Toss Report : GPU Hill Climbing Ensemble for Merlin framework

JUNGHWAN NA*, Seoul National University, Laboratory of. Molecular Developmental Genetics

SUNGHOON BYUN**, Soongsil University, Laboratory of. Computer Vision

이 보고서는 "토스 NEXT ML CHALLENGE : 광고 클릭 예측(CTR) 모델 개발 대회" 1위 솔루션을 소개합니다. 학습 데이터셋은 약 10M행으로 구성되어 있으며, 'clicked' 타겟 컬럼을 포함해 총 119개의 컬럼이 존재합니다. 대회는 실제 프로덕션 환경에서의 CTR 예측 모델 활용가능성과 그에 따른 평가 지표를 포함하고 있으며, 구체적으로 Average Precision과 Weighted-Logloss의 합으로 이루어져 있습니다. 추론 환경은 별도로 분리되어 모델 가중치를 로드해 성능을 테스트하며, CTR 예측 성능과 프로덕션 적용 가능성을 평가합니다. 본 솔루션이 Public과 Private에서 모두 1위를 차지했으며, 총 36개 모델의 Hill-Climbing 앙상블로 이루어져 있습니다. 비록 프로덕션 환경에서 앙상블이 쓰이는 경우는 드물지만, 본 솔루션은 CTR 특화 프레임워크인 NVIDIA-Merlin을 활용해 대부분의 연산을 GPU에서 수행해 앙상블 모델도 충분히 프로덕션 환경에서 쓰일 수 있다는 가능성을 제시합니다. 제한된 시간과 컴퓨팅 자원으로 테스트 데이터셋을 추론해야 하는 프로덕션 환경에서, 본 솔루션은 빠른 학습과 추론으로 "대규모 데이터셋에서의 대규모 앙상블(Large-Large Ensemble)"을 가능하게 했습니다. GPU 가속 프로덕션 환경의 이점을 충분히 살려 오픈소스 라이브러리인 NVTabular, RAPIDS cuDF, PyTorch와 Tensorflow 등을 활용해 최종 솔루션의 학습 및 추론 시간을 약 50배 이상 단축했으며, 이는 비즈니스 관점에서도 기존 추천 시스템 업데이트 및 실시간 파이프라인 주기를 단축하고 Merlin 프레임워크로 대체할 충분한 이점을 시사합니다.

핵심 개념 • **추천 시스템 → CTR 예측**

Introduction

CTR(Click-Through Rate)은 유저가 주어진 아이템 또는 광고를 클릭할 확률을 의미합니다. 이는 광고주와 서비스 제공자들의 이익 창출로 이어지며, CTR을 높이는 것이 곧 서비스 제공자의 매출과 직결되는 특성을 지닙니다.

그러나 비즈니스 관점에서 수익과 사용자 경험은 종종 일치하지 않으며, 이 때문에 수익 최적화(정확한 eCPM)와 좋은 추천 순서(Rank)는 종종 목표 균형을 요구합니다. 이 때문에 토스 NEXT ML CHALLENGE 대회에서 복합 메트릭을 요구한 것으로 풀이되며, 아래 메트릭은 확률 정확도와 순서 품질을 다차원적으로 평가해 "좋은 예측"과 "좋은 순서"를 동시에 달성하려는 실무적 접근으로 여겨집니다.

Table 1. Competition Metric

$$Score = 0.5 \times AP + 0.5 \times 1/(1 + WLL)$$

여기서,
- AP (Average Precision) = $1/|P| \times \Sigma\ P(k) \cdot rel(k)$
- P(k): k순위까지의 정밀도
- rel(k): k순위 관련성 (클릭=1, 미클릭=0)
- |P|: 전체 클릭 수

- WLL (Weighted LogLoss) = $-1/2 \times$ [클릭 로그손실 + 미클릭 로그손실] = $-1/2 \times [1/n_1 \times \Sigma \log(p_i) + 1/n_0 \times \Sigma \log(1-p_i)]$
- 클릭/미클릭 50:50 가중치 적용
- n_1, n_0: 각 클래스 샘플 수
- p_i: 예측 확률

*NVIDIA Merlin 프레임워크 도입 및 NVTabular 최적화, 약 40개 모델 앙상블 baseline에 기여하였습니다.
**Wide&Deep 단일 모델 고도화 및 전처리 변형을 통한 최종 앙상블 성능 향상에 기여하였습니다.

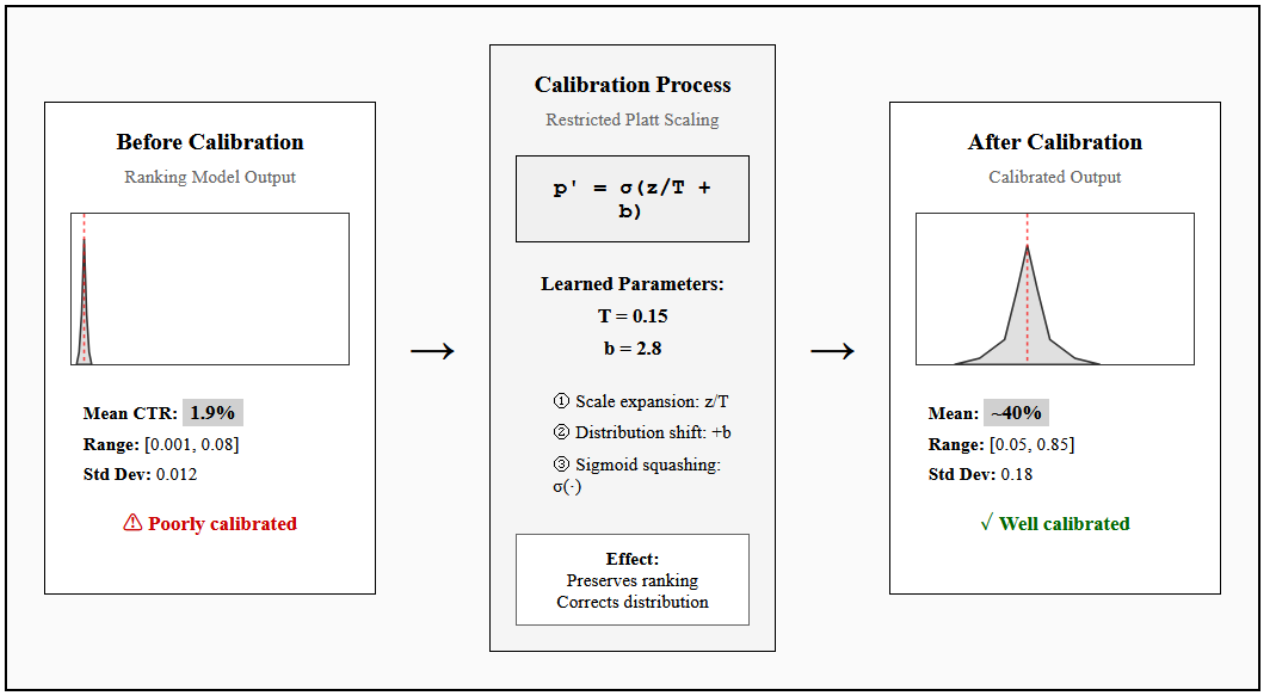
1. Complex Metric Optimization

- BCE with AUC early stopping, Temperature scaling

일반적으로 복합 메트릭에 F1 또는 AUC가 포함되어 있는 경우, 이를 직접적으로 최적화하는 건 계산적으로 다루기 어렵고 미분 불가능한 문제가 있습니다. 그러나 Binary Cross Entropy(BCE)와 같은 대리 손실을 최적화한 후, 임계값을 적용하면 통계적으로 일관된 추정치를 얻을 수 있습니다.[1] 또한, 크로스 엔트로피 손실을 최소화하면 최적 베이즈 분류기에 근사하는 결정 함수를 가진 분류기를 생성해 결과적으로 AUC를 최대화할 수 있습니다.[2] 이 근거를 토대로, 본 솔루션은 복합 메트릭을 직접 최적화하는 대신 BCE로 학습하면서 AUC로 early stopping을 두는 방식을 사용합니다. 이는 학습과정에서 rank를 최대화하며 좋은 추천 순서인 사용자 선호도를 극대화합니다.

한편으로, Temperature scaling은 로짓을 스칼라 온도 T로 재조정하는 간단한 후처리 기법입니다. 이 변환은 argmax를 변경하지 않으므로 정확도를 보존하고, 순위를 보존하므로 AUC를 유지하며, 보정을 크게 개선하는 효과가 있습니다.[3] 일반적인 Temperature scaling은 logit Z를 T로만 나누지만, 본 솔루션에서는 T와 bias 두 개의 파라미터를 사용하며 이는 최종 score 최적화를 위해 T로 예측 분산을 조정하면서도 bias로 평균을 이동시키는 방식으로 엄밀한 의미로는 Platt scaling의 변형(Restricted Platt Scaling)에 가깝습니다.

Figure 1: Restricted Platt Scaling for CTR probability Calibration



결과적으로 이 방식은 학습 때 rank를 높인 후 후처리로 최종 score를 극대화할 수 있으며, 각 단계를 optuna 등의 방식으로 개별 최적화할 수 있어 효율적입니다.

학습 및 후처리의 모든 단계는 Merlin framework상에서 이루어져 GPU 최적화 및 가속화의 이점을 받았으며, 대규모 데이터셋에서 구체적으로 어떤 방식으로 이점을 얻었는지는 다음 페이지에서 상세히 기술하겠습니다.

2. NVIDIA Merlin Framework

NVIDIA Merlin은 **추천 시스템을 위한 end-to-end GPU 가속 프레임워크**로, 데이터 처리부터 모델 학습, 추론까지 전 과정을 GPU에서 수행할 수 있도록 설계되었습니다. Merlin Framework 도입으로 대규모 앙상블 모델이 현실적인 프로덕션 적용 가능성을 가질 수 있게 되었습니다.

2.1 Merlin Framework 구성요소

Merlin은 추천 시스템 파이프라인의 각 단계를 담당하는 아래 모듈로 구성됩니다:

NVTabular (데이터 전처리)
- GPU 가속 feature engineering 라이브러리
- cuDF 기반으로 pandas와 유사한 API 제공
- Categorify, Normalize, FillMissing 등 추천 시스템 특화 연산자
- 본 솔루션의 핵심 가속 컴포넌트로 활용

Merlin Models (모델 학습)
- TensorFlow/PyTorch 기반 추천 모델 라이브러리
- Wide&Deep, DLRM, DCN 등 산업 표준 모델 제공

HugeCTR (대규모 CTR 특화)
- C++ 기반 고성능 CTR 모델 학습 프레임워크
- Multi-node, Multi-GPU 학습 최적화
- Docker 이미지 빌드 요구사항과 대회 환경 제약으로 인해 본 솔루션에서는 제외

Merlin Systems (프로덕션 배포)
- Triton Inference Server 통합
- 실시간 추천 서빙 최적화

본 솔루션은 NVTabular의 데이터 전처리 파이프라인과 Merlin Models의 딥러닝 모델을 활용했으며, 이를 통해 복잡한 환경 설정 없이도 GPU 가속의 이점을 최대한 활용할 수 있었습니다.

2.2 GPU Categorify: 혁신적인 병렬 처리

기존 CPU 기반 범주형 변수 인코딩의 한계는 명확합니다. Python의 LabelEncoder는 10.7M개 행을 순차적으로 처리하며, Python GIL(Global Interpreter Lock)로 인한 멀티스레딩 한계와 pandas DataFrame의 반복적인 메모리 복사로 인한 병목 현상이 발생합니다.

반면 NVTabular의 GPU Categorify는 근본적으로 다른 접근을 취합니다. GPU 병렬 처리 아키텍처이며, 10.7M행인 본 대회의 데이터셋을 처리할 때 10,700개 블록 × 1,000 스레드 = 10,700,000개와 같은 동시 처리 방식으로 작동합니다. 각 CUDA Core가 독립적으로 행을 처리하며, GPU Hash Table을 통해 O(1) 시간복잡도를 보입니다. 이러한 massive parallelism은 단일 스레드 처리 대비 이론적으로 수만 배의 처리량 향상을 가능하게 합니다.[4]

2.3 Zero-Copy I/O: 메모리 효율성의 극대화

데이터 I/O는 대규모 데이터셋 처리의 주요 병목 구간입니다. 일반적인 pandas 파이프라인과 Merlin의 차이는 아래와 같습니다.

Pandas 파이프라인: Disk → OS Buffer → User Space → pandas → numpy → GPU (총 5번의 메모리 복사 발생)
NVTabular 파이프라인: Parquet(Arrow Format) → Memory Map → GPU DMA (Zero-copy: 메모리 복사 없음)

Arrow 포맷의 Parquet 파일은 GPU가 직접 읽을 수 있는 columnar format이며, Memory-mapped I/O와 DMA(Direct Memory Access)를 통해 중간 직렬화 과정 없이 GPU 메모리로 직접 로드됩니다. 이는 특히 10M 규모의 데이터셋에서 드라마틱한 성능 향상을 가져옵니다.

결론적으로, 하드웨어 수준의 병렬화, 메모리 계층 최적화, Columnar Processing(행 단위가 아닌 열 단위 처리) 등의 장점이 있으며, Parquet 파일 및 NVIDIA CUDA 생태계와도 완벽하게 통합되어 시너지를 발휘하는 프레임워크로 정의됩니다.

코드에서는 올바른 적용을 위해 다음과 같은 패턴을 사용했습니다:

```
workflow = nvt.Workflow(
    categorical >> ops.Categorify(
        freq_threshold=0,     # 모든 카테고리 유지
        max_size=50000       # GPU 메모리 제약 고려
    )
)

# Cross-validation시 데이터 누출 방지
for fold in range(10):
    workflow = create_workflow()        # 각 fold마다 독립적인 workflow
    workflow.fit(train_data_only)       # validation 데이터 제외
    workflow.transform(val_data)        # fit 없이 transform만 적용
```

NVTabular는 workflow를 사전 생성해 학습 전 미리 GPU dataframe에 전처리된 데이터를 로드하며, 이 덕분에 학습 단계에서 CPU->GPU 복사나 CPU 병목으로 인한 속도 저하로부터 자유롭습니다. 전체 데이터셋에 대해 nvt.workflow 생성은 5~10초 내로 완료되며, 재실행시 workflow를 로드하는 형태로 더욱 빠른 학습이 가능합니다. 이 방법을 통해 36개 모델의 대규모 앙상블임에도 불구하고 실제 프로덕션 환경의 시간 제약 내에서 학습과 추론을 완료할 수 있었고, 최종적으로 약 50배 이상의 속도 향상을 달성했습니다.

3. Data Preprocessing

3.1 Sequence Processing

본 대회의 데이터셋은 약 10M행, 119개 열로 이뤄진 정형 데이터입니다. 각 feature는 익명화되어 있으며, 가변 길이 시퀀스 열이 존재합니다. 가변 길이 시퀀스 열은 문자열로서, [512, 24, 3, 46...] 과 같은 형식으로 정수와 쉼표로 구성된 연속적인 유저 행동 데이터입니다. 토스 앱 내부의 item이나 특정 영역을 누를 때 로깅되는 값으로 추정되며, sequential한 특성으로 특정 패턴 또는 등장 횟수 등이 CTR에 영향을 줄 것으로 예상됐습니다.

시퀀스 열의 전처리는 크게 세 가지 문제가 있었습니다.

1. max_len=6590으로 너무 긴 정수 배열
2. 가변 길이로 인한 embedding 난이도 증가
3. 적은 vocabulary 및 시퀀스 열과 CTR과의 직접적인 연관성 부재

우선 시퀀스 열의 길이가 너무 길었기에 cudf GPU dataframe에 올리기 어려운 문제가 있었습니다. cudf는 문자열 길이 제한이 있기에 truncate하거나 다른 dtype으로 변환해야 했으며, 이에 **정수열 리스트로 변환해 NVTabular에 통합**하였습니다. 딥러닝 모델에 시퀀스 열을 넣을 때는 embedding 형태로 넣는 게 일반적인데, 이때 가변 길이를 padding하거나 길이에 맞춰 일일이 자를 경우 메모리 폭증 또는 학습 속도 지연이 불가피합니다. 이를 해결하기 위해 마스킹 벡터화 같은 방법을 도입했으나, 그보다

중요한 건 **전체 시퀀스를 넣는 게 아니라 truncate하는 방식**입니다. 전체 시퀀스를 넣었을 때와 truncate했을 때 유의미한 성능 차이는 없었으며, 이는 전체 시퀀스의 패턴보다는 지역적인 패턴, 또는 시퀀스 열 자체의 표현력이 CTR에 충분한 정보를 주지 못한다는 의미로 해석됩니다.

또한 시퀀스 열의 vocabulary를 분석했을 때 유니크한 정수 개수가 약 590개로 평균 시퀀스 길이(약 530)에 비해 적었습니다. 이는 반복 아이템이 많이 등장한다는 의미이며, 대규모 vocabulary나 텍스트가 등장하지 않아 정보가 제한적임을 시사합니다.

구체적으로, 아래 시퀀스 데이터 EDA 결과에 따라 전체 시퀀스 열을 학습하는 건 효용이 크지 않다는 결론을 얻었습니다.

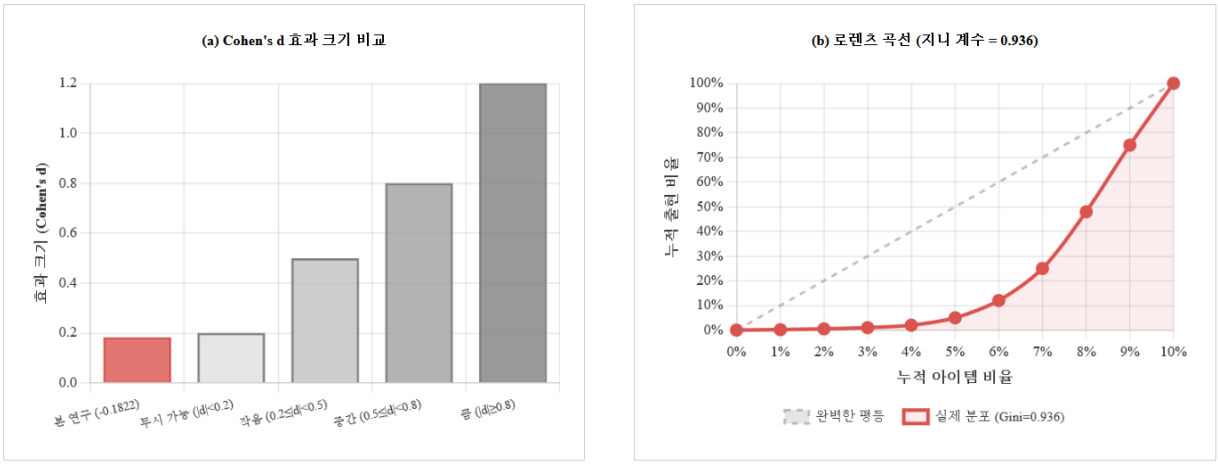


Figure 2. (a) Cohen's d = -0.1822는 '무시할 수 있는(negligible)' 수준의 효과 크기를 보여주며, 이는 '작은(small)' 효과의 기준인 0.2보다 훨씬 낮은 수치입니다. 통계적으로는 유의미하지만($p < 0.001$), 시퀀스 길이가 CTR에 미치는 실질적 영향은 극히 미미합니다. (b) 로렌츠 곡선은 지니 계수 0.936의 극단적인 아이템 인기도 집중을 보여줍니다. 상위 10%의 아이템이 전체 상호작용의 80% 이상을 차지하여, 긴 시퀀스가 제공하는 정보의 가치가 심각하게 제한됨을 알 수 있습니다.

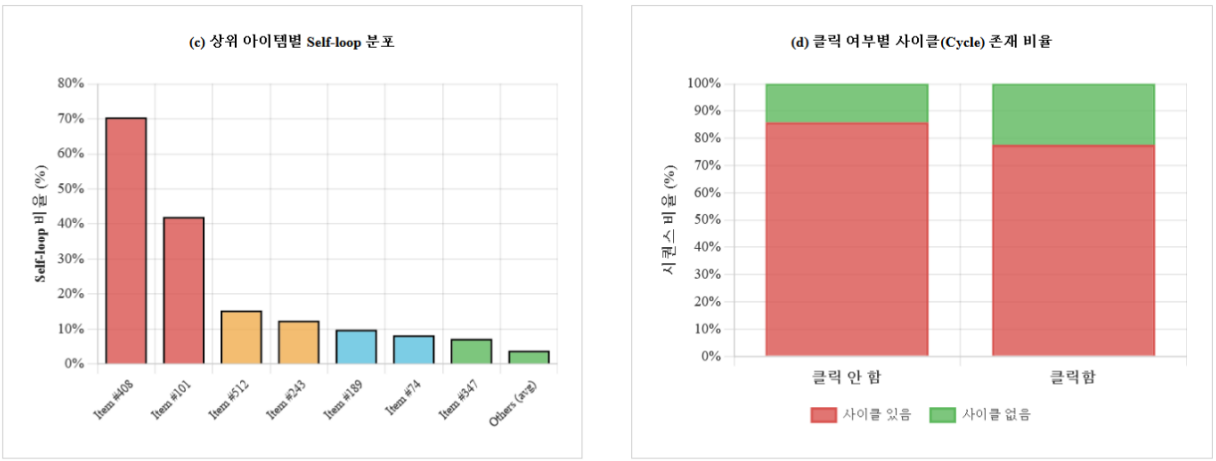


Figure 3. (c) Self-loop 분석 결과, 특정 아이템들이 70% 이상의 자기 전이(self-transition)를 보이는 극단적 중복성이 발견되었으며, 전체 self-loop 비율 10.8%는 데이터셋 전반에 걸쳐 상당한 반복적 행동이 존재함을 나타냅니다. (d) 사이클 탐지 결과, 클릭하지 않은 시퀀스의 85.85%와 클릭한 시퀀스의 77.62%에서 반복 패턴이 발견되었습니다. 이는 전체 시퀀스를 처리하는 것이 다양한 사용자 관심사보다는 대부분 중복된 정보를 포착하고 있음을 보여줍니다.

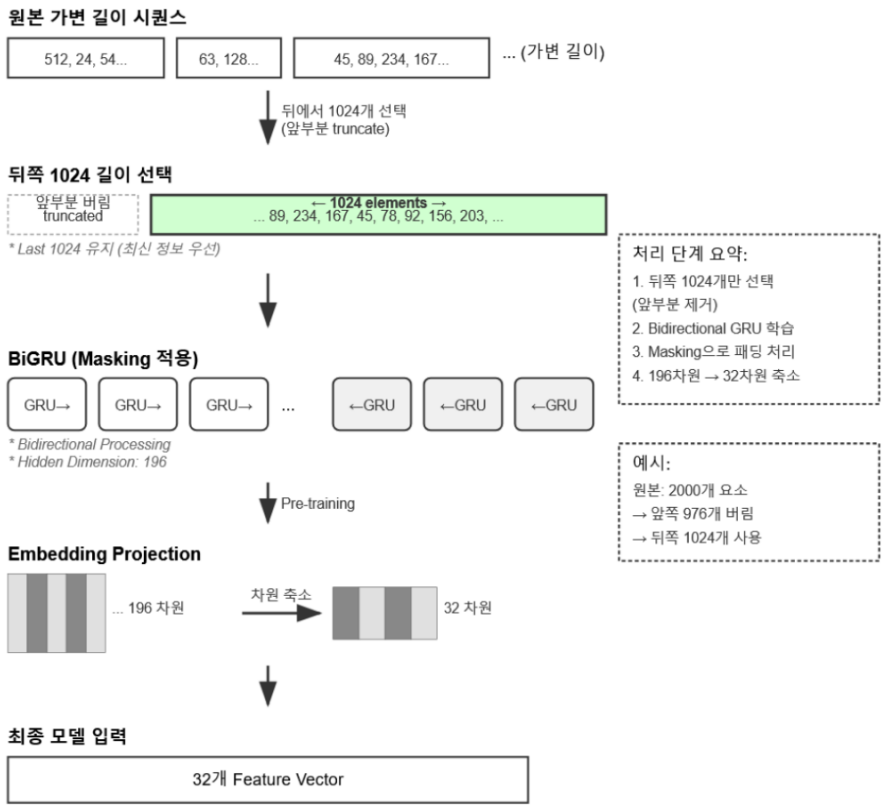
-**Cohen's d**는 두 그룹 간 평균 차이를 표준편차를 나눈 값으로, 여기서는 클릭과 비클릭 그룹간의 길이 차이를 의미합니다. Cohen's d의 절댓값이 0.2 이하일 경우 통계적으로 무시할 수 있다는 의미이며, 시퀀스 길이와 CTR과의 관계가 실질적으로 무시할만한 수준임을 뜻합니다.

-**로렌츠 곡선**은 불평등 정도를 시각화하는 곡선으로, 여기서는 누적 인구 대비 누적 빈도 비율입니다. 곡선이 아래로 처질수록 불평등이 심하며, 본 보고서에서는 극소수 아이템이 대부분의 상호작용을 독점하는 것을 뜻합니다. **지니 계수**는 로렌츠 곡선과 평등선 사이의 면적 비율로 1에 가까울수록 극단적 불평등을 의미합니다. 소수의 인기 아이템만 반복 노출되며, 대부분의 인기 아이템은 거의 나타나지 않음을 의미합니다. 즉, 시퀀스가 길어도 동일한 인기 아이템만 반복되어 정보 가치가 낮다는 점을 시사합니다.

-**Self loop**은 동일한 아이템을 연속으로 클릭하는 패턴을 뜻합니다. 페이지 새로고침, 콘텐츠 집중 소비, 탐색, 시스템 오류 등을 뜻하며, Self loop은 새로운 정보를 제공하지 않으므로 정보 중복 문제가 있음을 시사합니다.

결국, **긴 시퀀스는 많은 정보가 아니라 같은 정보의 반복**이며, 이러한 근거로 약 1000의 길이로 truncate하는 전략을 사용했습니다. 앞에서부터 1000을 truncate하는 것과 뒤에서부터 1000을 truncate하는 방식 모두 메모리 효율성과 성능의 균형을 잡을 수 있었으며, 두 방식은 큰 성능의 차이가 없었습니다.

Figure 4: Simple bi-GRU seq embedding



Truncate 방식은 너무 긴 시퀀스의 중복 정보를 쳐내면서도 적절한 패턴을 학습하는 효과가 있습니다. 패턴을 학습하는 방법은 저수준부터 단순 시퀀스 통계 추출 < GRU < LSTM < Attention 순으로 복잡도가 커지며, 각각의 방식은 표현력과 학습 속도의 trade-off 관계입니다. 단순 시퀀스 통계 추출은 후술할 Feature Engineering의 노이즈 문제로 인해 유의미한 feature인지 판단하기 어려운 문제가 있었으며, attention 또는 attention pooling 방식은 과도한 복잡도로 성능이 오히려 하락하는 결과를 얻었습니다. 결과적으로, Bi-GRU sequence embedding 방식은 본 데이터셋에 딱맞는, 실험적으로 도출한 적정 아키텍처입니다.

GRU(Gated Recurrent Unit)의 설계 철학은 LSTM의 핵심 아이디어는 유지하면서도, 이를 최대한 단순화한 아키텍처로 볼 수 있습니다. GRU는 리셋 게이트와 업데이트 게이트만을 활용해 정보의 흐름을 제어하고, 별도의 메모리 셀 없이 하나의 hidden state로 모든 정보를 관리하며, 이는 불필요한 복잡성을 제거하면서도 성능을 유지하는데 핵심적인 역할을 합니다.[5]

일반적으로 추천 시스템의 시퀀스 열은 복잡한 패턴이 있어 많은 논문에서 embedding 후 직접 딥러닝 모델에 넣으라는 식의 권장사항이 있습니다. 이럴 경우 embedding 자체가 gradient에 들어가기에 표현력은 좋아지지만, 문제는 **긴 시퀀스 열이 학습시 매 배치마다 backward pass로 들어가는데, 이때의 gradient 계산이 학습 속도를 매우 느리게 만듭니다.** 왜냐하면 GRU 학습시 forward pass 동안 저장되는 모든 중간 활성화 값들이 backward pass에서 필요하므로, 시퀀스 길이가 길수록 메모리 요구량이 선형적으로 증가하기 때문입니다. 더군다나 GRU는 LSTM보다 간단한 구조임에도 sequential한 특성 탓에 병렬화가 어렵습니다. 이 문제를 해결하기 위해 **embedding projection 방식을 채택**해 차원을 축소하고, feature 형태로 직접 주입하는 방식을 택했습니다. 이는 표현력을 어느 정도 유지하면서도 **빠른 학습의 장점을 유지**합니다. Embedding projection은 정보의 상당 부분을 버리는 문제가 있지만, 복잡도를 줄이고 메모리 효율성을 높여 학습 속도를 가속화하고, 성능을 오히려 높이는 장점이 있었습니다.

결과적으로 약 1000개의 시퀀스로 truncate한 후, 학습 데이터셋 전체에 대한 Bi-GRU 사전학습을 수행했습니다. 이때 가변길이에 대응하기 위해 masking 방식으로 학습 효율성을 높였으며, 96차원의 forward와 backward GRU, 총 192차원의 Bi-GRU으로 학습된 embedding을 32차원의 feature로 projection했습니다. 고차원 표현에는 상당한 중복성이 있기에, **projection 방식은 오히려 노이즈를 필터링하고 의도적인 정보 병목을 만들어 성능을 높이는데 기여**했습니다.

3.2 Stratified 10-fold CV

데이터셋을 시계열로 해석해 **sunday validation(temporal split)**을 했을 때 CV-LB차이가 가장 적었습니다.
이는 monday-saturday로 train, sunday로 validation하는 방식입니다.
CV : 0.352100, LB : 0.3509591693
CV : 0.351963, LB : 0.3508297851

이는 test에 sunday가 많이 포함되어 있거나 sunday 패턴을 따른다는 의미로 해석됩니다.
그러나 추가 실험으로 "**LB는 sunday가 train에 많이 포함될수록 높아진다**"는 점을 발견했습니다.

예를 들어, 평일과 주말의 중간 패턴인 friday를 validation으로 삼았을 때 **단일 모델 LB가 가장 좋았습니다.**
이때 CV-LB 차이가 있지만, 상관관계는 안정적입니다.

[Friday validation CV-LB]
CV : 0.357602, LB : 0.3511151209
CV : 0.357661, LB : 0.35125415
CV : 0.357699, LB : 0.3512969618
CV : 0.357947, LB : 0.3517318686 (단일 모델 best)

이는 Sunday가 학습에 온전히 쓰였기 때문에 점수가 높았다는 뜻으로 해석됐으며, 이를 바탕으로 **"test의 sunday 패턴을 맞추는 게 관건이며, sunday 패턴을 맞추려면 train에서 sunday를 학습해야 한다"** 는 결론을 얻었습니다. Sunday의 domain을 맞춰야 하는 **Domain adaptation task**인 셈입니다.

이를 토대로 **[day_of_week, clicked] 값이 각 fold에 모두 균등하게 들어가는 stratified-10fold CV로 확정했습니다.**
이는 sunday를 train에 최대한 균등하게 넣으면서도, OOF 앙상블 및 안정적인 LB 예측을 가능하게 합니다.

이때부터 CV-LB 갭은 있지만 차이가 안정적이므로, CV를 본격적으로 신뢰할 수 있었습니다. 다만 sunday 데이터가 fold로 인해 줄어든만큼, 단일 모델의 성능은 소폭 하락합니다. (약 -0.0001)

3.3 Feature Engineering

머신러닝에서 Feature Engineering을 할 때는 일반적으로 min_improvement를 둡니다. 통계적으로 **"3-sigma rule"**이라 하여 표준편차의 3배를 넘는 improvement를 채택하는 게 안정적입니다.[6] 그러나 **CTR 과제에서는 sparse-interaction과 signal/data ratio 문제**가 있고, 설상가상으로 대규모 데이터일수록 개별 feature의 영향력은 통계적으로 약해집니다. 그렇다고 **표준편차를 계산하자니 계산량이 기하급수적으로 늘어나는 문제**가 있습니다.

예를 들어, 5fold-CV가 0.353183 ± 0.000688라면,
안정적으로 채택할 수 있는 min_improvement는 약 + 0.002입니다. (약 99%의 신뢰도)
이런 improvement를 보이는 feature는 찾을 수 없었고, 그렇다고 표준편차를 낮추기 위해 multiple-seed, 10fold 등으로 계산량을 기하급수로 늘릴 수도 없었습니다. 그렇다면 이런 상황에서는 **검증하기 어렵고 효과가 미미한 Feature Engineering**보다는,
확실하게 검증할 수 있는 **모델 구조 및 파라미터 튜닝**에 집중해야 한다는 결론을 얻었습니다. 또한 튜닝 이후에는 개별 모델의 예측을 보정해 줄 수 있는 **multi-model ensemble**이 핵심이라고 봤습니다.

결과적으로 feature engineering은 이번 대회에서 성능을 올려주는 확실한 수단이라기보다는, 그 자체로는 모델 성능에 큰 영향을 주지 못하며 앙상블 모델 다변화 측면에서 유용한 방법 정도로 생각됩니다.

4. Our Solution

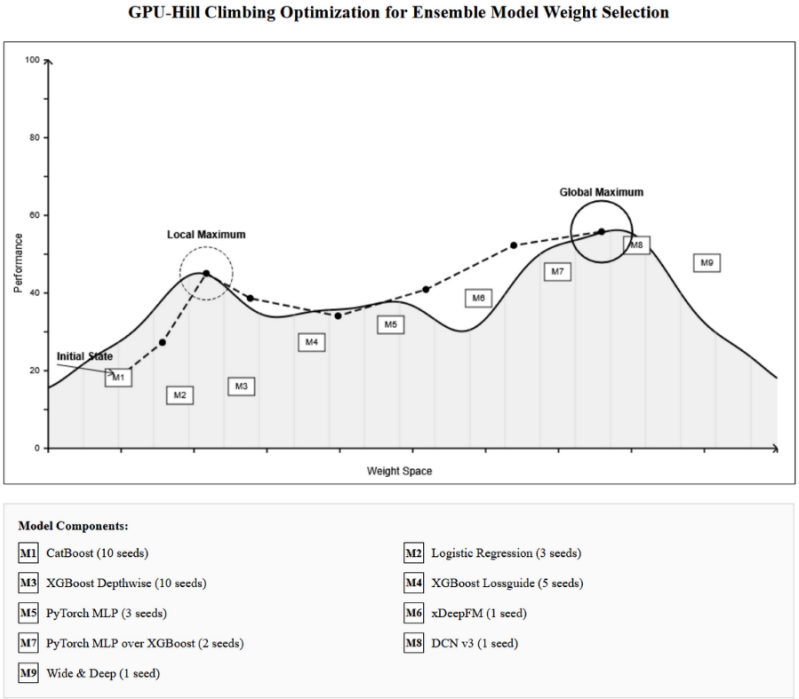


Figure 5: GPU-Hill Climbing Ensemble Architecture

4.1 GPU-Hill Climbing Ensemble

Hill climbing ensemble은 **여러 OOF 모델이 있을 때 해당 모델들의 최적 가중 평균을 찾는 기법**입니다.[7] 이 방식은 각 모델의 seed 개수에 상관없이 안정적으로 최적 조합을 찾고, 과적합에서 자유롭다는 장점이 있으며, **각 모델의 영향력을 수치상으로 확인**할 수 있다는 점에서도 큰 장점이 있습니다. 본 대회에서는 각 모델에 개별 Temperature scaling을 적용했기에, 파이프라인 특성상 calibration하기 전의 예측값으로 OOF를 내야하는 stacking 등의 기법은 적용하기 어려웠습니다.

각 OOF 모델이 1000만행 x 2열에 달했으므로(prediction, target), cudf를 활용해 GPU 상에서 competition metric과 hill climbing을 빠르게 계산하는 방식으로 가속화하였습니다. scikit-learn으로 AP를 계산하면 이 연산이 매우 느리므로, 반드시 cupy 등 GPU 상에서 계산하는 방식이 권장됩니다. 이때 ranking 계산을 위해 필연적으로 쓰이는 argsort를 제외하면 병목이 없어 아주 빠르게 계산됩니다. Hill climbing은 본질적으로 완전 탐색이 아니라 local optima에 빠질 수 있으므로, random restart/shotgun 방식으로 여러 번 초기화 탐색을 진행합니다. XGBoost로 기본 파이프라인을 구축한 후, **Feature Engineering이 없는 동일 파이프라인에서 모델 구조만 바꾸며 최대한 다양한 모델을 탐색**할 수 있었습니다.

이때 한 모델의 multiple-seed보다는 **다른 모델 구조일수록 성능 향상**이 있었습니다. 최종적으로 36개의 모델 앙상블 중 25개 모델의 실제 가중치가 사용되었습니다. Hill-climbing에서 동일 모델의 seed는 상관성이 높아 가중치가 분산되므로, 단일 시드 모델인 wide_deep의 영향력이 가장 높게 측정됩니다.

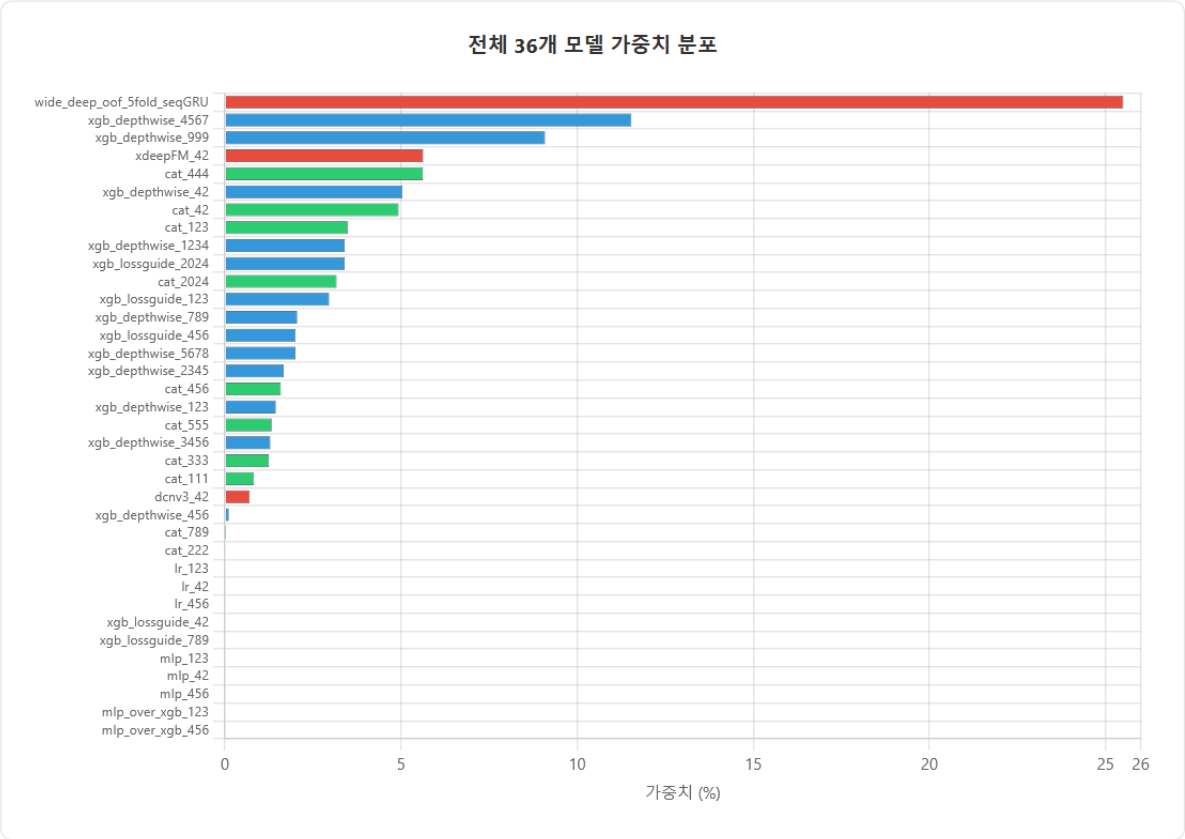


Figure 6: GPU-Hill Climbing Ensemble Weight distribution

4.2 Tree-based Models

XGBoost Depthwise

Negative sampling 등의 기법 없이 scale_pos_weight 값만으로 클래스 불균형을 해결했습니다.

Stage1에서 optuna로 scale_pos_weight값만을 탐색해 최적값을 확정한 후,

Stage2에서 본격적인 모델 파라미터 튜닝을 위한 optuna를 수행했습니다.

Stage1, Stage2 모두 LR=0.1, num_boost_round=150 고정의 빠른 설정으로 탐색했으며, RTX A6000 GPU 환경에서 1trial 약 5초 가량으로 빠른 탐색이 가능했습니다. 탐색된 파라미터는 아래와 같습니다.

```
scale_pos_weight=2.7314
Best hyperparameters:
max_depth: 15
max_leaves: 503
subsample: 0.8922288170923977
colsample_bytree: 0.6965797963245798
min_child_weight: 83
gamma: 1.0035755449048576
reg_alpha: 0.5576358212867057
reg_lambda: 0.8416579109274109
```

이후 위 파라미터로 본학습은 LR 약 0.007, num_boost_round를 길게 주고 early stopping=300으로 LR을 14배 가량 낮춰 안정적인 수렴을 유도했습니다. 트리 모델에서의 LR은 각 단계에 이미 최적화된 트리를 추가하는 방식이므로, 초기 LR이 높으면 오히려 새 트리가 업데이트하지 못하는 문제가 있습니다. 따로 LR Schedular 없이 낮은 LR로 천천히 쌓는 방식을 택했으며, LR은 학습 곡선을 확인하면서 낮추는 과정을 반복해 최적 지점을 찾았습니다.

stage2의 score보다 0.0001 이상의 CV 점수 향상이 있었습니다. 소요 시간이 optuna 탐색에 비해 14배 걸리나, 충분히 감당할 수 있는 범위였습니다. 단일 모델 중 가장 높은 score를 기록했으며, 이는 본질적으로 rank에 강한 트리 모델의 강력한 성능을 보여줍니다. 또한, 깊은 depth와 높은 min_child_weight 등을 감안할 때 high-order interaction과 sparse 데이터의 적절한 과적합 제어가 중요하다는 점을 알 수 있습니다. Depthwise 파이프라인 구축 후, 모델 구조를 바꿔가며 아래 파이프라인들을 빠르게 설계할 수 있었습니다.

XGBoost Lossguide

Depthwise가 같은 깊이의 노드를 동시에 분할하는 방식이라면, Lossguide는 LightGBM처럼 가장 손실 감소가 큰 리프 노드를 우선적으로 분할하는 방식입니다. XGBoost grow policy를 “lossguide”로 변경해 반영하며, 빠른 수렴과 높은 정확도를 제공하지만 불균형한 트리를 만들 수 있고, 과적합 위험이 상대적으로 높습니다. 단, loss 기반으로 불균형에 대응하므로 따로 scale_pos_weight 값을 적용하진 않습니다. 본학습의 LR 관련 설정은 depthwise와 동일합니다.

```
scale_pos_weight=1
Best hyperparameters:
max_depth: 0,
max_leaves: 420,
Subsample: 0.9595408158807616,
colsample_bytree: 0.6816936796084421,
min_child_weight: 24,
gamma: 0.49314855072003777,
reg_alpha: 0.43966411003417155,
reg_lambda: 0.3826998943189582
```

Catboost

앙상블 다양성 측면에서 LR을 낮추면서까지 학습하진 않았습니다. scale_pos_weight 값 탐색 - 모델 구조 탐색 이후 10fold 학습으로 타협하였습니다. 단, early stopping을 200으로 늘려 약간의 과적합을 허용합니다.

```
scale_pos_weight: 2.8609
iterations: 5000,
learning_rate: 0.1,
depth: 11,
l2_leaf_reg: 8.135893899106593,
border_count: 104,
bagging_temperature: 0.12132327165989898,
random_strength: 0.011268516320957822,
od_wait: 200,  #Early stopping patience
```

4.3 Deep Learning Models

Pytorch MLP

개별 모델의 성능이 낮아 앙상블 가중치에는 기여하지 않았으나, 딥러닝 모델 중 가장 단순한 아키텍처로 이를 baseline으로 삼아 더 복잡한 모델 파이프라인 구축이 가능했습니다. 딥러닝 모델은 트리 모델과 달리 모델별 적정 LR이 존재하므로 학습 곡선을 모니터링하면서 max_LR을 조정했습니다. optuna 탐색 단계에서는 scale_pos_weight 및 weight decay를 포함한 모델 구조를 탐색했고, 본학습에서는 100 epoch, 5epoch warm-up, max_LR: 0.002으로 5epoch 동안 개선이 없으면 50%의 reduce-on-plateau LR schedular 방식으로 학습 곡선을 모니터링하면서 max_LR만 조정해 적정 수렴값을 찾을 수 있도록 설계했습니다.

```
pos_weight: 2.1858
dropout: 0.5991099733648887,
use_batch_norm: False,
weight_decay: 6.342062801770587e-06,
activation: 'relu',
hidden_dims: [2048, 1024, 512]
```

xdeepFM

xdeepFM은 고차 feature interaction을 학습하는 CIN layer와 암묵적 고차 상호작용을 학습하는 DNN layer로 구성됩니다.

```
pos_weight: 2.1837
CIN_layer_size: [128, 128]  #CIN layer output sizes
DNN_hidden_dims: [512, 384, 256]
xdeepFM_dropout: 0.285
xdeepFM_use_batch_norm: True
xdeepFM_weight_decay: 5.9e-07
xdeepFM_activation: 'silu'
use_linear_part: False  #First-order linear part disabled
split_half: True  #CIN: split half for next layer
```

deepFM이 Factorization Machine으로 2차 상호작용만을 학습하고 나머지를 DNN에만 의존하는 형태라면, xdeepFM은 CIN layer로 FM 계열의 장점을 흡수하면서도 CIN으로 고차 상호작용을 vector-wise로 학습하고, 추가로 DNN으로 복잡한 비선형 패턴을 학습해 두 출력을 concat하는 식으로 종합 예측합니다.

CIN_layer_sizes = [128, 128]는 2-layer CIN을 의미하며, 3차 상호작용까지 명시적으로 학습합니다. Pytorch_mlp 모델이 2048의 거대한 hidden dimension을 요구하는 반면, xdeepFM은 CIN layer가 explicit interaction을 학습해 DNN의 부담이 감소하는 형태로 볼 수 있습니다. 결국, 아키텍처가 복잡해질수록 오히려 DNN의 차원을 줄이며, CIN이 자체적으로 정규화 효과를 제공해 훨씬 효율적인 아키텍처로 최적화됩니다.

DCN_v3

DCNv3는 2024년 최신 CTR SOTA 모델로 모든 피처를 동일한 embedding_dim값(=128)으로 embedding layer에 올린 후, LCN-ECN-DNN 3개 경로로 병렬 전달하는 아키텍처를 지닙니다. xdeepFM이 각 카테고리마다 다른 차원을 할당한다면, DCNv3는 동일 차원으로 맞춰 이후 cross network가 균등하게 처리하는 방식입니다. LCN(Linear Cross Network)이 저차 상호작용을 학습하며, ECN(Exponential Cross Network)은 이를 지수적으로 증폭시킨 형태로서 고차 상호작용을 학습합니다. 암묵적 상호작용은 DNN이 처리하며, LCN/ECN이 직접 label을 예측하는 독립적 학습과 auxiliary loss를 결합한 Tri-BCE loss로 총 3개의 head를 사용하는 점이 특징이라 할 수 있습니다. 다만 복잡한 아키텍처의 특성상 최대한 아키텍처를 단순화하고, 모델 앙상블을 고려했기에 32차원으로 projection한 시퀀스 feature를 제외한 약 120개의 feature로 학습했습니다. 이론적으로 완벽한 단일 우수 아키텍처보단, 모델 다양성을 높이기 위한 선택이었습니다.

```
pos_weight: 2.4766
Embedding_dim: 128 #모든 피처를 128차원 통일
LCN_layers: 3
ECN_layers: 2
DNN_hidden_dims: [1024, 512, 256]
dropout: 0.081
use_batch_norm: True
weight_decay: 1e-5
Tri_BCE_W_LCN: 0.5 #LCN auxiliary loss
Tri_BCE_W_ECN: 0.5 #ECN auxiliary loss
Tri_BCE_adaptive: False
```

Wide_Deep

비록 트리 모델의 성능이 강력했으나, 강력한 단일 딥러닝 모델 역시 필요했고 이를 위해서는 시퀀스 임베딩을 projection하는 게 아닌, 직접적으로 gradient에 넣어 학습하는 새로운 관점의 모델이 필요했습니다. wide_deep 모델은 BiGRU + attention의 192 차원으로 시퀀스를 BiGRU로 인코딩해 forward pass에서 mask-attention을 통과하며, 마지막 hidden state만 사용하는 게 아닌 모든 hidden state의 가중 평균을 학습하는 방식을 사용합니다. 빠른 학습을 위해 512 길이로 시퀀스를 truncate했습니다.

```
pos_weight: n_neg / n_pos #자동 계산
embedding_dim_cat: 16 #범주형 열 차원
cross_layers: 2
hidden_units: [512, 256, 128]
dropout: [0.1, 0.2, 0.3] #Layer-wise dropout
use_batch_norm: True
weight_decay: 1e-5
```

wide_deep의 cross network은 DCNv3와 마찬가지로 저차 상호작용을 학습하지만, 훨씬 복잡도가 낮은 특징이 있습니다. 이외에도 강력한 시퀀스 임베딩 사용으로 빠르게 수렴하는 특징이 있어 5epoch의 짧은 학습만 수행했으며, SWA로 일반화 성능을 보완했습니다. 단, Bi-GRU가 backward pass로 들어가기에 학습 속도는 20배 이상 느린 단점이 있습니다.

데이터셋 관점에서 앙상블 다양성을 높이기 위해 다른 split으로 학습한 모델도 필요했고, 이에 wide_deep은 sunday 단일 split으로 학습해 OOF를 만들었습니다. 결과적으로 차별화된 모델 아키텍처와 split 등으로 hill-climbing ensemble에서 단일 seed가 약 25%의 가중치에 기여했습니다.

5. Accerlerating Inference With a Single GPU

추론 단계에서는 NVIDIA Merlin 생태계의 GPU 가속 기능을 활용해 대규모 데이터셋에 대한 빠른 예측을 수행했습니다. 특히 2단계 캐싱 전략과 NVTabular의 GPU 전처리를 통해 CPU 대비 약 9배 이상의 성능 향상을 달성했습니다.

5.1 Two-Phase Caching Architecture

추론 파이프라인은 전처리와 예측을 분리한 2단계 캐싱 전략으로 설계했습니다:

Phase 1: 전처리 캐싱 (1회만 실행)
- 10개 fold별 독립적인 NVTabular workflow를 로드
- Validation 및 Test 데이터를 GPU에서 전처리 후 RAM에 캐싱
- 소요 시간: 약 10분 (fold당 약 60초)

Phase 2: 모델 추론 (캐시 재사용)
- 캐싱된 전처리 데이터를 10개 seed 모델에서 재사용
- XGBoost GPU predictor로 100개 모델 (10 folds × 10 seeds) 병렬 예측
- 소요 시간: 약 3분 (모델당 약 2초)

이 접근법의 핵심은 계산 비용이 높은 전처리는 1회만 수행하고, 여러 seed 모델에서 재사용함으로써 10배의 가속을 달성한 것입니다. 별도의 환경에서 학습된 wide_deep 모델을 제외하고 RTX A6000 환경에서 캐싱 없이는 각 seed마다 전처리를 반복해 총 103분이 소요되었으나, 캐싱 적용 후 약 13분으로 단축되었습니다. 성능 극대화를 위해 LR을 낮춘 XGBoost-depthwise 모델의 10fold 10seed 학습 시간이 주된 병목입니다. 그러나 10M 규모의 데이터셋을 10fold로 여러 모델을 앙상블할 수 있다는 것 자체가 혁신적인 시도로 생각됩니다.

5.2 NVTabular GPU-Accelerated Preprocessing

추론 시 NVTabular은 이미 학습된 workflow를 활용해 빠른 전처리를 수행합니다.

Workflow Loading: 각 fold별로 사전 학습된 NVTabular workflow를 디스크에서 로드합니다.
이 workflow에는 Categorify 딕셔너리, FillMissing 통계량 등이 이미 계산되어 있어 즉시 사용 가능합니다.

GPU Transform Pipeline:
1. **Zero-copy I/O**: Parquet 파일을 MerlinDataset으로 GPU 메모리에 직접 로드
2. **GPU Operations**: Categorical encoding과 missing value imputation을 GPU에서 병렬 처리
3. **Efficient Transfer**: 최종 결과만 CPU로 전송하여 데이터 이동 최소화

단일 GPU(RTX A6000)를 활용했음에도 대규모 CTR 예측 대회에서 빠른 실험 반복이 가능했으며, 특히 마감 직전 여러 앙상블 조

합을 빠르게 테스트하는데 결정적인 역할을 했습니다. NVTabular의 GPU 가속 전처리와 2단계 캐싱 전략의 조합은 메모리 효율성과 계산 속도를 동시에 달성하는 실용적인 솔루션임을 입증했습니다.

6. Conclusion

본 솔루션은 토스 NEXT ML CHALLENGE에서 **최종 CV: 0.359380(AP: 0.089676 WLL: 0.589612) Public LB: 0.35297, Private LB: 0.35179**을 달성하며 전체 1위를 기록했습니다. 특히 **Public과 Private 리더보드 모두에서 안정적인 1위**를 유지했다는 점은 과적합 없는 robust한 모델링의 결과입니다.

핵심 기여점은 다음과 같습니다:

1. **프로덕션 가능한 대규모 앙상블**: NVIDIA Merlin 프레임워크를 활용한 GPU 가속으로 36개 모델의 대규모 앙상블이 실제 프로덕션 환경에서도 운용 가능함을 입증했습니다. 학습 및 추론 시간을 50배 이상 단축하여 실시간 서빙에 적합합니다.

2. **효과적인 시퀀스 처리**: 6,590 길이의 초장문 시퀀스를 Bi-GRU와 embedding projection으로 효율적으로 처리하여 메모리와 성능의 균형을 달성했습니다. 정보 이론적 분석(Cohen's d, Gini coefficient)을 통해 truncation 전략의 타당성을 검증했습니다.

3. **복합 메트릭 최적화**: BCE 학습 + AUC early stopping + Temperature scaling의 3단계 파이프라인으로 ranking과 probability calibration을 동시에 최적화했습니다. 이는 실제 비즈니스 요구사항인 "좋은 예측"과 "좋은 순서"를 모두 충족하는 실용적 접근입니다.

본 솔루션은 단순히 대회 우승을 넘어, GPU 가속 추천 시스템이 대규모 프로덕션 환경에서 실용적으로 적용될 수 있음을 보여줍니다. 특히 Merlin 프레임워크의 도입은 기존 CPU 기반 파이프라인 대비 극적인 성능 향상을 가져와 실시간 CTR 예측 서비스의 업데이트 주기를 획기적으로 단축할 수 있는 가능성을 제시합니다.

References

[1]Harikrishna Narasimhan, Rohit Vaish, and Shivani Agarwal. 2014. On the Statistical Consistency of Plug-in Classifiers for Non-decomposable Performance Measures. In Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'14). MIT Press, Cambridge, MA, USA, 1493-1501

[2]Lian Yan, Robert H. Dodier, Michael Mozer, and Richard H. Wolniewicz. 2003. Optimizing Classifier Performance via an Approximation to the Wilcoxon-Mann-Whitney Statistic. In Proceedings of the 20th International Conference on Machine Learning (ICML'03). AAAI Press, 848-855

[3]Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. On Calibration of Modern Neural Networks. In Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17). JMLR.org, 1321-1330

[4] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008. Scalable Parallel Programming with CUDA. Queue 6, 2 (March 2008), 40-53. ACM.

[5] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. arXiv preprint arXiv:1412.3555.

[6] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. Journal of the Royal Statistical Society: Series B (Methodological), 57, 1 (1995), 289-300.

[7] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. 2004. Ensemble Selection from Libraries of Models. In Proceedings of the 21st International Conference on Machine Learning (ICML '04). ACM, 18.

A. Neural Network Model Architecture

A-1 xdeepFM

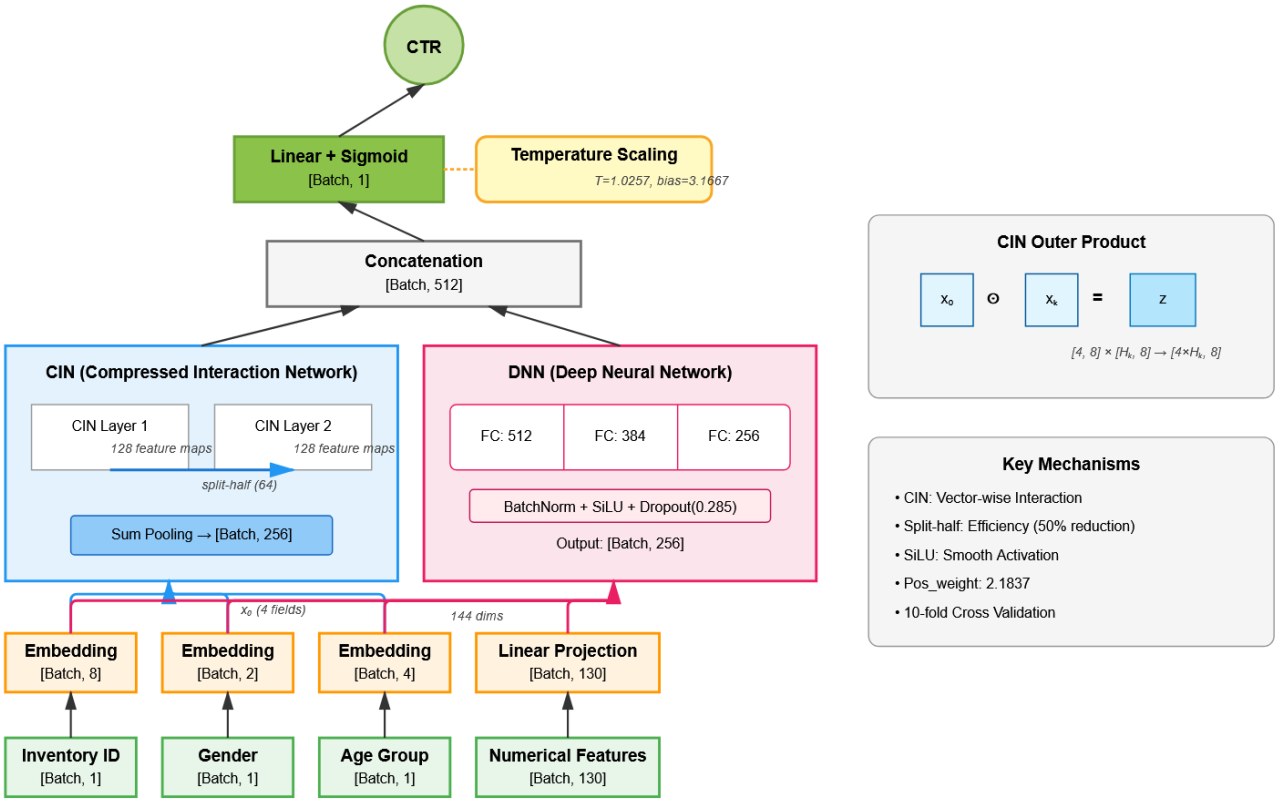


Figure 7. Visualization of neural network 1 - xdeepFM architecture.

A-2 DCNv3

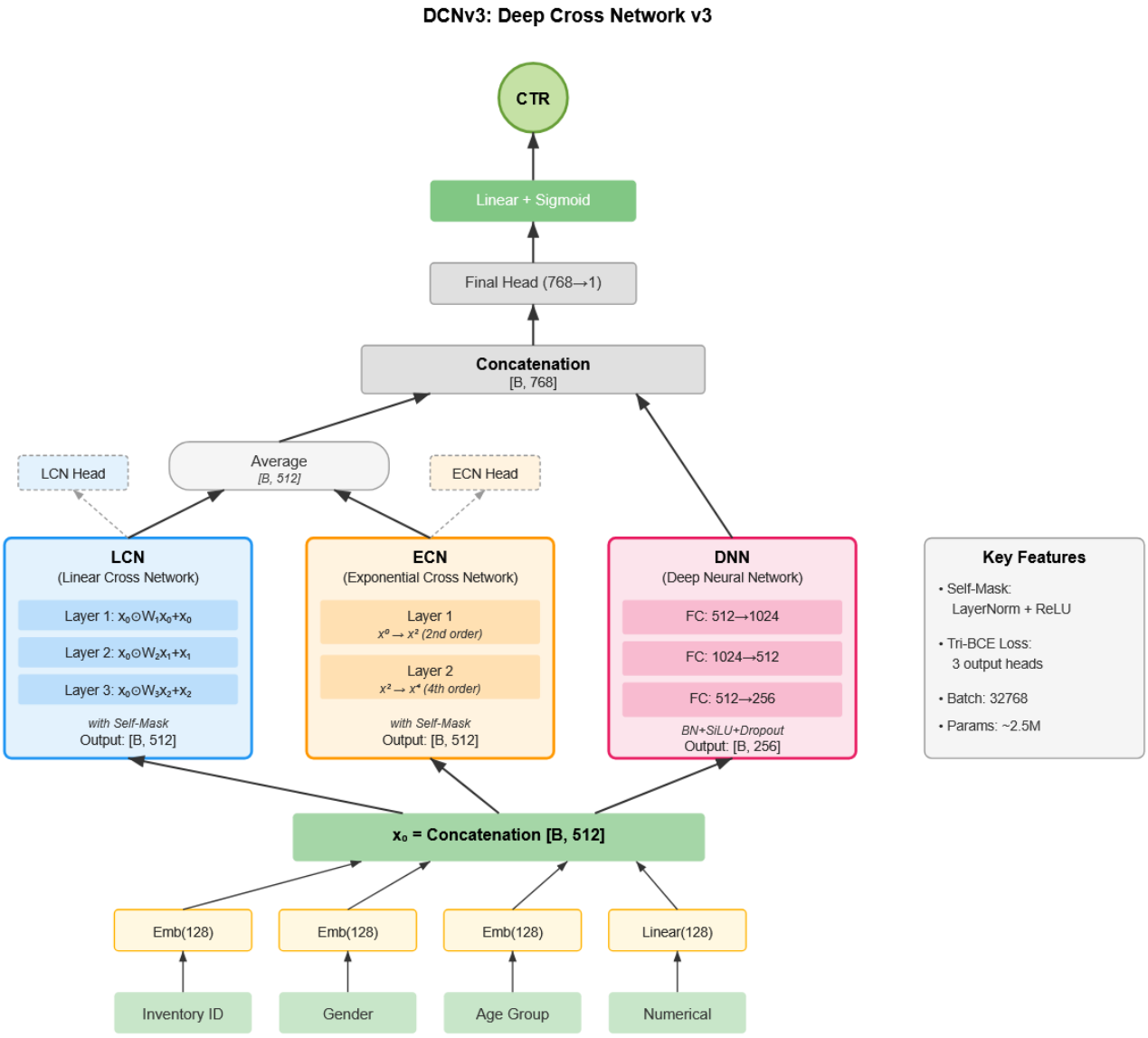


Figure 8. Visualization of neural network 2 - DCNv3 architecture.

A-3 Wide_Deep

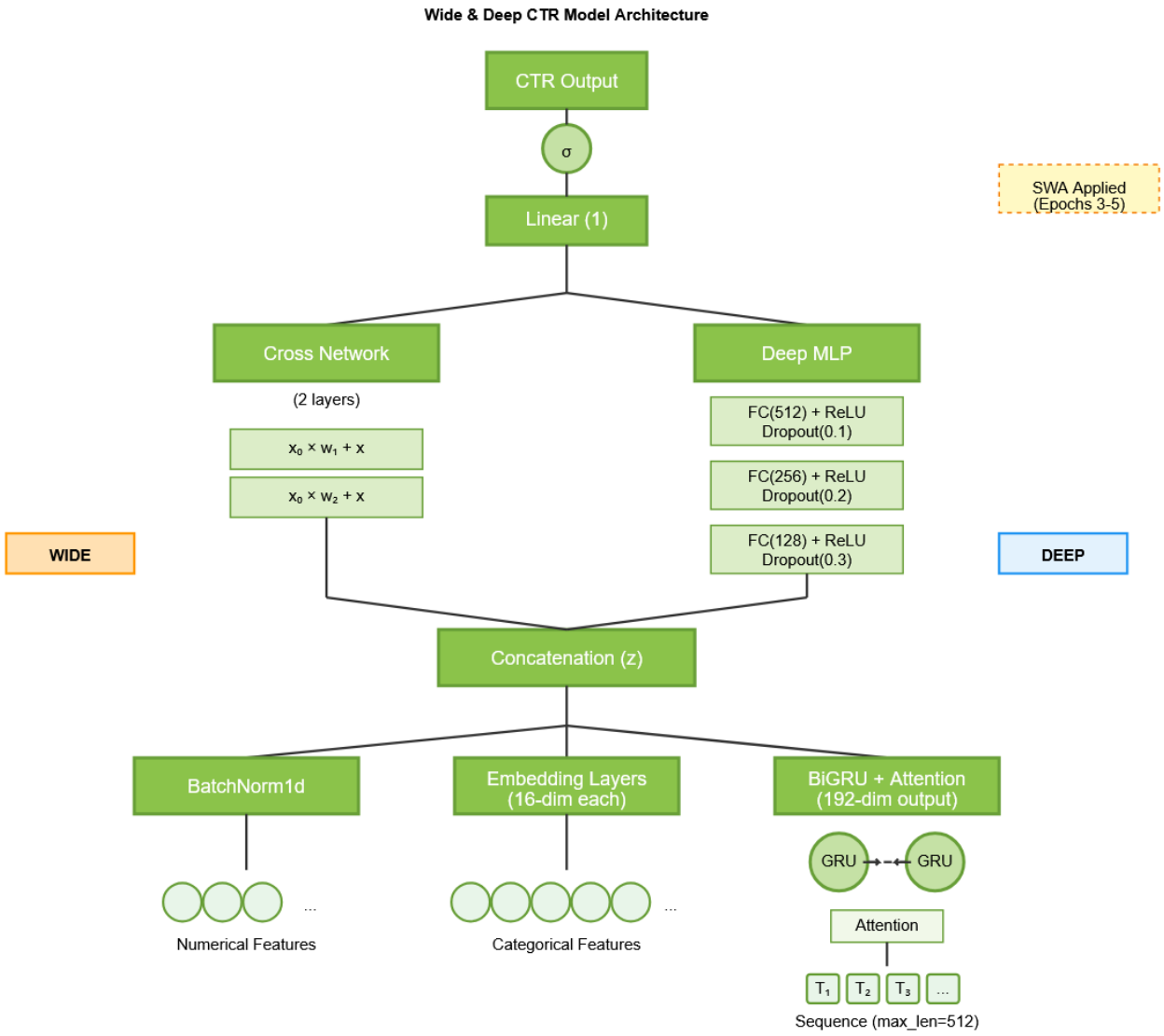


Figure 9. Visualization of neural network 3 - Wide_Deep architecture.