT components) will give the model more capacity to learn complex patterns. The A100 has plenty of memory to handle a larger model. This change alone might improve training performance (since a larger model can capture more signal, if it exists, at the cost of speed). Keep an eye on training loss vs. validation: if training loss is much higher than validation (after normalization), it means underfitting – bigger model can help. If training loss is far lower than val, then overfitting – bigger model would worsen that, so apply regularization if needed. Currently, train and val metrics are both flat, suggesting underfitting or no signal, so a capacity boost is worth a try.

- **Feature Configuration Mismatch:** Address the warnings about feature counts. The model code expects around 59 "current" features (perhaps they categorized features into basic/technical/etc.), but it found 189 and treated many as unspecified [13]. The log warning "Feature count mismatch! Expected ~59, got 0" indicates that none of the features were recognized under the predefined categories [14]. All features ended up lumped as dynamic features (189). This could mean some parts of the model (like certain Variable Selection Networks for feature subsets or specialized normalizations) might not be configured properly. **Ensure the feature columns in the config match the actual dataset.** Update `config.data.features` so that `basic`, `technical`, `ma_derived`, etc., lists include the correct column names from your dataset. By doing so, the model will "know" how to split features into the groups it expects (basic, technical, returns, etc.), and the internal architecture (like FrequencyAdaptiveNorm or SliceAdaptiveNorm for those groups) will function as intended. Right now, if those lists are empty, the model might be skipping some layers or treating everything homogenously, which could hurt performance. Double-check **ML_DATASET_COLUMNS.md** against your dataset and update the config accordingly. Once this mismatch is resolved, rerun training – it may improve the model's ability to learn from each feature type properly.

- **Loss Function and Objective Tuning:** One reason your validation metrics (Sharpe, IC, RankIC) are flat could be the training objective. Initially, the model is optimizing primarily a **quantile loss** (likely the pinball loss for median, since quantile weight = 1.0 in Phase 0) and not directly optimizing correlation or Sharpe in early phases [15] [16]. This can lead to a trivial solution: predicting the median of the target distribution for every stock, which minimizes quantile error but yields no cross-sectional signal (hence IC ~ 0). To combat this:

- **Increase the weight on Sharpe/IC loss terms:** In Phase 1 and beyond, a small weight (0.1 or 0.05) was used for Sharpe and correlation losses. You might experiment with boosting these weights. For example, set Sharpe loss weight to 0.5 and corr (IC) weight to 0.2 in the later phase, to explicitly push the model to improve those metrics. Be cautious – too high a weight might destabilize training (since correlation has a noisier gradient), but finding a better balance is key. The code has environment variables `USE_CS_IC` and `CS_IC_WEIGHT` (default 0.05) [17]. Try setting `CS_IC_WEIGHT=0.1` or higher, and similarly if there's a `SHARPE_WEIGHT` toggle, increase it.

- **Direct RankIC Optimization:** There is a flag for `USE_RANKIC` in the code (with default off) [18]. Enabling this (`USE_RANKIC=1`) adds a rank-correlation term to the loss. This could directly improve RankIC metric if that's crucial. The default weight for rank IC loss was 0.5 (from the code) which is fairly high; you can start with that and observe if the model starts picking up signal (even at the cost of a slight increase in quantile loss).
- **Phased Training Strategy:** The idea of phase-wise training (Baseline -> Adaptive Norm -> GAT -> Fine-tune) is to stabilize training, but if it's not yielding improvements, you might simplify the approach. For instance, you could try training all components end-to-end from the start (enable `use_gat=True` and `use_fan/SAN=True` from epoch 1) instead of phasing them in. Sometimes, phasing can lead to the model getting stuck in a local minimum (e.g., the baseline phase finds a trivial solution and later phases can't escape it because the initial weights are already at a bad minimum). If you allow the model to use GAT and adaptive norms from the beginning (and perhaps use a smaller learning rate warmup), it might learn a slightly better starting point. This is speculative, but worth an experiment if phase training continues to stall.
- **Longer Training or Different LR Schedule:** Noticing the logs, after the first epoch or two, the validation metrics didn't improve at all – in fact, by epoch 2 of Phase 0, val loss was flat and Sharpe/IC were flat thereafter. It could be that the learning rate was decayed too quickly (cosine annealed to a low value by epoch 5 in Phase 0). You used a Warmup+Cosine schedule per phase with only 5 epochs in Phase 0, so by epoch 3-5 the LR was tiny [19] [20]. The model might not have had time at a moderate LR to actually learn signal. Consider using a **different schedule**: for example, a single cosine schedule over the entire 75 epochs (rather than resetting each phase), or using a **ReduceLROnPlateau** on validation metric. The code even has an option for `sched_choice = plateau` [19]. Using plateau scheduling would keep the LR high as long as validation isn't improving, and only reduce it when the metric stops getting better. This could squeeze more performance out. Also, you might simply train for more epochs – if the model is very capacity-limited and data-limited, more epochs won't help, but if it's not overfitting, additional training time (with a cyclic or flat schedule) could let it inch up those metrics.

- **Sharper Objective** (optional): Some practitioners turn the problem into classification or ranking to get better signals. For example, predicting the rank or sorting stocks into buckets rather than regressing exact returns can sometimes yield higher IC. If quantile/MSE losses keep giving trivial answers, you could try formulating a classification task (e.g., top-quartile vs bottom-quartile stock prediction) and see if the model learns better differentiation. This is a more involved change, but I mention it as a potential direction if current regression approach remains flat.

- **Use of Snapshots and SWA:** You enabled `SNAPSHOT_ENS=1` and `USE_SWA=1`. Snapshot ensembling will save models at certain epochs and average their predictions, and SWA (Stochastic Weight Averaging) will average weights at the end of training. These techniques can improve generalization **if** the model has learned something non-trivial. In the current runs, since each epoch was yielding nearly the same result, ensembling didn't provide a boost (it was essentially averaging similar predictions). Once you make the above changes (increasing capacity, adjusting loss focus, etc.) and start seeing the model move off the trivial solution (e.g. validation Sharpe/IC start to fluctuate up and down instead of flat zeros), then **SWA and snapshot ensembling** could help solidify gains. SWA works best if you let the model run with a constant or cycling learning rate for a while at the end to sample different weight optima. You might extend the final phase (fine-tuning phase) and keep a small but not too tiny LR, allowing the model to wander around a bit, then do SWA. This can often yield a small bump in Sharpe/IC by averaging out noise.

Monitoring and Profiling

- **Monitor GPU Utilization:** While training, run `nvidia-smi dmon -s pucm -d 1` in a separate session. This will print GPU utilization, memory usage, and PCIe traffic each second. Ideally, you want the GPU utilization (%) to be high and fairly steady. If you notice it frequently dropping to 0 while the GPU memory is still mostly filled, that indicates the GPU is waiting for data (likely the single-threaded data loader is too slow). After enabling multi-workers, this utilization should increase. Also watch the **CPU** usage (using `htop` or `vmstat 1`). If one CPU core is at 100% (the Python GIL thread), that's a bottleneck – multiple workers should spread load across cores. Also check disk I/O (`iostat -xz 1`) to ensure the data reading isn't saturating the disk bandwidth. These system metrics will guide you: e.g., if CPU is maxed but GPU is low, add more workers or reduce data processing per sample.

- **Profile the Training Loop:** It's often unclear where the bottleneck is without profiling. Use the PyTorch Profiler as in the checklist snippet to identify hotspots. You can wrap a few iterations with `torch.profiler.profile(... on_trace_ready=tensorboard_trace_handler)`. Then open TensorBoard and inspect the trace. This will show you the time spent in DataLoader vs. forward pass vs. backward, etc. For instance, you might discover that 70% of the step time is data loading (in which case, definitely the loader changes are needed), or perhaps the GAT or some custom operation is taking unexpectedly long. Given the model is not huge, you might find that the Graph Attention or normalization layers are a bit costly per iteration (which is fine if data loading is fixed). Also profile memory to ensure no leaks.

- **Check for Any Stalls or Deadlocks:** The logs don't indicate crashes, but if you ever see the training hang, using `nsys` (Nsight Systems) profiling as suggested can capture if there are synchronization issues (like all-reduce stalls, etc.). This is more for multi-GPU debugging. In single GPU, just ensure each epoch time is consistent; if an epoch suddenly takes much longer, something might have stalled (e.g., lazy dataset loading or an external data fetch, which I don't suspect here).

By implementing the above suggestions, you should see improvements in both **training throughput** and **model predictive performance**:

- Enabling multi-threaded data loading and utilizing the GPU fully will shorten epoch times and allow you to iterate faster (or train more in the same time).
- Increasing model capacity and focusing the loss function on the metrics you care about (Sharpe, IC) should give the model a fighting chance to find signal in the data. Keep an eye on those validation metrics each epoch; with more aggressive settings, they may start to move (even if slightly). That's when techniques like SWA/ensembling can further solidify gains.
- Lastly, ensure that any known issues (feature config, etc.) are fixed so you're not inadvertently training the model in a handicapped state.

With these adjustments, the training should no longer plateau at essentially random-performance. It will take some tuning, but each change above targets a specific potential problem observed: the data pipeline will no longer starve the GPU [4], the model will be large enough to model complex patterns, and the optimization will directly attempt to improve the financial metrics of interest rather than just a generic loss. Good luck, and monitor the outcome of each change (only change a couple of things at a time so you can tell what helped)!

---

1 7 8 12 15 16 17 18 19 20 train_atft.py

https://github.com/wer-inc/gogooku3/blob/81494d1fbfe1e6edeb2b7c6766b727db63c919f2/scripts/train_atft.py

2 6 environment.py

https://github.com/wer-inc/gogooku3/blob/81494d1fbfe1e6edeb2b7c6766b727db63c919f2/src/gogooku3/training/atft/environment.py

3 9 10 11 TODO.md

https://github.com/wer-inc/gogooku3/blob/81494d1fbfe1e6edeb2b7c6766b727db63c919f2/TODO.md

4 5 data_module.py

https://github.com/wer-inc/gogooku3/blob/81494d1fbfe1e6edeb2b7c6766b727db63c919f2/src/gogooku3/training/atft/data_module.py

13 14 atft_gat_fan.py

https://github.com/wer-inc/gogooku3/blob/81494d1fbfe1e6edeb2b7c6766b727db63c919f2/src/atft_gat_fan/models/architectures/atft_gat_fan.py