Wide & Deep Learning for Recommender Systems

https://arxiv.org/pdf/1606.07792

0. Introduction

- 추천 시스템에서는 memorization(과거 데이터에 기반한 규칙 암기)과 generalization(새로운 조합에 대한 추론) 두 가지 능력이 동시에 중요하다는 점이 핵심 문제로 제시됨.
- 기존 방식은 선형 모델 + cross-feature(교차항) 변환을 사용해 memorization은 잘 하지만, 일반화는 약함.
- 반대로 딥러닝 기반 방식은 일반화에는 강하지만 과거 특이 패턴을 기억하지 못하거나 over-generalization 문제가 있음.
- 따라서 두 방식을 결합하는 Wide & Deep Learning 프레임워크가 제안됨. 이는 memorization과 generalization을 동시에 달성하는 것을 목표로 함.
- 주요 기여:
  - Wide 모델과 Deep 모델을 공동 학습(joint training) 하는 구조 제안
  - Google Play 추천 시스템에 실제 적용 및 실험 결과 제시
  - TensorFlow 기반 구현을 제공

1. Overview

- 핵심 아이디어 요약

  Wide 모델(선형 + 교차항)과 Deep 모델(임베딩 + 신경망)을 병합하여 각각의 장점을 상호 보완하는 구조를 만드는 것.

- 모델 구조 요약

  • Wide 컴포넌트: 일반화 선형 회귀식 y=wTx+by = w^T x + by=wTx+b 형태, 원시 및 변환된 특징 입력

  – 교차항(cross-product) 변환을 통해 희소 feature 간 상호작용을 포착

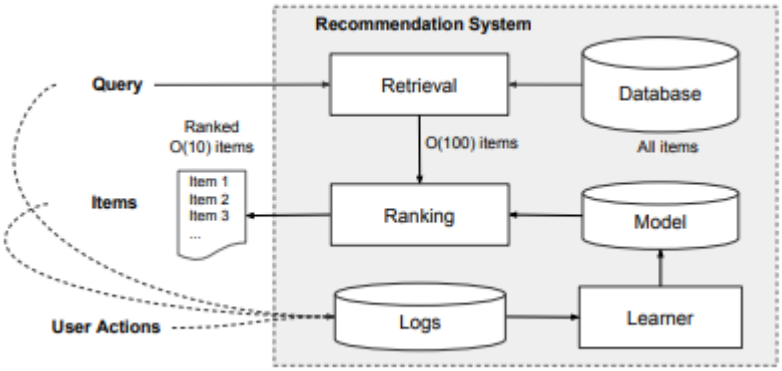
  - Deep 컴포넌트: 다층 퍼셉트론(MLP), sparse feature를 embedding으로 변환한 후 hidden layer를 통과
  - 두 모델의 출력을 weighted sum 형태로 결합하고, 하나의 로지스틱 손실 함수로 공동 학습
- 데이터셋 및 환경

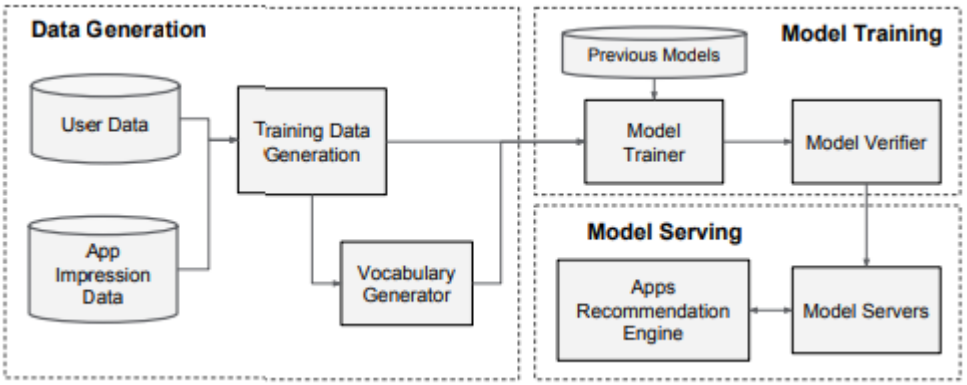
  Google Play 앱 추천 환경에서 실험 수행 (광범위한 사용자 기반, 수백만 개의 앱)
- 연구 목표 및 기대 효과
  - 기존 wide-only 또는 deep-only 모델 대비 추천 정확도 및 수익성 향상
  - 실제 운영 환경에서의 지연(latency) 제어 및 효율성 확보
  - 추천 시스템에서 memorization + generalization 동시 확보

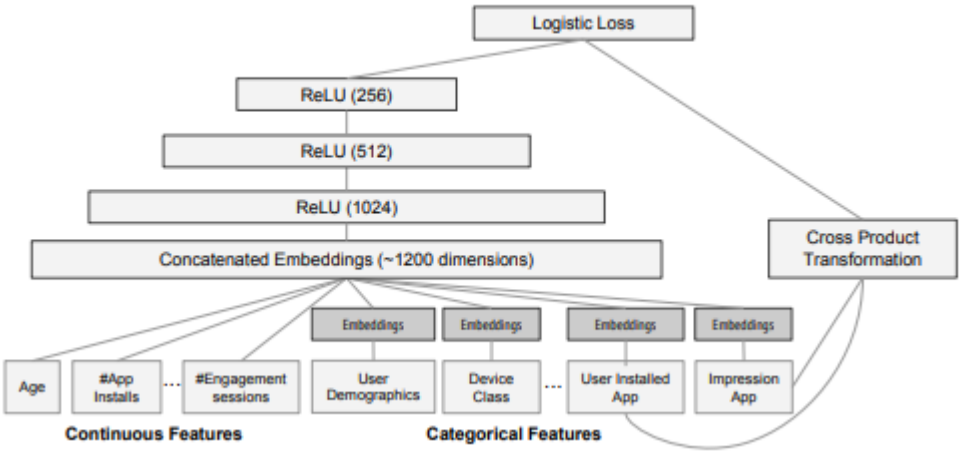
2. Challenges

- 기존 접근 방식의 한계
  - 선형 모델 + 교차항 방식은 memorization은 잘 하지만 일반화가 약함
  - 딥러닝 방식은 일반화는 잘하지만 예외적 패턴을 기억하기 어려움 → over-generalization 위험
  - 희소(sparse), 고차원(high-dimensional) feature 공간에서 embedding 학습이 어려움
- 모델 설계 및 학습 문제
  - 두 모델을 효과적으로 조합하는 방법 (가중치, 손실 함수 등)
  - 학습과 추론 시 지연(latency) 제어
  - 대규모 환경에서의 병렬 처리 및 serving 성능 유지
- 운영 및 확장성 문제
  - 실시간 추천 시스템 환경에서의 latency 요구
  - 매일 변화하는 사용자 행동과 신규 항목에 대한 빠른 적응 필요
  - 대용량 모델의 메모리 및 연산 비용 관리

3. Method







- **Wide 컴포넌트**

  입력 xxx(원시 + 변환된 특징)을 이용한 선형 모델 ywide=wTx+by_{wide} = w^T x + bywide=wTx+b.

  교차항(cross-product transformations)을 사용하여 feature 간 상호작용을 명시적으로 포착.

- **Deep 컴포넌트**

범주형(sparse) feature를 embedding vector로 변환하고, 다층 퍼셉트론을 통과시킴.

여러 hidden layer와 ReLU 활성화 함수를 사용하여 비선형 관계를 학습.

마지막 출력층에서 로지스틱 확률 예측.

- **Joint Training**

  Wide와 Deep의 출력(log odds)을 weighted sum 한 값을 최종 예측으로 사용.

  하나의 로지스틱 손실 함수(sigmoid + cross-entropy)로 두 모델을 동시에 학습.

  gradient가 두 부분에 모두 역전파되어 end-to-end 학습 가능.

- **Optimizer 및 규제**

  Wide 부분: FTRL(Follow-The-Regularized-Leader) + L1 정규화 사용.

  Deep 부분: AdaGrad 사용.

- **시스템 구현**

  Data generation, feature 변환, 학습, serving pipeline 포함.

  Serving 시 멀티스레딩(batch 분할) 전략을 통해 지연을 줄임.

  예: 단일 batch 처리 31 ms → 병렬 처리 후 14 ms로 감소.

- **Warm-start & 모델 업데이트**

  이전 모델의 파라미터를 초기값으로 활용하여 새 모델 학습 시 빠른 수렴 유도.

4. Experiments

- 데이터 설명

  Google Play 앱 다운로드/노출/클릭 로그 기반.

  입력 특징: 사용자 특성, 문맥(context), 앱 노출 및 행동 이력 등.

- 실험 설계

  온라인 A/B 테스트 및 오프라인 평가 병행.

  비교 대상: wide-only 모델, deep-only 모델, baseline 모델.

  각 사용자 그룹을 통제군과 실험군으로 구분.

- 평가 지표 및 방식

  주요 지표: 앱 획득률(acquisition), 클릭률.

오프라인 지표: AUC.

온라인 실험으로 실제 사용자 반응 평가.

- 추가 실험

  배치 크기, 스레드 수 변화에 따른 latency 비교.

  Ablation study: wide 또는 deep 구성 요소 제거 시 성능 변화 확인.

  Warm-start 효과 검증.

5. Results

Table 1: Offline & online metrics of different models. Online Acquisition Gain is relative to the control.

Model	Offline AUC	Online Acquisition Gain
Wide (control)	0.726	0%
Deep	0.722	+2.9%
Wide & Deep	0.728	+3.9%

Table 2: Serving latency vs. batch size and threads.

Batch size	Number of Threads	Serving Latency (ms)
200	1	31
100	2	17
50	4	14

- 정량적 성능 비교

  온라인 실험에서 Wide & Deep 모델은 wide-only 대비 앱 획득률 약 +3.9% 향상.

  Deep-only 모델 대비 약 +1% 추가 이득.

  오프라인 AUC 지표에서도 소폭 향상 관찰 (wide-only 대비 +0.002, deep-only 대비 +0.006 수준).

- 지연(latency) 성능

  대규모 batch 처리 시 31 ms → 병렬 처리로 14 ms까지 개선.

- Ablation/구성 변경 실험

  wide-only, deep-only 제거 시 성능 감소 확인 → 두 구조의 상호 보완 효과 입증.

- 한계 및 관찰

  오프라인 성능 향상 폭은 크지 않으나 온라인 환경에서는 유의미한 변화 발생.

  서버 지연과 자원 소모 사이의 trade-off 존재.

6. Insight

- 추천 시스템에서 memorization과 generalization을 동시에 달성할 수 있는 구조를 제시했다는 점에서 의미가 큼.
- 실제 제품 환경(Google Play)에 적용해 실사용자 데이터를 기반으로 유의미한 성능 개선을 입증.
- 두 모델의 강점을 결합해 더 안정적인 추천 가능
- latency와 효율성을 모두 고려한 실무 친화적 설계
- 오픈소스 구현으로 재현 및 확장 용이
- feature engineering 의존도 여전히 높음
- 복합 모델로 인해 메모리·연산 부담 증가
- cold-start 문제 해결에는 한계 존재
- offline-online 성능 간 괴리 가능성 존재
- 추천 시스템에서는 단일 모델보다 혼합 구조가 효과적일 수 있음
- latency 제약이 강한 환경에서는 병렬 처리 전략 필수
- feature engineering과 임베딩 설계의 균형 유지 중요