Technical Summary of the Program

This project implements a Transformer-based factor prediction pipeline for quantitative equity modeling, integrating data preprocessing, sequence construction, model training, and rolling prediction. The outcome of the model is the **trading signal of a portfolio**.

1. Data Processing (dataproceed.py)

This module handles raw financial data transformation. It standardizes numeric features, manages missing values, applies outlier clipping, and constructs factor tensors suitable for sequence models. It builds both *cross-sectional* and *temporal* input representations, aligning with the factor definitions used in the research. It also implements rolling-window sampling logic, allowing dynamic dataset updates as new market data arrives.

2. Model Architecture (model_utils.py)

The model utilities define the core Transformer and Attentional LSTM (ALSTM) architectures for multi-horizon return prediction. The Transformer encoder employs multi-head scaled dot-product attention with feedforward layers, layer normalization, and dropout regularization. Custom modules such as *Weighted MSE* and *IC-based loss functions* are included to directly optimize correlation between predicted and realized returns.

3. Training Pipeline (train_pip.py)

This script integrates model initialization, optimizer setup, and full-cycle training with validation and early stopping. It supports *rolling-window training* to simulate realistic out-of-sample forecasting. Model checkpoints and best-performing weights are saved automatically. The pipeline also tracks performance metrics such as RankIC, ICIR, and Sharpe ratio for continuous evaluation.

4. Main Execution and Testing (main.py, test.py)

main.py serves as the experiment entry point, controlling all configuration parameters. It orchestrates the rolling retraining procedure across trading months or weeks. test.py validates trained models on hold-out periods and generates predicted factor values for downstream portfolio construction and backtesting.

Limitations

- The model does not rebuild training sets progressively within each month's loop — therefore, it's not fully aligned with a rolling backtesting framework, making it less suitable for "timing retraining."
- Missing value handling remains debatable.

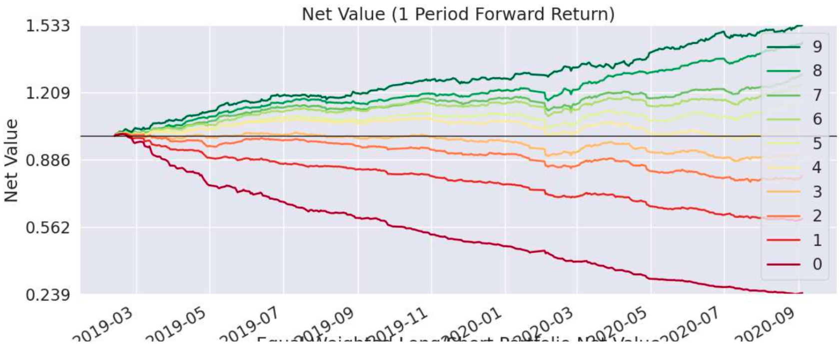
Workflow

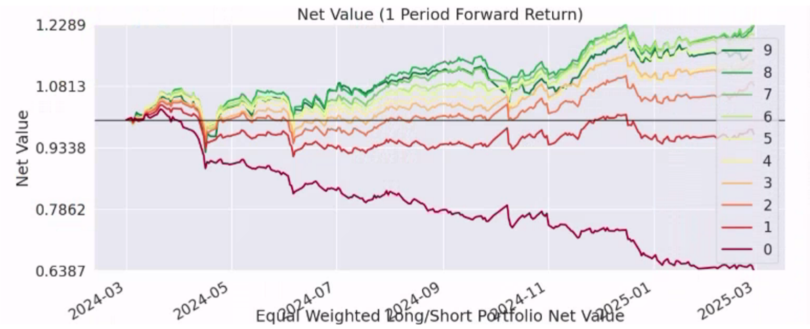
Writing new loss function

Loss function

```Python
squared_error = torch.pow(y_pred - y_true, 2)
weights = torch.exp(y_true)
weighted_loss = weights * squared_error
```

Outcome:





However, extending the backtest still showed that high ranks (top 1–2 groups) sometimes underperformed.

The exponential weighting created excessive loss gaps between G1 and G2.

So, Two versions were tested:

```
Sigmod(x) = 1 / (1 + torch.exp(-x))
    z = 40 * (x - 0.8)
  y = 1 / (1 + torch.exp(-z))
  y_fixed = 0.1 + 0.9 * y.
```

```
    z = 15 * (x - 0.5)
  y = 1 / (1 + torch.exp(-z))
  y_fixed = 0.5+ 0.5 * y.
```

The **second version performed better**, lifting the lower bound while extending the upper plateau. But after that I directly use the IC as the loss function.

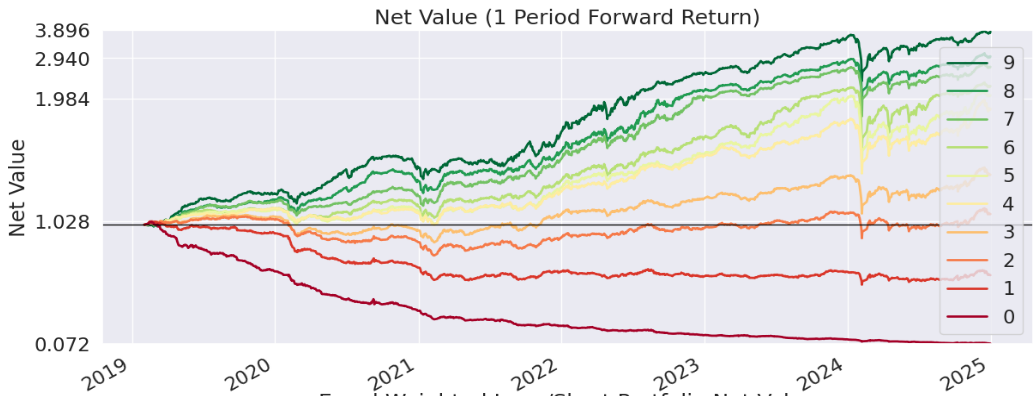
Parameter Finding

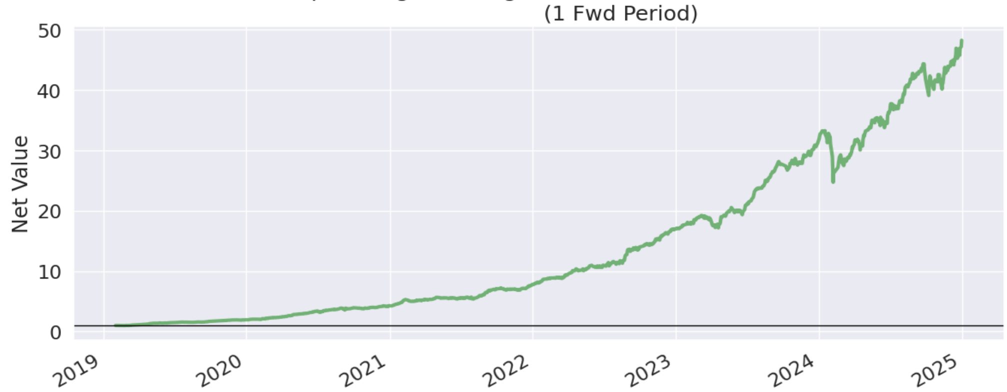
Earlier conclusions may be less reliable; see *Transformer Training* for specific backtest outcomes.

- Best lookforward = 10 days
- Best sequence length (seq) = 5 (may change for deeper networks)
- IC Loss significantly outperforms MSE Loss
- For IC Loss, larger batch sizes (1k–20k) yield more stable IC computation
- Small networks may overfit after 30 epochs (val set not fully out-of-sample)
- Longer lookforward horizons perform better

Optimal Parameters Combination(AMP enabled)

```python
model_size = 'ori'
  #amp = True
   date_start = None
   date_end   = None
   date_start = "2018-01-01"
   date_end   = "2024-12-31"
   n_epochs   = 30
   lr         = 1e-5
   patience   = 20
   valid_days = 0
   lookback_grid = 252
   seq_len    = 5
   use_amp    = True
   batch_size=20000
```





AMP experiment

While the need for FP32 master weights is not universal, there are two possible reasons why a number of networks require it. One explanation is that updates (weight gradients multiplied by the learning rate) become too small to be represented in FP16 - any value whose magnitude is smaller than 2−24 becomes zero in FP16. We can see that approximately 5% of weight gradient values have exponents smaller than −24. These small valued gradients would become zero in the optimizer when multiplied with the learning rate and adversely affect the model accuracy. Using a single-precision copy for the updates allows us to overcome this problem and recover the accuracy.

- While not all networks require FP32 master weights, many do for two main reasons:
- Updates (weight gradients learning rate) may be too small for FP16 representation.
- About 5% of weight gradients have exponents < 24 become zero in FP16 accuracy loss.
- Maintaining a single-precision copy for updates resolves this issue.

FP16 vs FP32 Computations

- Converted to FP16: nn.Linear, matmul
- Kept in FP32 (numerically sensitive):
- softmax, LayerNorm, BatchNorm, reductions (sum, mean), and loss

Speed Improvement Test (5 Epochs, 24th Month)

`seq_len=5,`

`n_epochs=5,`

`valid_days=0,`

`train_model_no_val`

`batch_sizes=(512,1024,2048,4096),`

`amp_options=(False, True),`

`repeats=3,`

`split_index=23,`

`criterion=ICLoss()`

Default model sizes:

batch_size	use_amp	repeats	avg_time_sec	std_time_sec	min_time_sec	max_time_sec
512	False	5	37.778353	0.397365	37.191360	38.427222
512	True	5	38.888162	0.310285	38.467932	39.173111
1024	False	5	44.276889	0.434376	43.649449	44.820548
1024	True	5	44.580894	0.701683	43.872169	45.877156
2048	False	5	47.029617	0.199370	46.801178	47.302885
2048	True	5	47.051206	0.352638	46.632982	47.642976
4096	True	5	58.678543	0.474201	57.896710	59.371733
4096	False	5	58.945991	0.315233	58.454034	59.314086

Medium

`d_model=128, num_heads=4, ff_dim=512, num_layers=4, seq=5`

```
====== Benchmark Summary (split #24) ======
model = TransformerNet(input_dim=len(factor_cols),d_model=128, num_heads=4, num_layers=4, ff_dim=512)
 batch_size  use_amp  repeats  avg_time_sec  std_time_sec  min_time_sec  max_time_sec
       1024    False        3     48.363309      0.186900     48.203201     48.625493
       1024     True        3     48.545183      0.352808     48.209780     49.032790
        512    False        3     52.888417      0.648896     52.290412     53.790238
        512     True        3     56.642478      0.190116     56.400383     56.864810
        256    False        3     99.624951      2.860424     97.517334    103.668994
        256     True        3    110.293901      0.499010    109.658346    110.877328
```

Large

d_model=256, num_heads=8, ff_dim=1024, num_layers=6,seq=5

```
====== Benchmark Summary (split #24) ======
TransformerNet(input_dim=len(factor_cols),d_model=256, num_heads=8, num_layers=6, ff_dim=1024)
 batch_size  use_amp  repeats  avg_time_sec  std_time_sec  min_time_sec  max_time_sec
        512    False        3     75.769370      0.087357     75.656439     75.869214
        512     True        3     73.196284      0.226641     72.894548     73.440776
       1024     True        3     57.019442      0.141353     56.872506     57.210292
       1024    False        3     68.941256      0.080487     68.875533     69.054601
       2048     True        3     68.246054      0.755563     67.277876     69.121663
       2048    False        3     68.080577      2.408195     65.850089     71.424678
       4096     True        3     86.149766      1.154961     85.111487     87.760870
       4096    False        3     87.190958      0.560415     86.656037     87.964868
```

d_model=256, num_heads=8, ff_dim=1024, num_layers=6,seq=21

```
====== Benchmark Summary (split #24) ======
TransformerNet(input_dim=len(factor_cols),d_model=256, num_heads=8, num_layers=6, ff_dim=1024) seq =21
 batch_size  use_amp  repeats  avg_time_sec  std_time_sec  min_time_sec  max_time_sec
       1024     True        1     83.357674           0.0     83.357674     83.357674
       1024    False        1    151.712116           0.0    151.712116    151.712116
       2048    False        1    139.938540           0.0    139.938540    139.938540
       2048     True        1     78.905366           0.0     78.905366     78.905366
       4096    False        1    129.795845           0.0    129.795845    129.795845
       4096     True        1     82.101910           0.0     82.101910     82.101910
```

```
====== Benchmark Summary (split #24) ======
 batch_size  use_amp  repeats  avg_time_sec  std_time_sec  min_time_sec  max_time_sec
       1200    False        3    151.402769      0.146969    151.203169    151.552764
       1200     True        3     89.216995      0.138210     89.109503     89.412116
       1500    False        3    147.265340      1.288445    145.762713    148.909244
       1500     True        3     85.905587      1.080236     84.530604     87.169648
       1800    False        3    149.424227      0.281072    149.052962    149.732839
       1800     True        3     87.453449      0.208405     87.268903     87.744734
```

Conclusion

- AMP yields **significant speedup** in large networks.
- In small networks, AMP can **increase** computation time instead.