Training Deep AutoEncoders for Collaborative Filtering

13기 조상연

ABSTRACT

목적: 추천 시스템에서의 별점 예측 모델 제안

데이터: 넷플릭스 유저 평점 데이터

핵심 인사이트

1. 오토인코더의 레이어가 깊은 것이 얇은 것보다 훨씬 좋다.
2. 음수 부분의 Non-linear 활성함수가 굉장히 중요하다. (SELU)
3. Overfitting 방지를 위해서 Dropout과 같은 정규화 기법이 절실하다.

새로운 제안: Sparseness 를 극복하기 위한 새로운 학습 알고리즘 제안함

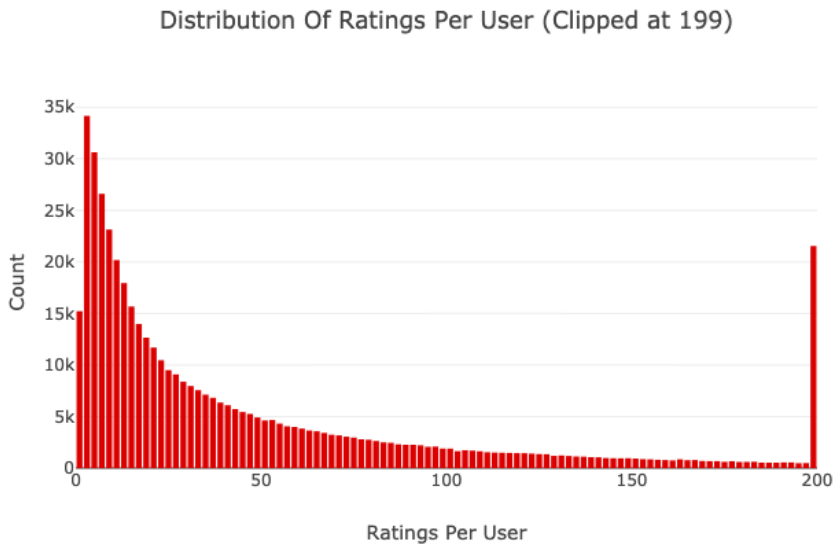
Introduction

수많은 서비스들이 추천 시스템을 사용 중

그 중 Netflix Prize 데이터를 이용하여 전통적인 CF(Collaborative filtering) 를 오토인코더를 통해 시도함

데이터 개요

1. 총 2400만 평점 데이터
2. 데이터 구조: User(id) – Movie(id) – Rating(1:5)
3. 총 17,770 개 영화 / 총 47 만 유저
4. 대부분 2~5개 리뷰를 남김, 매우 Sparse 함



관련 연구

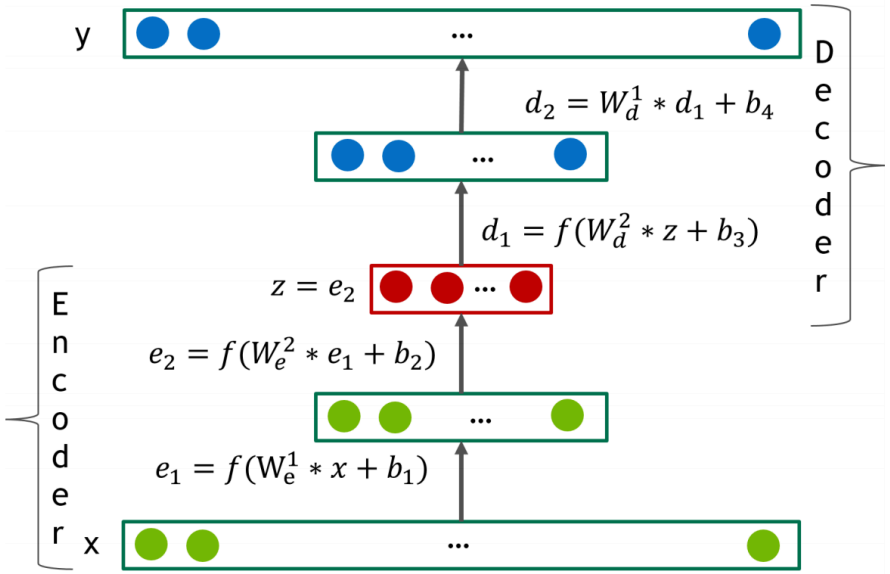
1. AutoRec 그 중에서도 User-based AutoRec에서 감명 받음

사진 출처: https://www.kaggle.com/morrisb/how-to-recommend-anything-deep-recommender

Model

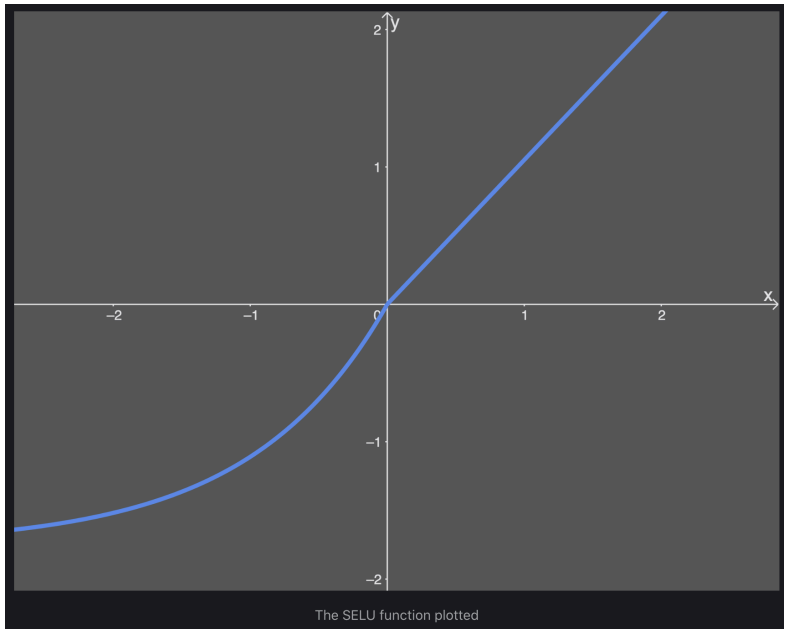
사용 모델

1. Auto-Encoder를 사용한다.
2. Encoder에 해당 유저의 영화별 평점 Matrix가 들어가고 하나의 Latent Vector로 변환된다.
3. Decoder에선 해당 Latent Vector를 변환하여 원래의 Input값이 되도록 한다.



차별화된 점

1. 기존 논문보다 더 Layer를 쌓았다.
2. SELUs 란 활성함수를 사용하였다.
3. 매우 높은 Dropout Rate를 적용하였다. (0.8)
4. 학습 중 반복적 Output Re-feeding 기법을 적용하였다. (중요)



SELU

Iterative output re-feeding

핵심

1. 처음 Input으로 예측된 값으로 다시 학습을 진행함
2. 한 Iteration 내에서 여러 번도 가능
3. 이를 통해 성능 향상 (아래 초록색 선)

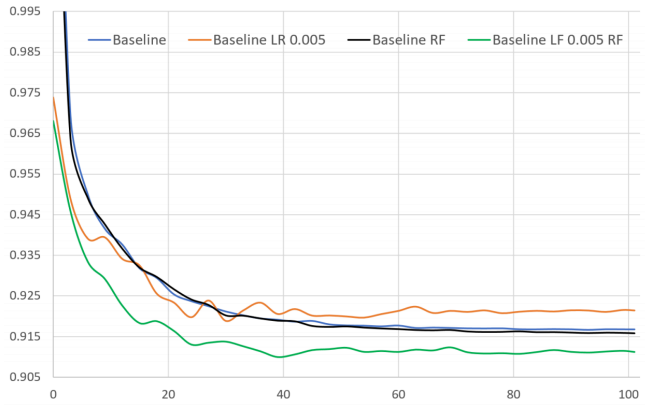


Figure 5: Effects of dense re-feeding. Y-axis: evaluation RMSE, X-axis: epoch number. Baseline model was trained with learning rate of 0.001. Applying re-feeding step with the same learning rate almost did not help (Baseline RF). Learning rate of 0.005 (Baseline LR 0.005) is too big for baseline model without re-feeding. However, increasing both learning rate and applying re-feeding step clearly helps (Baseline LR 0.005 RF).

처음 학습하는 부분

다시 학습하는 부분

- Variable(outputs.data)로 이전에 Inference된 결과를 가져옴
- Aug_step만큼 반복할 수 있음, 즉 f(f(f(x))) 도 가능

```
for i, mb in enumerate(data_layer.iterate_one_epoch()):
  inputs = Variable(mb.cuda().to_dense() if use_gpu else mb.to_dense())
  optimizer.zero_grad()
  outputs = rencoder(inputs)
  loss, num_ratings = model.MSEloss(outputs, inputs)
  loss = loss / num_ratings
  loss.backward()
  optimizer.step()
  global_step += 1
  t_loss += loss.item()
  t_loss_denom += 1

  if i % args.summary_frequency == 0:
    print('[%d, %5d] RMSE: %.7f' % (epoch, i, sqrt(t_loss / t_loss_denom)))
    logger.scalar_summary("Training_RMSE", sqrt(t_loss/t_loss_denom), global_step)
    t_loss = 0
    t_loss_denom = 0.0
    log_var_and_grad_summaries(logger, rencoder.encode_w, global_step, "Encode_W")
    log_var_and_grad_summaries(logger, rencoder.encode_b, global_step, "Encode_b")
    if not rencoder.is_constrained:
      log_var_and_grad_summaries(logger, rencoder.decode_w, global_step, "Decode_W")
    log_var_and_grad_summaries(logger, rencoder.decode_b, global_step, "Decode_b")

  total_epoch_loss += loss.item()
  denom += 1

  #if args.aug_step > 0 and i % args.aug_step == 0 and i > 0:
  if args.aug_step > 0:
    # Magic data augmentation trick happen here
    for t in range(args.aug_step):
      inputs = Variable(outputs.data)
      if args.noise_prob > 0.0:
        inputs = dp(inputs)
      optimizer.zero_grad()
      outputs = rencoder(inputs)
      loss, num_ratings = model.MSEloss(outputs, inputs)
      loss = loss / num_ratings
      loss.backward()
      optimizer.step()
```

https://github.com/NVIDIA/DeepRecommender/blob/a32a8a5c23092c551616acf6fac5b32e1155d18b/run.py#L189

Conclusion

1. Netfilx 데이터에서 여러 모델보다 우수한 성적이 나왔다.

2. 시간적 길이가 길수록 깊고 더 많은 파라미터가 필요하다.

3. 왜인지는 모르겠으나 그렇게 자랑하는 re-feeding을 결국 1번만 해서 모델을 완성했다. Re-feeding도 그 자체만으론 효과가 적고 Lr를 0.005 정도로 높게 주었을 때 효과가 좋았다는 언급은 있다.

Table 3: Test RMSE of different models. I-AR, U-AR and RRN numbers are taken from [19]

DataSet	I-AR	U-AR	RRN	DeepRec
Netflix 3 months	0.9778	0.9836	0.9427	**0.9373**
Netfix Full	0.9364	0.9647	0.9224	**0.9099**

Table 4: Test RMSE achieved by DeepRec on different Netflix subsets. All models are trained with one iterative output re-feeding step per each iteration.

DataSet	RMSE	Model Architecture
Netflix 3 months	**0.9373**	n, 128, 256, 256, $dp(0.65)$, 256, 128, n
Netflix 6 months	**0.9207**	n, 256, 256, 512, $dp(0.8)$, 256, 256, n
Netflix 1 year	**0.9225**	n, 256, 256, 512, $dp(0.8)$, 256, 256, n
Netfix Full	**0.9099**	n, 512, 512, 1024, $dp(0.8)$, 512, 512, n

SELU