Using Transformer Neural Nets to Generate Good Sequences in Recommender Systems

Luke Rabbitte

A Final Year Project Report

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Bachelor of Arts in Computer Science and Language

Supervisor: Douglas Leith

April 2024

Using Transformer Neural Nets to Generate Good Sequences in Recommender Systems

Luke Rabbitte, Bachelor of Arts in Computer Science and Language

University of Dublin, Trinity College, 2024

Supervisor: Douglas Leith

This work presents a novel implementation of a sequence-aware recommender system (RS) based on the Decision Transformer model. Despite limited precedent, we successfully adapted this reinforcement learning-oriented model for RS tasks. Our approach spanned generation of suitable training data, modification of the Decision Transformer architecture, and creation of a suitable evaluation process for recommended items. We train models to support slate recommendation with high Normalised Discounted Cumulative Gain (NDCG), and sequential recommendation with high Cumulative Gain (CG). Several limiting factors to Decision Transformer-based RS are surfaced and discussed as areas of essential future work for the field.

Acknowledgments

One day I will look back and realise all the wider love and support that made my time at TCD possible.

For now I will thank those who saw me through the quieter moments, particularly during the long nights of my final year. To both Nanas, Eleanor, Mom, Dad, Arron, Ellen, Caroline, as well as my closest friends - thank you for seeing things I sometimes lose sight of in myself.

Special thanks also go to Douglas Leith and Dilina Rajapakse for supervising and inspiring me throughout my work.

Luke Rabbitte

University of Dublin, Trinity College
April 2024

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recommender systems (RS) lie at the centre of modern user-centric system design, and attempt to solve the problem of information overload. Since the field's early commercialisation from internet companies such as Netflix and Amazon in the 1990s, RS has been as much an industry as an academic concern. Systems are often designed around business metrics - news RS designed to maximise click-through rate on articles, e-commerce RS optimised for customer conversion, and social media RS focused on user retention, to take three examples.

At their best, RS can simplify the user experience of applications and offer suggestions respecting the intellectual growth and developing tastes of a user. It is for this reason that the present research focuses on *sequence-aware* RS - where recommendations are generated with respect to a given sequence of interactions - rather than traditional RS, which take the user's taste to be unchanging over time.

In considering a sequence-aware RS, we naturally fall upon the terminology of *reinforcement learning* (RL), which models sequences as *actions* taken by an agent in an environment according to the observed *state* of the environment, thus receiving *rewards* based on the quality of these actions. This pattern of interaction is known as a *Markov chain* and is inescapable in the literature. Reinforcement learning is interesting to RS because it treats decision-making in a changing environment as a first-class concept.

The motivation for the last of the three intersecting domains of this research, the use of the *transformer* architecture, arises for various other reasons. Transformers are a deep learning architecture designed for sequence-to-sequence tasks, and are deeply rooted in the field of Natural Language Processing (NLP). Famously, they lie behind *Generative Pre-trained Transformers* (GPTs) - to which we feed tokens of input text and receive a probability distribution of the most likely subsequent tokens.

This framework turns out to be surprisingly powerful beyond language. As such, a

growing portion of the RL community has begun to investigate whether the transformer is powerful enough to accept RL-style sequences of data resembling a *Markov chain* and by doing so completely bypass the need to apply traditional RL algorithms. The results, at least for well-established RL tasks, are competitive with the state of the art. Not only this, but transformer models appear to be exceptionally adaptable to disparate modalities; one particular study shows how a single set of weights in a transformer neural network can play Atari games, caption images, generate text, and control a robot arm [1].

1.1 Motivation

Thus, we arrive at the motivation of the present research. Given that recommendation is not a well-established RL task, we explore whether or not this work can be ported to the modality of RS. This would yield a sequential awareness of user ratings akin to reinforcement learning, with a far simpler and more scalable training setup. In order to explore this question, we need to select appropriate recommendation datasets, adapt and train an existing transformer-based model to work for these datasets, and evaluate the ability of this model to produce sequences of recommendations likely to be highly rated by users.

1.2 Problem Formulation

Let U be a user and I be the set of all possible items that can be recommended. At each timestep t, the recommender system suggests a new item $i_t \in I$ to the user U. The user interacts with the item and provides a rating r_t in the range [1,5]. The goal of the recommender system is to select a sequence of items $i_1, i_2, \ldots, i_t$ over t timesteps to maximize the cumulative rating $\sum_{t=1}^{T} r_t$, where T is the desired length of the recommendation sequence.

Chapter 2

Literature Review

This chapter offers background and discussion on traditional RS, RL, and transformers, as well as a review of relevant literature and an introduction to some of the common problems encountered in the field.

2.1 Traditional Recommender Systems

Collaborative and content filtering are two of the most dominant techniques seen across the field of RS. Though now widespread, they represent some of the most important early advancements in RS, and introduce concepts important in the design of our own RS.

2.1.1 Collaborative Filtering

Collaborative filtering relies on the underlying assumption that the rating a user will give to a previously unseen item is similar to the rating that other like-minded users have given to this item. In its simplest application, this can involve algorithms which take the average of the ratings given by the top-n most similar users in a system as the predicted rating for the present user, where the similarity is given via cosine similarity, Pearson correlation, or another suitable method.

Collaborative filtering systems can also involve some form of matrix factorisation, where the matrix of user-item interactions is decomposed into two smaller matrices representing users and items, and the predictions are modelled as the dot product between the corresponding user and item vectors in a latent feature space. Matrix factorisation techniques, such as Singular Value Decomposition (SVD) and Probabilistic Matrix Factorisation (PMF), are used to discover these latent features underlying the interactions between users and items. These latent features might represent inherent characteristics

of the items (i.e. genres, budget, director for film recommendations) or other user preferences.

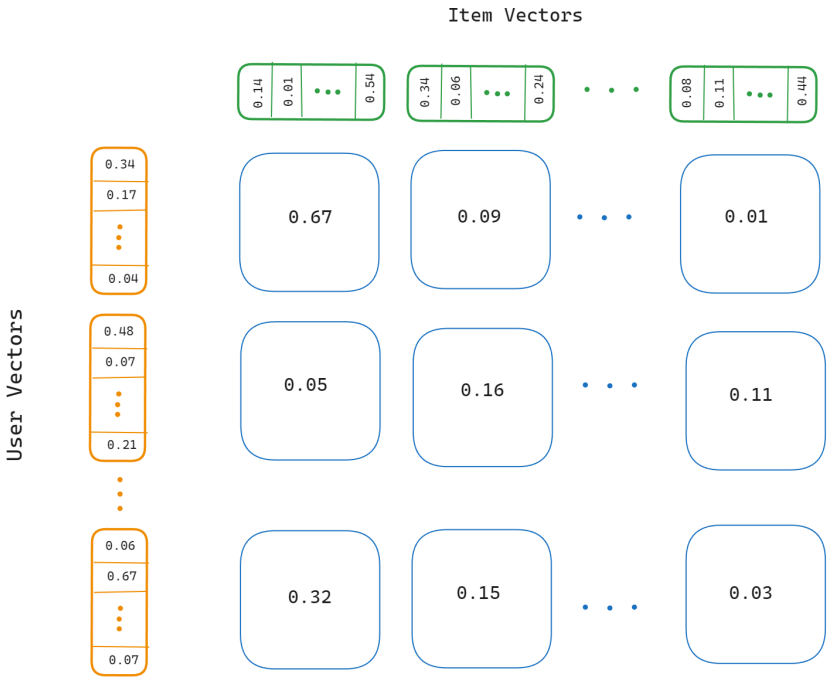


Figure 2.1: A simplified depiction of collaborative filtering, where ratings (blue) can be found as the dot product between a user's embedding vector and the item's embedding vector.

However, most collaborative filtering algorithms assume a comparatively *high-density* dataset, where density is the ratio of actual ratings in the dataset to the total number of ratings that could have been given, according to the number of users and the number of items. They are thus ill-suited to *sparse* data, a problem particularly for new users and new items entering the system with no history. As we will see, the density of the data at hand informs much of the design of RS.

2.1.2 Content Filtering

Another traditional method in RS, *content filtering*, involves gathering detailed metadata both on the items and the users in the system. Items are recommended based on the proximity of the content to the user's taste, as well as possibly being supplemented with external information such as time of day, current location, or even current weather. Many online platforms make an effort to build up this metadata before the user has even interacted with the system, asking the user to rate sample items, complete questionnaires, fill out checklists, etc. This helps to address the problem of integrating new users into

the system, known as the cold-start problem, as found in collaborative filtering, yet introduces a requirement for close domain knowledge of the problem at hand when it comes to manually assigning features to an item.

2.1.3 Hybrid Approaches

When building RS at scale, it becomes proportionally harder to incorporate domain knowledge describing the features of each item. In the current work, we seek to implement an RS that takes inspiration from collaborative filtering by learning a representation of users and items rather than using these directly, and also a system that avoids the scalability problems inherent to content filtering approaches which rely on data with manually labelled features.

There are certain *hybrid* algorithms that blend aspects of both collaborative and content filtering. Examples such as Probabilistic Memory-Based Collaborative Filtering (PMCF) reduce memory and other computational demands by considering only a selected subset from the history of user ratings, mixing this with knowledge of user profiles, and perform favourably compared to Pearson correlation and model-based algorithms such as naïve Bayes [2]. Although training on a given subset lessens the memory and processing demands on the system, the ever-growing datasets generated by modern platforms demand an alternative which achieves a similar computational gain through more efficient processing alone, without losing information.

2.2 Reinforcement Learning

In order to understand the more complex methods of efficient training, we first consider basic applications of RL to RS. Discussing RL algorithms grounds us in a simple yet powerful framework of describing sequential decision making in an environment. Perhaps the most significant benefit of considering RL in designing our RS is that it provides first-class support for user cold start, building up a user profile from scratch based on their growing interactions.

2.2.1 Mutli-armed Bandits

Multi-armed bandit algorithms provide an elegant expression of the explore-exploit trade-off, something absent from traditional RS that is crucial in any sequential decision-making process. With multi-armed bandits, the exploitation of the user's existing taste is balanced with exploration of items that may fall outside this taste, with the aim being to maximise

value over a certain time window [3]. The ϵ-greedy algorithm, for example, builds a ranking of a set of documents based on observations of past rewards. With a frequency ϵ, it will recommend a random document; otherwise, it will recommend the known highest-ranked item [4]. A less binary version of this involves using the softmax policy; a value probability is given across all documents, and documents are sampled according to a given *temperature* hyperparameter τ. Setting a high τ corresponds to more random document selections, with a low τ meaning the document with highest known value is almost always chosen [5].

While such algorithms sufficiently address difficulties such as cold-start, and successfully handle the trade-off between immediate and future rewards, they do not capture the additional richness of a true reinforcement learning scenario whereby actions taken can in turn influence the environment and reward. This oversight is harmful when it comes to RS, as it assumes the user's taste to be unchanging over time. True RL algorithms, especially deep reinforcement learning algorithms such as deep-Q networks (DQN) and actor-critic networks, are better able to handle dynamic environments. Both algorithms use a neural network to better handle high-dimensional input. A DQN model's end goal is to maximise reward over a given timeframe, usually in a future-discounted manner. It is *value-based*, meaning it maps actions taken in a given state to their expected value, rather than *policy-based*, where actions are directly outputted. Actor-critic algorithms, on the other hand, use two networks, the actor being policy-based and the critic being value-based, and use output from the critic network as feedback in training the actor network. Both algorithms use deep neural networks to efficiently capture high-dimensional input and scale to larger datasets [6].

We take inspiration from the deep RL paradigm in our current RS, particularly when considering the amount of reward we desire over a given timeframe, and in the facilitation of a dynamic environment, where the user's taste can change over time. However, we seek to simplify the problem further. Instead of deliberately fitting a policy to predict actions, we explore whether the sequence-modelling capabilities of the transformer architecture can generate optimal actions directly. Transformers are chosen instead of the deep neural networks found in DQN and actor-critic networks primarily due to their greater computational efficiency. Their introduction allows us to shift not only the data handling power but also the decision-making power to one place, allowing for a more elegant and efficient sequential RS. This helps us remove complex RL algorithms from our system design entirely.

2.3 Reinforcement Learning Using Transformers

In moving from RL algorithms to a transformer-only model, it is natural to predict actions rather than predicting rewards. Where methods like DQN mapped states and actions to a desired sum of rewards, or return, we take inspiration from Upside-Down Reinforcement Learning (UDRL), in which the desired rewards over a sequence of interactions, as well as the current state, become the 'task-defining input' of our system [7] mapping to a desired action. This is helpful when building RS, as we can generate new recommendations simply by specifying how much cumulative reward we want over a specified episode of recommendations, as well as the current state of the system.

A key reference for the current work is the Decision Transformer architecture [8], where reinforcement learning problems are abstracted as a sequence modelling task. Similar to UDRL, Decision Transformer predicts actions rather than rewards, and does so by specifying desired return over a given time horizon. Its main contribution is in giving the theories behind UDRL a concrete implementation in an auto-regressive GPT model. The model promises to generate optimal actions, even when trained on sub-optimal data, and to remove the RL overhead associating with intentionally learning a policy.

An in-depth analysis of Decision Transformer is spared for the technical Section 3.1 of this report, however it suffices here to attribute much of the power of Decision Transformer to the *attention* mechanism of transformers. This enables each element of an input sequence to be weighted according to how influential it is in generating new elements, allowing the model to capture complex, long-term dependencies in data. It also provides a natural answer to the *credit-assignment* problem traditionally seen in RL, where we attempt to determine which past actions were most responsible for a distant future reward.

Precursors to the transformer such as Recurrent Neural Nets (RNNs) are available, but handle sequences with less efficiency. In an RNN, each step in the sequence can only be reached in linear time, combining many dot products' worth of relevant information propagated from previous matrices in the current. Due to this simpler handling of sequences, RNNs may also place too high a priority on directly adjacent items, missing the nuance of item sequences having dependencies that are more far reaching, interspersed with distractor signals [9]. Even among transformer models, alternatives such as the Bidirectional Encoder Representations from Transformers (BERT) model exist, which use an input masking technique borrowed from NLP to handle input data bidirectionally rather than unidirectionally, as is the case with GPT and others [10]. During training, BERT learns to predict artificially concealed input tokens scattered throughout the sequence, learning the context in which certain items are recommended. Though adaptable to RS, BERT models do not learn from a user's instant feedback in the same way as RL-based

models, and thus represent a different line of research.

2.4 Mapping Between RL and RS

In extending the Decision Transformer architecture to the area of recommendation, we quickly find need for a succinct map between RL and RS concepts. This is a difficult task and any attempt to implement a RS using RL, or even RL-style input data, as is our focus, typically falls victim to a number of open problems identified within the literature. The most pressing of these problems include finding a suitable representation of the *state*, and building a plausible environment for *evaluation* of the model's performance [11]. Formulating *reward* is also a problem, although far more pressing for RS with implicit ratings tracking behavioural impulses of users such as clicks and watch times; we mitigate this by only working on RS with explicit ratings (see Section 3.2.6).

Decision Transformer is by no means plug-and-play with RS data. Table 2.1 previews the mapping between common RL terms and concepts as required by Decision Transformer and their presumed RS equivalents. The inherent modal differences between classic RL data and popular RS data made this mapping one of the biggest challenges of this research. The table is presented here for the convenience of the reader, with some of these challenges introduced in the following subsections. However, the mapping is further explained in the technical discussion in the following chapter and beyond.

Concept in RL	**Concept in RS**
State	One-hot encoding of the user group
Action	Item stored in the system
Reward	User's rating of a given item
Transitions	Interactions
Episode	User's interactions from start to finish
Terminal State	User's final or most recent interaction
Return	Sum of ratings across an entire episode
Return-to-go	Sum of ratings across the remainder of the episode, from the current position

Table 2.1: Mapping from Reinforcement Learning (RL) to Recommender Systems (RS)

At the basic level, we consider recommendation as a Markov process in which the agent is the RS, the environment is the user, and the reward passed between the two is the rating that users give to recommended items.

2.4.1 State: A Hard-to-Map Concept

State in a Markov process is some rich observation of the underlying environment at a given moment in time. In fact, the Markov property holds that future states must be reached by looking at a single past state, rather than a long history of past states. On a small scale, some RS considers the state of the system to be the current item under consideration. However, this quickly becomes difficult to scale, especially in modern platforms with ever-increasing item spaces. States thus typically represent some indication of a user's preference, such as their previous activity up to a given timestep. In a browsing sequence, for example, each timestep of state can consist of a tuple of the item recommended, the user's feedback, and the user's dwell time [12]. While much of the literature indicates that state should explicitly model the interaction history between users and items, there is a lack of consensus on how much history should be included [13] as well as a severe lack of study of state representation for the particular application of transformer-based RL models. Early papers following the same research path as ours either likewise represent state as interaction history [14] or revert to relying on explicit user features and preferences [15]. In general, state representation in the field of RS is considered 'more art than science' [11]. For our solution, we focus on finding a suitably rich representation of the user's preference rather than simply loading in a certain number of past ratings, as our architecture already gives the model access to previous ratings when making every new recommendation.

2.4.2 Evaluation Without Online Deployment or RL-style Simulation

The Markov process demands feedback from the environment to be passed back to the model as new decision-making insight. The status quo for most RL research is to train in popular and well-established environments, often exposed via convenient libraries, where this feedback is given by a specialised and separately-trained model performing simulation of a real-world environment. The evaluation of a model is trivial, resembling the process shown in Figure 2.2.

Such libraries and simulation environments are far less common in the field of RS. In e-commerce systems, researchers can avail themselves of the *VirtualTB* environment, a simulation of online retail built upon the OpenAI Gym library. VirtualTB provides built-in evaluation metrics suitable for the problem modality, such as click-through rate [16]. Some methods used to create a simulation of the user in RS include Logistic Matrix Factorisation [17] as well as Adversarial User Models [18], the latter of which can be applied on the MovieLens dataset of user's five-star ratings of movies, introducing external

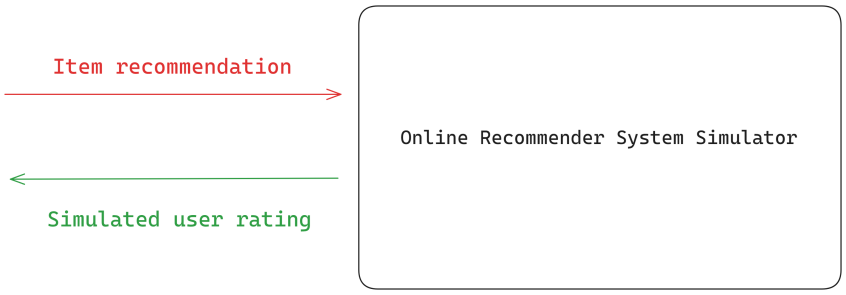


Figure 2.2: A simplified view of the sort of simulation environments exposed in traditional reinforcement learning research.

movie information and considering the context of a month's worth of movies released. PyRecGym is another proposed, yet not openly available, OpenAI Gym-inspired simulation environment for RS in the literature, implementing both a random user model as well as a multi-armed bandit-based model [19]. Most of these user-model based simulations are a compromise, failing to capture the complexities of real user interactions, and are not seen as reflective of an online production environment [20].

Implementing the best simulation available, we can explore the model's ability to generate good recommendations according to traditional recommendation metrics such as recall, precision, and Normalised Discounted Cumulative Gain (NDCG). Recall and precision are related in that they rely on a confusion matrix for binary classification. In a confusion matrix, an item successfully predicted to receive a high rating is a true positive, an item predicted to receive a low rating that in fact receives a high rating is a false negative, and so on as shown in Table 2.2.

	Recommended	**Not Recommended**
Actual Positive	True Positives (TP)	False Negatives (FN)
Actual Negative	False Positives (FP)	True Negatives (TN)

Table 2.2: Confusion Matrix of Classification for Recommended Items.

Recall measures the ratio of true positive classifications to the sum of true positive and false negative classifications, whereas precision measures the ratio of true positives to all predicted positives. Being rooted in binary classification, these metrics are not as suited to the explicit rating datasets we construct in this project.

NDCG, on the other hand, is a measure of the relevance and the rank appropriateness of a number of recommended items, giving more weight to the initial items. Section 3.2.7 indicates how we implement *slate* recommendation at each timestep in our model. While NDCG is used to assess the quality of these slate recommendations, our primary goal is to assess the evolution of our model across *many* timesteps, and as NDCG does not express

the changing nature of model performance over time, it is less important to our work.

In any formulation of RS as an RL problem, there is a trade-off between using these traditional RS metrics, which can be too short-term in their evaluation, and classic RL metrics such as cumulative rewards, which can be too sensitive to the accuracy of the simulation environment [18]. The question of a suitable evaluation metric for the type of sequential model we are implementing is very much an open question in the research [21], although we satisfy this in the current work by using a blend of the described concepts from RS and RL.

Of course, the best feedback we could receive for our generated recommendations would be from real users. In practice, though, only a select few organisations and teams have the resources available to deploy online experiments to production and perform experiments based on real user feedback [11]. Aside from such online environments, or the simulation environments as just described, the most valid alternative is to let certain splits or reformulations of offline data serve as a proxy for the opinions of real users. This, in fact, is the approach that best suits our work, and is described further in section 4.1.

Chapter 3

Design and Implementation

For the purposes of our research, we implement a Decision Transformer-based RS. This is a sequence-to-sequence model with an architecture inspired by natural language generation. During the initial phases of this project, our model was built directly on top of Decision Transformer. However, the Decision Transformer codebase includes boilerplate designed to load the Atari data for which it was intended. Our architecture thus builds directly on top of the GPT model that itself was used as a starting point for Decision Transformer [22].

The project is developed using the PyTorch machine learning framework in Python, designed to simplify the process of writing and running code intended for the GPU and expose efficient APIs for common machine learning tasks such as data loading, optimisation, and gradient computation [23]. The Pandas library was used extensively in generating synthetic datasets, manipulating tabular data, as well as gathering statistical insights on datasets via utility scripts [24]. Visualisations were plotted using the Matplotlib library [25] and further numerical analysis was conducted using the NumPy library [26]. All project dependencies were managed using Conda, a tool allowing for simple recreation of virtual environments in Python [27]. Git was used as version control software and was particularly useful in running experiments, where new branches were created to test experimental feature or a different set of hyperparameters.

3.1 Decision Transformer: Details and Critique

Decision Transformer, being an offline RL paradigm, is designed to accept either discrete or continuous data. Discrete data is the type of data generated in RL environments with a limited action space. This covers, for example, Atari video game data in which the agent is permitted to move in one of four directions. At every timestep in such a discrete

dataset, the action will be drawn from the set $\{0, 1, 2, 3\}$. Continuous data, on the other hand, is data generated from RL environments such as robotic control tasks, where the agent is a mechanical arm permitted to move freely through space in any given direction. This data takes the form of a real-valued vector.

Decision Transformer benefits from using the widely-studied Atari data as pioneered by researchers at DeepMind [28], who apply a Deep Q-Learning (DQN) algorithm to predict the future value, or expected rewards, of taking a certain action at a certain timestep. Unlike traditional Q-Learning, this algorithm can accept high-dimensional input state such as the pixels of an Atari game at any given timestep.

Interestingly, the source data for Decision Transformer originates from DQN playthroughs of the desired games, where the agent is acting in a online Markov process generating transitions between states, determined by actions determined by the Q-function at the given moment in training, and receiving certain rewards for doing so. In addition to receiving a reward at each timestep, the agent also receives a signal indicating whether the current 'episode' of transitions has reached a terminal state. For Atari data, the terminal state is well-defined: the end of a game. The authors note that the final training data takes one percent of the transitions generated by this agent, then repeats this until it has performed the same one percent collection for three different seeds.

This surfaces a potential weakness in the Decision Transformer model, in that the data it is trained on, from which all downstream performance originates, is generated according to exactly the type of traditional RL algorithm that it is designed to replace. It is also unclear from the original paper whether the transitions were generated from a pre-trained DQN agent, or collected over the course of training an agent from scratch. This could potentially have significant implications when it comes to adapting Decision Transformer for other modalities, such as is the goal of the present work, as there may not exist datasets with expert-level demonstrations for these modalities.

With access to this large offline dataset, Decision Transformer creates a circular replay buffer designed to efficiently pass the vast history of transitions into a data structure readily accessible from the core training loop. In this case, the states are bundles of four stacked frames of raw Atari pixel data (frame-stacking is a common approach in RL in situations where taking a single frame could fail to give an indication of some of the nuances of the modality, such as the direction of travel of moving objects). The actions are discrete, as described previously.

The rewards, however, are not used directly. Atari games, such as *Breakout*, represent a highly sparse reward environment where the agent can take many steps without receiving a reward. Table 3.1 shows an analysis of the sparsity of the rewards in the offline Breakout dataset.

Metric	Value
Mean episode length (no. timesteps)	1633.6
Mean return per episode	50.25
Sparsity of reward signal	0.036

Table 3.1: Reward Sparsity in an Atari Breakout RL Environment

Over the course of an episode in Breakout, the agent receives rewards only ≈3.6% of the time, representing an average of ≈59 timesteps between rewards. An important aspect of any RL setup is the problem of credit assignment - being able to adequately map good actions to the rewards they yield, even if there is a long delay between taking the action and receiving the reward. In Atari games, the sparsity of reward means that the greedy strategy of optimising for immediate reward often leads to a worse outcome than a strategy designed to maximise the total expected return over an episode. For this reason, Decision Transformer prefers returns-to-go, which represents the sum of the remaining rewards of the episode at each timestep.

Inside the training loop, batches of data are created by visiting a random index in the offline data and taking a window or context length's worth of data. This data consists of tuples of the form (s, a, r), where:

- s is the state,
- a is the action, and
- r stands is the return-to-go.

These batches are passed to the transformer model for processing. Before being forwarded through the model, states, actions, and returns-to-go are given an embedding. In the context of Decision Transformer, embeddings enable the model to capture the complex relationships between states, actions, and returns-to-go in a dense, continuous space. By transforming these discrete tokens into continuous embeddings, the model is able to learn more complex patterns across the data and generalise when it comes to unseen data. This is useful in the RL setting, where our agent often needs to make decisions based on states that it has not encountered before.

These embeddings are then passed through an auto-regressive GPT model designed to predict actions tokens at every timestep of the gathered (s, a, r) input tokens. The predicted action is compared to the ground truth action for that timestep, and this comparison informs the loss function according to which the model is trained. Figure 3.1 gives a visual intuition of this process, showing how a context's length worth of data is fed into the model, and how the corresponding actions are generated.

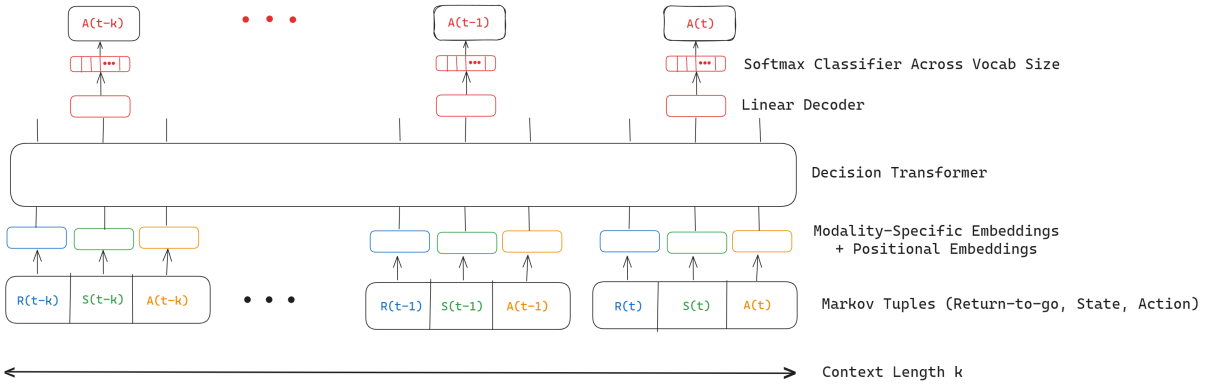


Figure 3.1: Decision Transformer-style architecture of our model, taking a context length of input timesteps in the form of (returns-to-go, state, action) tuples, giving these a suitable embedding, passing them through an auto-regressive GPT model, and generating a probability distribution across potential actions corresponding to each timestep of input.

When it comes to evaluating the model, Decision Transformer benefits from the existing Atari Learning Environment (ALE) library, which provides a standard interface for interacting with Atari games. The model's performance is evaluated at the end of every epoch of training by creating a game instance and letting it run through a certain number of episodes, and recording the total reward obtained in each episode. The average of these total rewards is then used as the performance metric, giving a measure of how well the model is able to maximize the total expected return over an episode.

3.2 Implementation

In applying Decision Transformer to the task of RS, we make significant changes to the type of data it is designed to accept, implementing new datasets and methods of data loading, finding acceptable embeddings for the new data modalities, and designing a sampling function that fulfills certain desired qualities of our RS.

3.2.1 Data Selection

In order to arrive at a suitable dataset for our training, we first select a category of data. Data in RS generally falls into two categories: implicit and explicit. Implicit ratings include actions such as article clicks, viewing time, or purchase history. These types of data are often abundant but can be noisy - for example, clicks and viewing time could be a result of sensational content, and purchase history could be confounded by purchases made for other people.

Explicit ratings, on the other hand, are direct indications of a user's preference, such as a star rating or a written review. This type of rating lends itself far better to the creation of a *sequence-aware* RS, as explicit ratings are naturally associated with discrete data points across a temporal span. Several considerations need to be made before deciding to rely on explicit data for a production system; for instance, far less data is collected than would be by logging passive behaviours. Explicit ratings also ask more of the user, essentially shifting onto them a layer of pre-processing in order to get a more declarative indication of their preferences. On the other hand, explicit rating gives the user more power in determining what recommendations they receive, and removes the more invasive need to track their passive behaviours over time, improving privacy. The costs of working with sparser explicit datasets is thus offset by the higher signal-to-noise ratio and enhanced user experience.

In order to arrive at a suitable dataset for our training, we consider both the density (the inverse of sparsity) and schema. The density of a dataset is expressed as the ratio of the total number of ratings to the total possible number of ratings for the given number of users and items present.

$$\text{Density} = \frac{\text{Total number of ratings}}{\text{Number of users} \times \text{Number of items}}$$

The densities of some popular explicit rating datasets are summarised in Table 3.2, for reference. We also show the density of a typical synthetic Goodreads dataset that we create for the purposes of our work, described in the upcoming Section 3.2.2. We achieve as similar a density as feasible, considering the relatively small action space.

Dataset Name	**No. Users**	**No. Items**	**Total No. Ratings**	**Density**
MovieLens	943	1682	100000	0.0630
Netflix	480189	17770	100000000	0.01172
Jester	59132	150	1700000	0.1918
Goodreads (Synthetic)	2048	273	99371	0.1777

Table 3.2: Density Statistics for Common RS Datasets as well as Our Synthetic RS Dataset

The schema of a dataset, on the other hand, refers to the structure and type of data our system is designed to record and absorb. Opting for explicit ratings, where items are rated on a scale from one to five stars, the schema followed across this project closely mirrors that of the MovieLens ratings dataset. After generalising further from movies to items, and from Unix timestamps to timesteps denoting the order of the rating within the user's history of interactions, we arrive at a suitable schema. Under this schema, the

column headers are shown in Table 3.3, and each row represents a unique rating given to a movie by a certain user.

user_id	**item_id**	**rating**	**timestep**

Table 3.3: Target Schema for our Datasets (Inspired by MovieLens)

The original MovieLens ratings dataset contains a corresponding user dataset containing demographic information for the users as well as a movie dataset containing metadata associated with each film. These datasets can be useful in designing algorithms for first-time users experiencing the cold-start problem. However, such data adds complexity to the system and, despite being anonymised, can still compromise user privacy. This was the case for Netflix data released as part of a crowdsourcing competition, where supplementary online information could be used to identify users [29]. Our system, in focusing simply on long sequences of interactions, avoids these issues and becomes easier to scale.

3.2.2 Dataset Generation and Pre-Processing

Within RS, users can be clustered according to their tastes and preferences. Importantly, we seek to incorporate an awareness of these user groups into our system. To achieve this, we use pre-existing data where users from popular datasets were grouped using the Monte-Carlo Tree Search (MCTS) algorithm [30]. For example, for the popular *Goodreads* book rating dataset, this algorithm yields the real-valued mean and variance of ratings of 273 different books across four different user groups. We use this mean and variance information to create a Gaussian distribution from which we can draw a random real number from the range $[1, 5]$. This number is rounded to the nearest natural number, serving as the rating a hypothetical user from this group has given the item in question.

This flexible approach lets us create synthetic recommender-style datasets with an arbitrary number of users, albeit with a fixed number of items. We closely mimic the number of ratings each user gave in real-world datasets; for the MovieLens datasets, for example, we observe that all users have rated above a certain base number of items, with a highly positive skew and a small number of outliers who have rated a large number of items.

In generating our synthetic data, we thus implement a Beta distribution, a continuous probability distribution defined on the interval $[0, 1]$. It is parameterised by two positive shape parameters, denoted by a and b, that appear as exponents of the random variable and control the shape of the distribution. The Beta distribution can become more or less positively skewed depending on the given values of a and b. It is useful in this context as it allows us to match the number of items each user rates to MovieLens or any other dataset

of our choosing. When adding a new user to our dataset, we determine the number of items they have rated, r, according to the formula

$$r = \lfloor \text{Beta}(a, b) \times (M - m + 1) + m \rfloor \tag{3.1}$$

where

- $\text{Beta}(a, b)$ is a random number drawn from the Beta distribution with parameters a and b (note, here $a = 1$ and $b = 12$, resulting in a highly positive skew),
- M is the maximum number of desired ratings per user, and
- m is the minimum number of desired ratings per user.

This synthetic distribution of rating counts is shown in Figure 3.2, and considering there are less items available for rating, it mirrors the positive skew of the distribution across a comparable real-world dataset as shown in Figure 3.3.

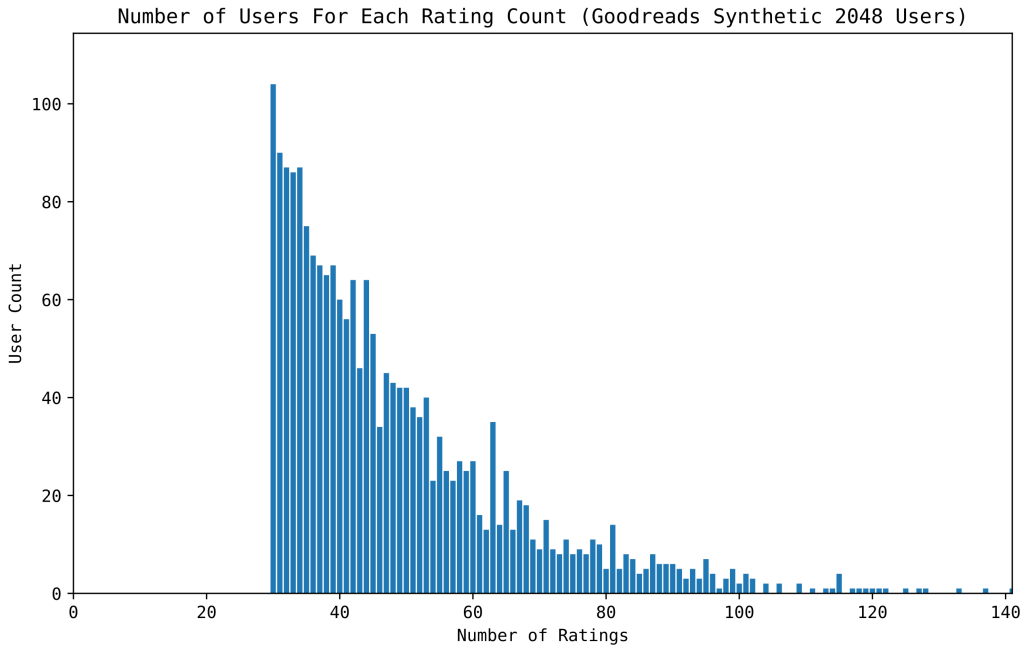


Figure 3.2: Rating distribution for users in our synthetic Goodreads dataset

3.2.3 Data Loading

One of the challenges in feeding recommender data into a Decision Transformer-style model lies in how we define an episode of interaction between the user and agent. It is

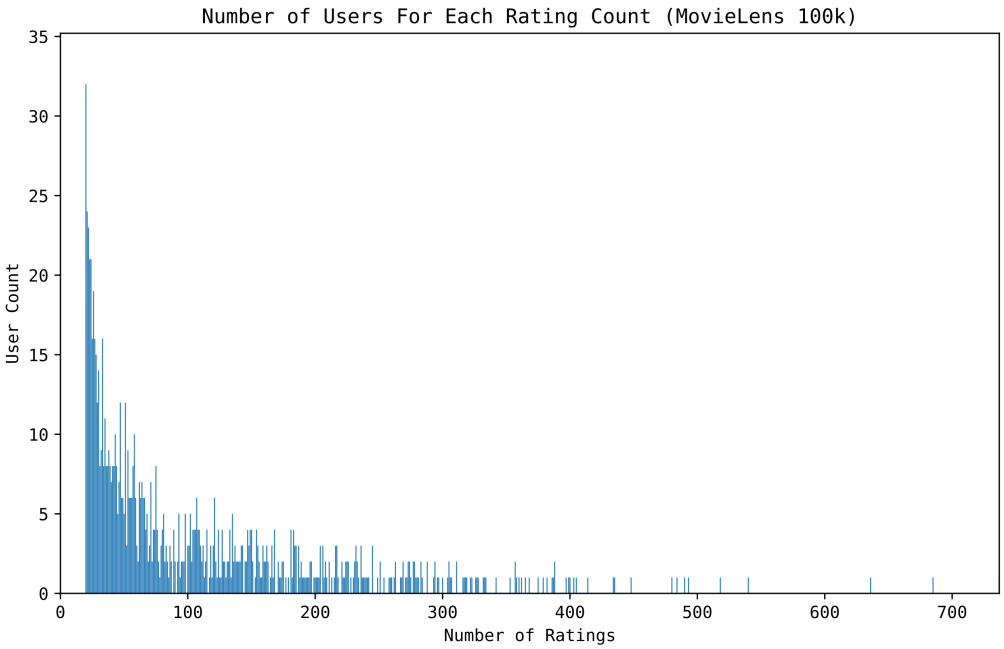


Figure 3.3: Rating distribution for users in the MovieLens 100k dataset

difficult to map the concept of an episode between an Atari-style environment, where the user-agent interactions typically last over a thousand steps and ends in an unambiguous terminal state, and a recommender environment, where the interactions are typically no longer than a couple dozen steps and there is no natural terminal state. This is one of the natural compromises of performing large-scale RL in an offline setting - clearly an online RS has no such restrictions and can simply keep suggesting new items according the user's interactions.

We overcome this weakness by taking the end of each episode to be the most recent rating given by each user in our offline dataset. We also customise the DataLoader class in PyTorch to ensure that data fed into the model during training never crosses the boundaries between different user episodes. Specifically, when we feed a context length's worth of input data to our transformer, we visit a random index in the training dataset. If the context length starting from this index surpasses a terminal index, we shift the indices left such that the end index is the terminal index. Ultimately when performing inference on this model, we are generating recommendations for a single user - this approach guarantees that when performing training, we are similarly only ever viewing the interaction history of one user at a time.

3.2.4 Modality-Specific Embeddings

Before feeding input data into the model, the data must be given a domain-specific embedding. For returns-to-go, we simply create a linear embedding layer between each return-to-go and a chosen number n. This number is a hyperparameter, designed to be easily configured between experiments, representing the desired dimensionality of embeddings being fed into our model.

Our model is designed to predict actions (or items, as according to the mapping in 2.1). Thus for actions, we borrow the concept of vocabulary size from NLP models designed to predict words. When setting up language models, we create a vocabulary containing all the words used in the textual source data. In our case, our vocabulary consists of all the actions possible to take within our environment - in other words, all of the items featured in our source rating dataset. We thus create a linear embedding layer between the size of the vocabulary and n.

As described in section 2.4.1, this mapping becomes particularly challenging for state representations. At the simplest level, we could feed the user IDs into the model directly to serve as state. The problem with this type of approach is that there is nothing particularly semantic about the user's id - we desire a state that serves as a representation of a user's preference. At this stage, some RS introduces demographic information designed to serve as a proxy for preference. However, as we seek to avoid the usage of demographic information in this system, we create a one-hot encoding for the user that categorises them according to the group they were drawn from in the creation of the synthetic dataset. One-hot encoding is suitable for categorical data across a small number of classes. We assign a binary value of either 1 or 0 to each categorical value. In our case, there is a single 1 value in each vector, denoting the group of membership. We then create a linear layer mapping the 4-dimensional input to our n-dimensional output.

Later, when it comes to performing inference on our model, where we seek to produce a sequence of new recommendations for a given user, it therefore suffices to specify the initial state as the one-hot encoding of the user's preference group.

3.2.5 Sequence Representation and Attention

We interleave the return-to-go, state, and action embeddings to form the token embeddings sequence, where k is the context length of our model:

$$\hat{r}_1, s_1, a_1, \hat{r}_2, s_2, a_2, \ldots, \hat{r}_k, s_k, a_k$$

Positional embeddings are added to these per timestep in order to preserve sequential

ordering between tokens. This is a constraint imposed by the transformer's attention mechanism, which does not process input tokens sequentially as other models such as RNNs would, but rather in parallel. For each 3-tuple of input (return-to-go, state, action) we predict a corresponding action. In predicting this action, the attention mechanism weighs each token against all previous tokens. Throughout the training process, the previous tokens most influential in predicting the next are given strong attention weights. We visualise the average attention weights across all heads using attention heatmaps as shown in Figures 3.4, 3.5, and 3.6.

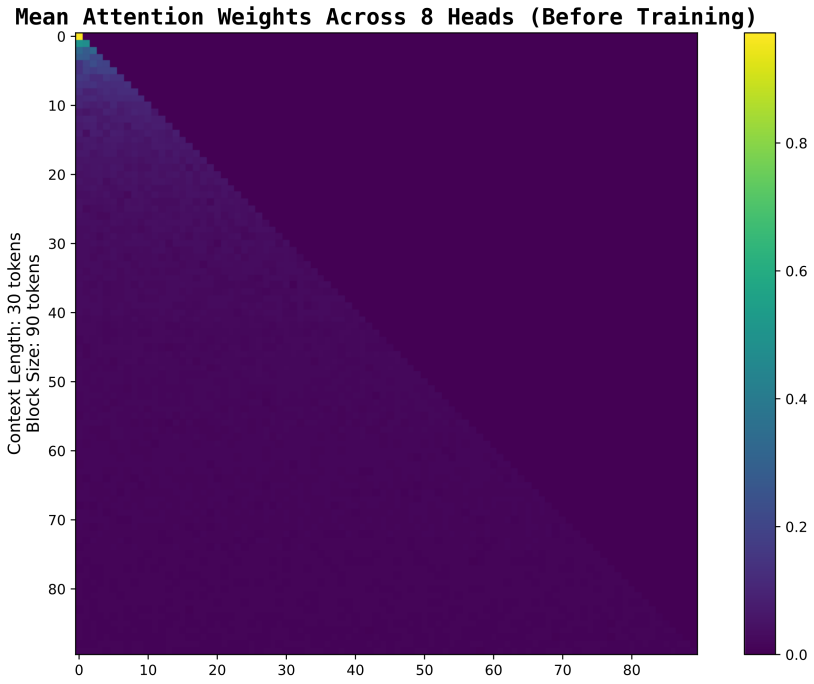


Figure 3.4: Attention weights across interleaved return-to-go, state, action input tokens, for a context length of 30, before training begins. Visualisation is from the training of the model depicted in **??**)

The rows in these heatmaps show how the attention weights develop from considering only one previous token to considering an entire block size's worth of previous tokens (note that the block size is three times any given context length, representing the length of the interleaved sequence). Before training, the attention weights are uniform, with random initialisation. During training, we see certain previous tokens gain stronger attention weight, indicating their influence on new predictions. At the end of training, we see a diagonal line across the plot typical of auto-regressive models showing the current token's attention to itself. Actions are the most strongly weighted tokens as these are what the

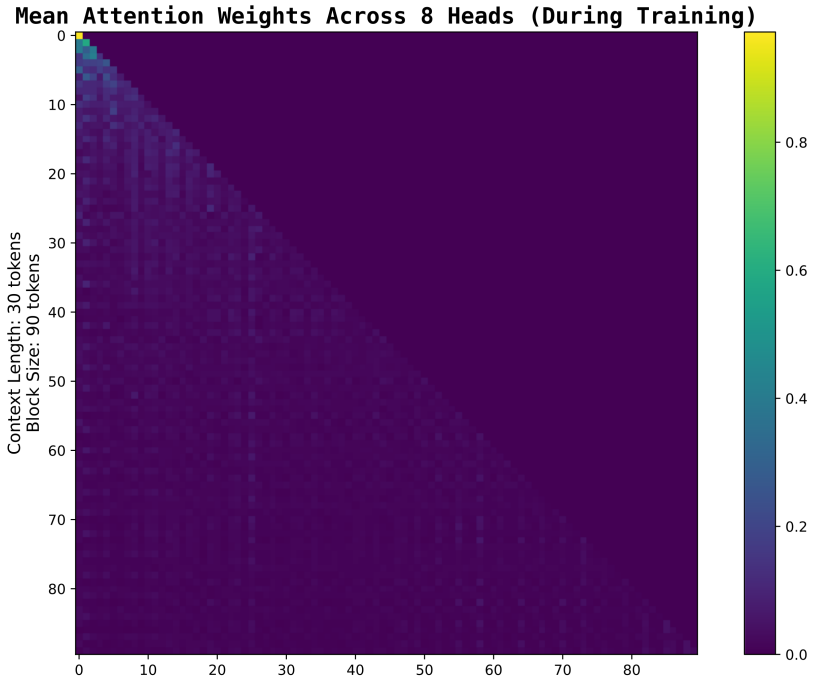


Figure 3.5: Same attention weights as shown in Figure 3.4, this time taken from an early snapshot *during* training. Previous tokens most influential in generating the current begin to be weighted stronger.

model is being trained to predict. Our model also seems to give a relatively stronger weighting to returns-to-go when predicting tokens.

3.2.6 Loss Function

We apply a suitable loss function to compare predicted and ground-truth actions. The choice of function depends greatly on the task at hand. For continuous data, mean-squared error (MSE) is often chosen. In these tasks, the predicted actions will be a real-valued vector and can be compared to the ground truth actions by considering the average squared difference between the two, thus penalising errors of magnitude to a far stronger degree than errors of direction.

For discrete tasks such as recommendation, actions occupy a finite space at any given moment. The problem of predicting the correct action can thus be approached as a classification problem. Cross-entropy loss is preferred in this situation as it measures the similarity between the predicted probability distribution over this finite number of classes and the ground truth probability distributions for actions, where the actual action has a

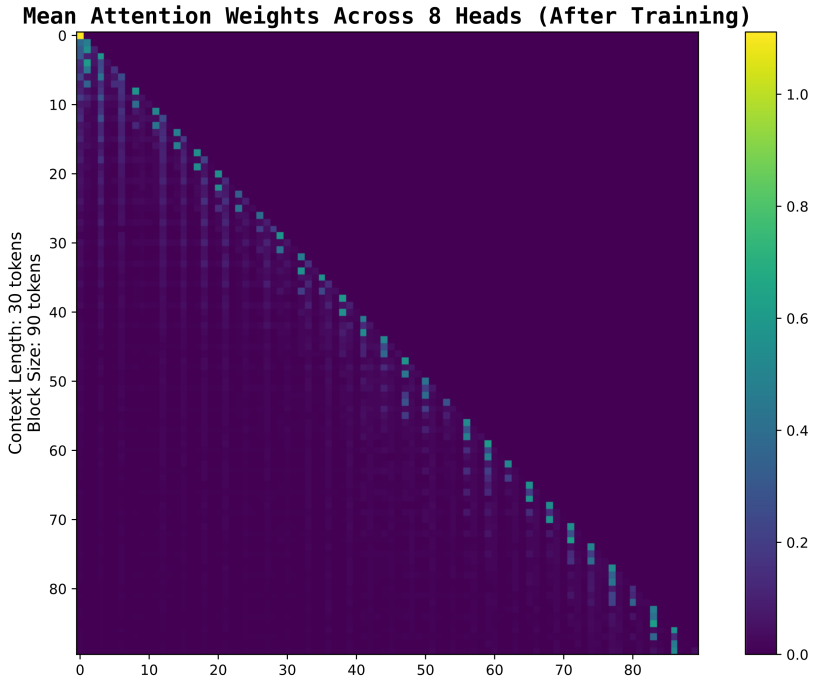


Figure 3.6: Same attention weights as 3.4 *after* training is complete. The diagonal pattern is typical of auto-regressive models, where recent tokens are given stronger weight.

probability mass of 1 and all others actions 0.

It is important to understand how the model is learning from the input data and producing new recommendations. The final layer of our model is a tensor with the shape (batch size, context length, vocabulary size). As our vocabulary size represents a finite set of actions to be predicted, we use cross entropy as our loss function, applying the formula between this final layer and the ground-truth values for each timestep.

3.2.7 Sampling Function for Explore-Exploit Balance and Slate Recommendation

When predicting the next action according to a conditioning sequence, we create a softmax probability distribution across the final layer of our model. As explored in section 2.3, softmax allows for custom scaling of the 'randomness' of recommendations and gives our system the ability to balance between low-temperature, 'exploitative' recommendations, and high-temperature, 'explorative' recommendations. In addition, unlike a solely RL-based RS, we can return any top A most probable recommendations from this distribution.

Pseudo-code of this sampling process is given in Figure 3.7.

```
1. If the number of states given exceeds the context length, trim to only consider the
     last context length worth of states.
2. Do the same for actions, checking first that actions is not empty (as is the case
    at the very beginning of a sequence where we only provide an initial state and
    desired returns).
3. Do the same for returns_to_go.
4. Do a forward pass of these states, actions, returns_to_go, and timesteps through
    the model in evaluation mode and get the resulting logits from the final layer.
5. Scale these logits by the chosen temperature, and apply softmax to then convert
    them to probabilities.
6. Return either the most probable action or a slate of A most probable actions from
    the softmax distribution.
7. Predict the strongest action that has not yet been taken.
```

Figure 3.7: Pseudo-code for Sampling Actions (i.e. Making Recommendations)

This solves the problem of slate recommendation, where we present many recommendations to the user at once. Slate recommendation drives many of the desirable user interfaces of RS, such as letting the user visit the landing page of a platform and explore a tray of dynamically generated items (consider common 'for you' user interfaces). Slate recommendation in traditional RL models is not easy to achieve, typically involving a brute force search across all possible slates for the slate with the highest expected value, growing exponentially in complexity with how many recommendations we desire per slate. However, by following the procedure described above, our model is able to handle both slate and single-item recommendation with ease.

Chapter 4

Evaluation

As foreseen in section 2.4.2, designing a suitable evaluation method for our system is difficult. Without the ability to deploy to online users, and considering the dearth of suitable simulation environments for RS, we turn to offline evaluation. We design a form of offline evaluation to address the common deficiency of training a RL-style RS based on offline data - namely, not having an elegant way to receive user feedback for new items recommended.

4.1 Offline Evaluation

Offline evaluation is achieved by returning to the grouped user data generated by the MCTS as described in section 3.2.6. In parallel to creating various synthetic training datasets, we create a synthetic evaluation dataset where each user has rated every single item - thus creating a *complete matrix* of user ratings. For example, from the synthetic Goodreads data, we generate a certain number of users for each of the four groups, and then for each user, we sample a rating for each of the 273 items in the dataset. Figure 4.1 shows the intuitive difference between our training and evaluation datasets.

At the end of every epoch of training, we measure the current performance of our model according to the algorithm described in Figure 4.2. We kick off this evaluation process with nothing but a given state, in our case a one-hot encoding of user group, as well a declaration of ideal returns, and pass this first step through the model. The top action predicted is referenced against the evaluation dataset to find the corresponding rating. This rating is then fed back into the model when predicting for the next timestep, allowing for a growing sequential awareness.

If performing slate recommendation, we return a given number of top actions at each timestep, rather than the single highest-ranked one. In order to track the sequential

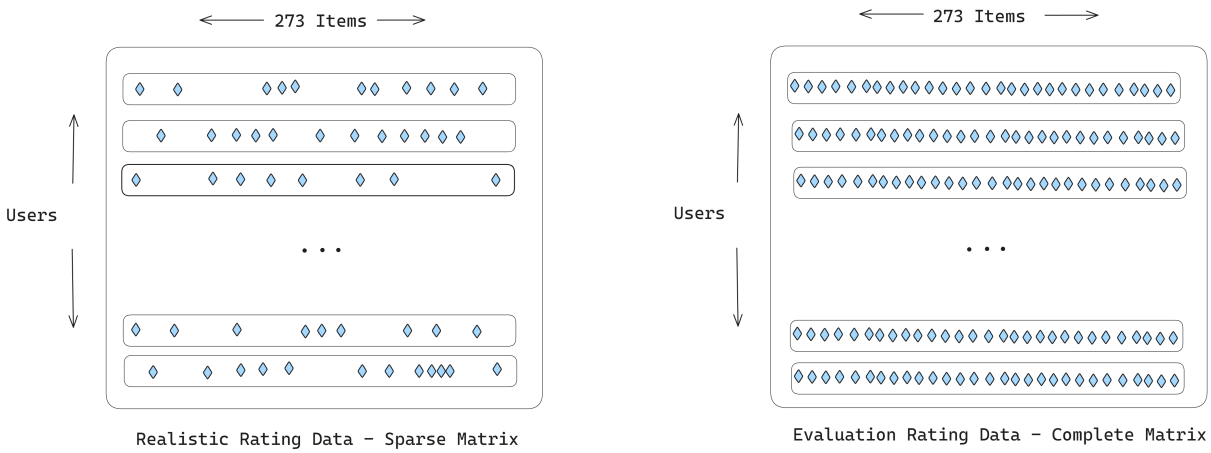


Figure 4.1: Visual intuition of the difference between training and evaluation datasets.

awareness of our model, where we expect recommended items to be stronger and better ranked as the user rates more items, we report some NDCG values from within the slates produced at each timestep. NDCG, however, is less appropriate when used as a measure of the overall sequence of the top-rated item per timestep. It assumes the initial values in a recommendation sequence to be the strongest, which is not the case for our evaluation in which recommendations receive more feedback over time. *Cumulative gain* (CG) is measured instead, denoting the sum of user ratings across the timesteps of this overall sequence. The evaluation algorithm described also allows for calculating the mean CG across many randomly-selected users. It is assumed that the sum of ratings across timesteps will be higher in a stronger recommendation model. We track CG during the training process, plotting its evolution across epochs.

4.2 Experiments and Results

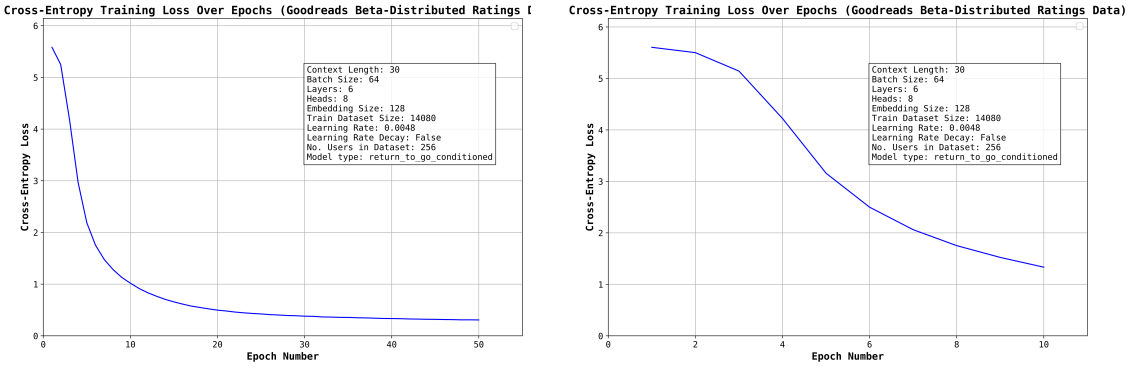
The experimentation phase of this project involved a careful blend of hyperparameter tuning and dataset selection. Hyperparameters controlling how the model ingests data include context length - or how many timesteps of interaction history we feed into the model at a time - and batch size - how many examples we process in parallel, with a high batch size speeding up training and a low batch size using less memory. We can modify the complexity of our GPT model by adding transformer layers and attention heads, however these were found to have little impact on the fitting of our model. With the datasets and evaluation method available, we track the ability of our Decision Transformer-based model to serve as a convincing sequential RS.

```
function get_returns:
    Determine ideal_return based on model_type
    Switch model to evaluation mode
    Initialise a global list to store all the user rating sequences
    Select random users for evaluation

    For each user:
        Initialise reward and return-to-go tensors with ideal_return value
        Initialise a list to store current user rating sequence
        Initialise the actions tensor as empty
        Fetch evaluation data for current user

        For each recommendation in the number of recommendations desired per sequence:
            Initialise state tensor
            If this is the first recommendation:
                Initialise tensors for first actions, first rewards or first returns-
                    to-go, and timesteps, where first_actions will be empty
                Sample top actions from model
            Select top action not previously added to actions tensor
            Concatenate this action to action tensor
            Find index of action in evaluation actions
            Fetch user rating corresponding to this index
            Concatenate to reward/return-to-go tensors appropriately
            Add rating to current user rating sequence list
            Concatenate next state to state list
            Concatenate next timestep to timestep list
            Sample next top actions from model
            Append current user rating sequence list to the global user rating
                sequence list

    Use global user rating sequence list to calculate average user rating sequence
        list
    Switch model back to training mode
    Return the sum of all the ratings in the average user rating sequence list
```

Figure 4.2: Pseudo-code for the Evaluation Process.

4.3 Early Training on Small Datasets

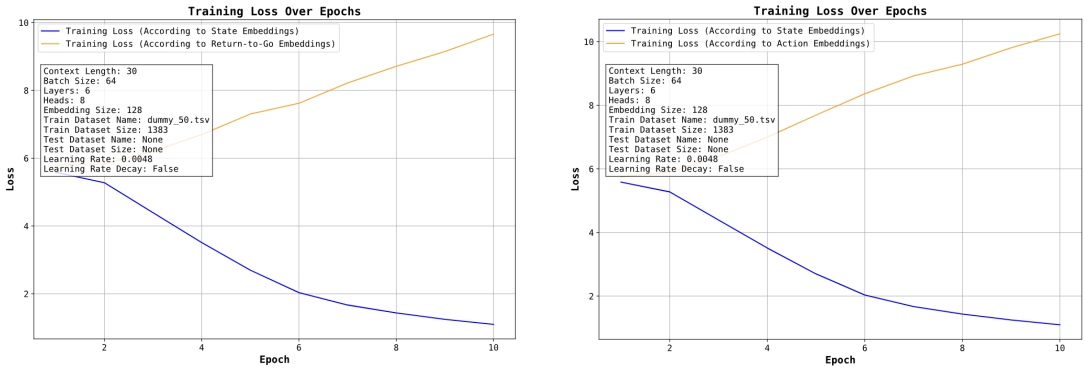
Over most of the earlier training runs, small synthetic Goodreads datasets were created. Figure 4.3 gives an early indication of the relationship between the loss function and the number of epochs covered during training. For a dataset of this size, $\sim 14,000$ ratings, 50 epochs of training are enough to converge to an optimal solution. The same experiment is repeated for less epochs, showing that most convergence happens in the first 10 epochs, and that around this point is where the loss begins to plateau for a dataset of this size. These results indicate the relative ease of fitting a model to a small dataset. In practice, however, it is desirable to capture patterns from a larger and more diverse dataset, in order to improve the ability of the model to generalise to unseen data.



(a) 50 Epochs (b) 10 Epochs

Figure 4.3: Hyperparameters and Cross-Entropy (CE) loss over an early training run of 50 and 10 epochs on a dataset containing $\sim 14,000$ ratings.

Decision Transformer, like traditional RL, must satisfy the Markov property where actions are predicted according to their corresponding state. In the final prediction layer of our model, we focus only on the actions corresponding to state embeddings, and calculate loss accordingly. Figure 4.4 shows how the loss otherwise fails to converge when focusing on actions corresponding to return-to-go embeddings and action embeddings, respectively, providing a useful sanity check on the satisfaction of the Markov property for our RS model.



(a) Return-to-go conditioned (b) Action conditioned

Figure 4.4: Comparison of the convergence of cross-entropy loss for a correctly state-conditioned model and incorrectly return-to-go and action-conditioned models.

The most common hyperparameter adjustments made throughout the experimental phase of this project were to learning rate, and adjusting the learning rate decay. Figure

4.5 visualises the process of learning rate decay on a relatively small dataset of 7,000 users, which we can use to help converge on an optimal solution. Here, we use linear warm-up followed by a cosine decay, after which loss drops more significantly.

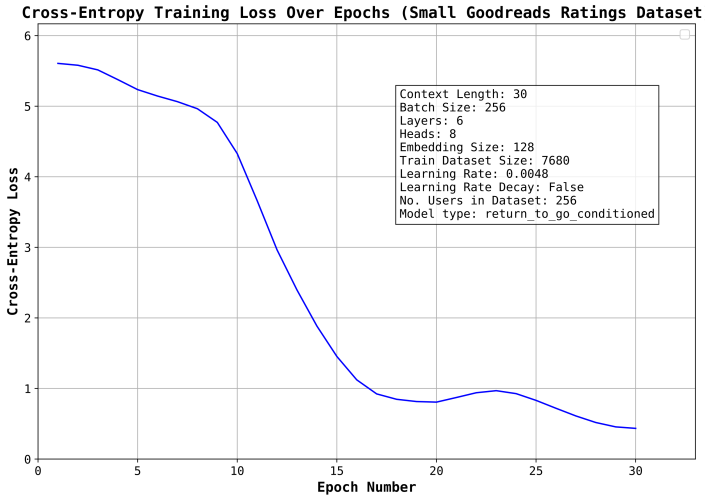


Figure 4.5: Loss on a dataset containing $\sim 7,000$ ratings, with learning rate decay over time visible.

The results from evaluating this model are shown in Figure 4.6. At the end of every epoch of training, after splitting the dataset into batches and updating the loss based on a full pass through these batches, we make a call to the evaluation loop as shown in Figure 4.2. The mean Cumulative Gain (CG), or the average sum of ratings given by users across the sequences of items the model recommends to them, is plotted over epochs. This is compared to the performance achieved by recommending random items.

Because all of the parameters in our model are randomly initialised, it is reasonable to expect the model to start off by making arbitrary recommendations and receiving ratings that reflect the averages of the dataset, resulting in a low mean CG. We expect, however, that as loss decreases and the model makes more full passes through the training data, the sum of ratings provided by the users will climb above the baseline of randomness. This pattern holds in Figure 4.6, however in other training runs with identical conditions, such as that shown in Figure 4.7, the mean CG jumps unusually high at the beginning of training, before stabilising to a level above the random baseline. One source of inconsistency at this stage of experimentation was the representation of state as simple user ids, rather than as a one-hot encoding of the user's group; later models trained with one-hot encoding show less inconsistency between experiments. Another source is the relative brevity of the datasets, especially when compared to later training runs.

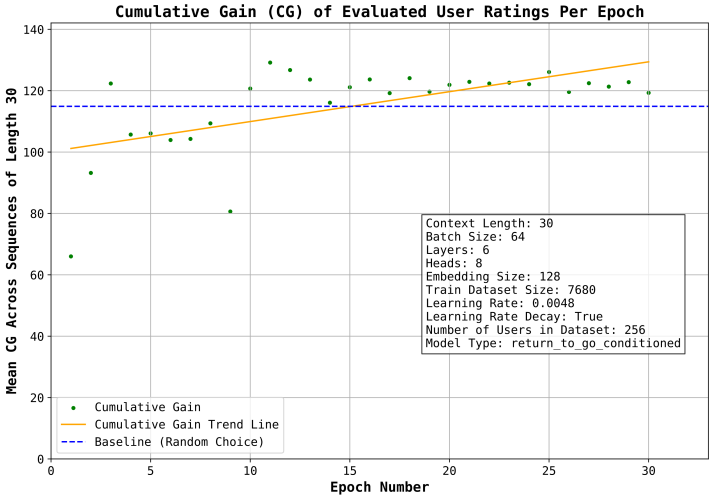


Figure 4.6: Performance of the model shown in 4.5 - mean cumulative gain is calculated across the recommendation sequences given to many users at each evaluation.

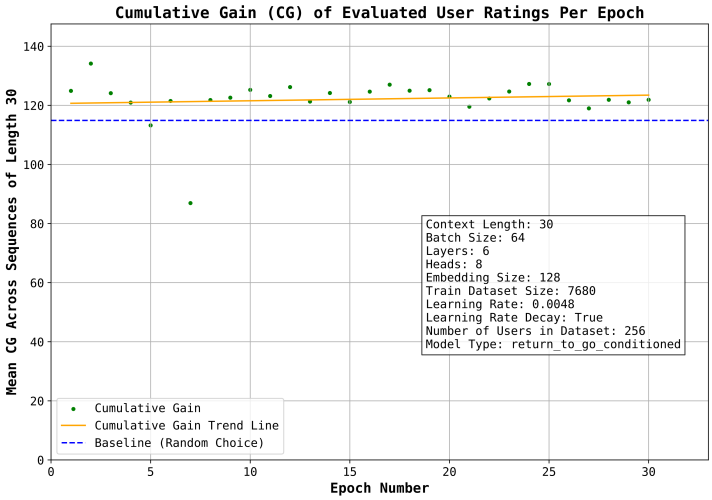
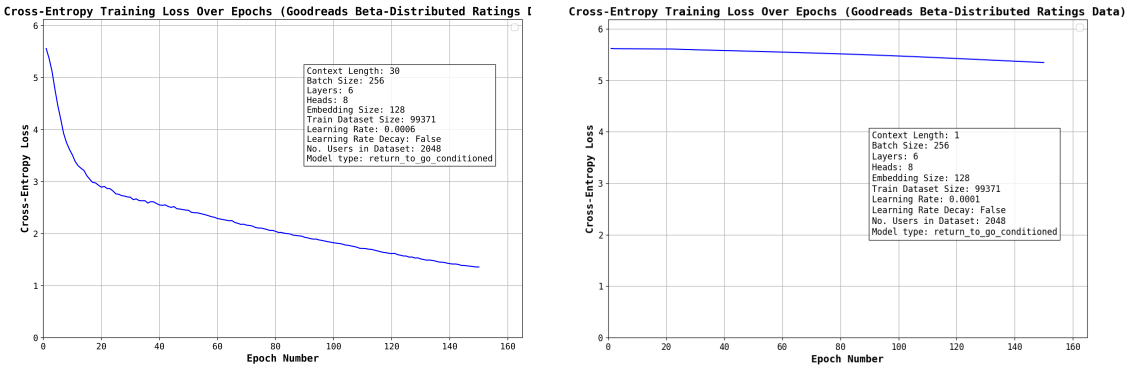


Figure 4.7: Performance of a newly-trained model identical to that in 4.6, with inconsistent result.

4.4 Training on Larger Datasets

Moving to a larger dataset, we find the most suitable hyperparameters largely through experimentation. Figure 4.8(a) shows the development of loss when training on a synthetic Goodreads dataset of $\sim 100,000$ ratings, highlighting the importance of running the model

for more epochs, as well as finding the largest batch size possible for the available system memory. It also demonstrates how, in this case, a smaller learning rate resulted in better convergence. We perform ablation experiments on context length to determine its role in training, and find that although slightly larger or smaller context lengths did not alter the loss curve significantly, reducing context length to the extreme of 1 caused a complete breakdown of training, as seen in the comparison shown in Figure 4.8(b).



(a) Context Length of 30

(b) Context Length of 1

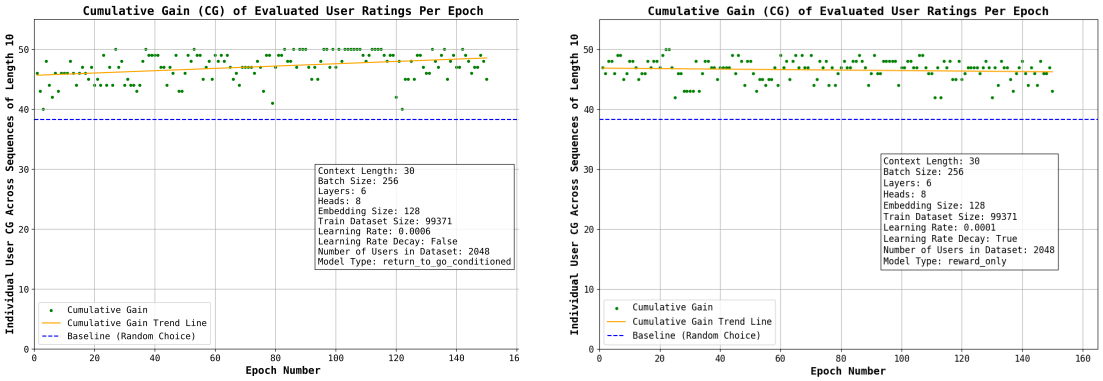
Figure 4.8: Hyperparameters and Cross-Entropy (CE) loss over a training run of 160 epochs, with context length of 30 and 1, respectively, on a dataset containing $\sim 100,000$ ratings.

4.4.1 Modelling Returns-to-Go Versus Direct Reward

Our model is modified to handle not only the return-to-go-conditioned situation, where the model sees not the direct rating but the cumulative remaining ratings over the current sequence of interactions, but also the the reward-only situation, where user ratings for an item at each timestep are used directly as rewards. The motivation behind this is the observation that Decision Transformer is designed originally for sparse reward environments, where feedback is given intermittently between many actions. In RS, we receive feedback for every action, and thus may not have as strong a need to keep track of cumulative remaining reward at each timestep.

This theory is tested in experiments such as those shown in Figure 4.9. These were conducted on models trained with the same methodology as the model shown in Figure 4.8(a). For both of these experiments, instead of performing the evaluation process over a given number of randomly-selected users, and returning the average of the ratings these users gave the items, we select a single random user. Tracking the journey of just one user as they interact with our RS is more computationally straightforward, making it

easier to train and rapidly iterate on these larger datasets, and also offers a more intuitive simulation of the end user experience.



(a) Return-to-Go Conditioned

(b) Reward-Only

Figure 4.9: Performance of the return-to-go conditioned versus the reward-only model for a dataset of size $\sim 100,000$.

Performance is strong across both the return-to-go-conditioned and reward-only models. As these experiments reflect the performance for a single random user's taste rather than the mean of many converging tastes, we see a strong boost above the random baseline. The return-to-go-conditioned model sees an increase in performance over epochs, whereas the reward-only model achieves a similar mean CG throughout, suggesting that the model is more capable of improving over epochs of training when it sees cumulative remaining ratings (returns-to-go) rather than direct ratings.

4.4.2 Finding a Good Sequence Length

Another explanation of the improved performance seen in these experiments is in the selection of the number of recommendations to be given to the user during each evaluation. Although some user experiences in the field of RS involve suggesting an item even if it has already been interacted with, i.e. streaming platforms with a 'watch it again' feature, for most sequential RS we do not want to give repeat recommendations. Our model is thus designed to generate actions without replacement; once an item has been recommended and appended to the growing sequence of user ratings, it is not eligible to be recommended again. This shrinks the number of high-relevance items eligible for recommendation over time, an effect that is even more pronounced in datasets with smaller action spaces, such as our Goodreads data with 273 items. We typically notice a worsening in performance when moving from a user rating sequence of length 10, as shown previously, to a sequence of length 50 as shown in Figure 4.10, in otherwise comparable conditions. We find the

ideal number of recommendations to be between 10 and 30 for the current data, although a production RS trained on data with a larger action space would be able to extend this to a far greater sequence length.

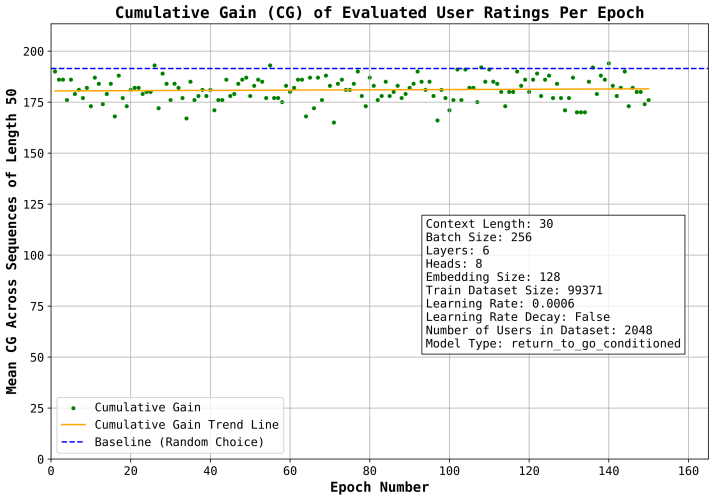


Figure 4.10: Performance of the return-to-go conditioned model on a longer evaluation sequence for a dataset of size $\sim 100,000$.

4.4.3 Setting Ideal Return over an Episode of Recommendations

Some of our experiments were conducted with a different value for ideal returns. For the experiment shown in Figure 4.11, the ideal return was set as the highest cumulative ratings given by any one user in the training dataset, which is 571 for the 2048 user, $\sim 100,000$ rating dataset.

Starting the recommendation sequence with a desired return that is unrealistically high appears to confuse the model's ability to recommend good items, with performance close to the random baseline. This is a problem identified in the original Decision Transformer, where some domain knowledge was needed to set a sensible expert return for various Atari game episodes. In all other experiments, we set the ideal return as the maximum possible rating of 5 multiplied by the number of recommendations we are giving the user in a row, `num_recs`. These other experiments show that taking the sequence as an episode reaching a terminal state after `num_recs` steps, and setting the ideal return accordingly, results in better performance. The requirement to declare an ideal expert-level return for non-traditional RL data remains a key challenge in applying Decision Transformer to less traditional RL settings.

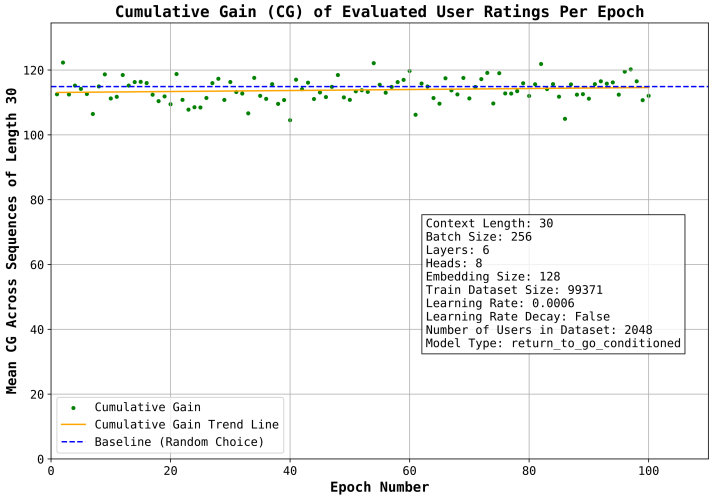


Figure 4.11: Performance of a model with ideal return determined as the highest cumulative rating achieved by any user in the training dataset.

4.5 Training on Further Alternate Datasets

In moving from a synthetic Goodreads dataset with $\sim 100,000$ ratings to a dataset with $\sim 200,000$ ratings, we re-observe the strong inverse correlation between cross-entropy loss decrease and performance increase that was present on the smaller models and datasets, shown in Figures 4.12 and 4.13. The model learns to predict better and better items as the loss converges, until settling to predictions around the random baseline as the loss plateaus and the model appears to stop learning from the data.

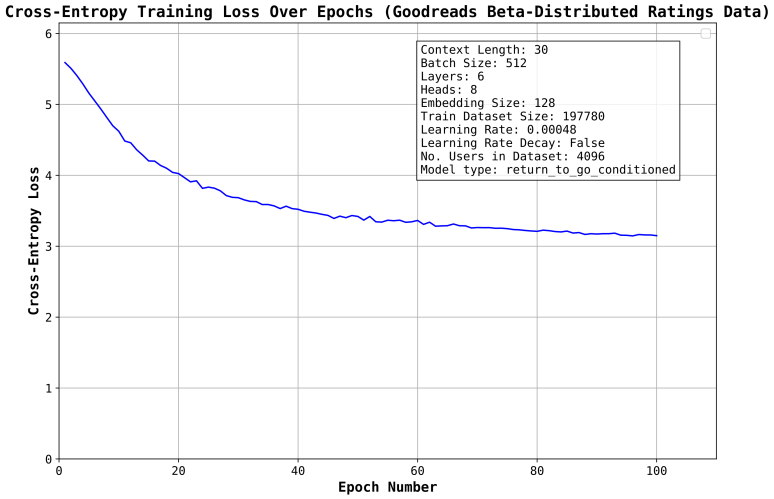


Figure 4.12: Hyperparameters and Cross-Entropy (CE) loss for a training run on a dataset containing $\sim 200,000$ ratings.

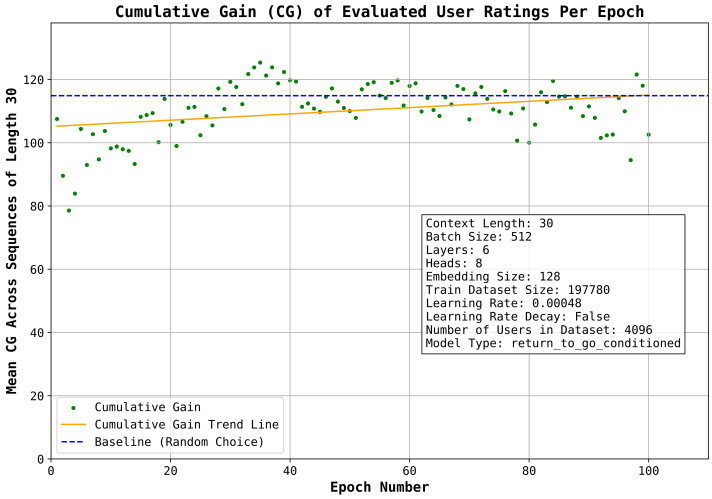


Figure 4.13: Performance of a model on a $\sim 200,000$-rating dataset, with improvement correlated to falling CE loss.

Changes in learning rate, and increases in epochs, embedding sizes, layers, and heads, do not resolve this issue of loss converging at around the value of 3. This could be in part due to stretching the synthetic data available to its limit - datasets of this size across such a small action space are highly uncommon in reality.

4.5.1 Training on MovieLens Data

Returning to the dataset that was the original inspiration for the schema behind our synthetic data, we train our model on the MovieLens dataset of 100,000 film ratings. Changing the UNIX timestamps to timesteps, and the movie IDs to item IDs, is all that is required before passing the dataset to our model for training, shown in Figure 4.14.

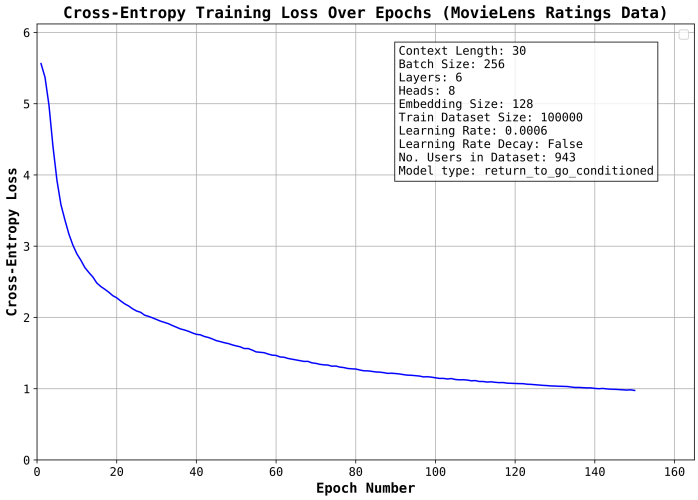


Figure 4.14: Hyperparameters and Cross-Entropy (CE) loss for a training run on the MovieLens dataset containing 100,000 ratings.

The difficulty in evaluating datasets that were not created synthetically becomes clear after training on MovieLens data - even though our model generates a number of top actions at each timestep, we have no suitable complete matrix of ratings to reference this item against. This remains a problem even after training on an 80% split of the original dataset and holding out 20% purely for evaluation - 20% of the ratings in the MovieLens data, with the rating count for each user highly positive skewed towards the base of 20 items, still does not come close to offering offline evaluation for the 1,682 items the model is capable of recommending at any given time.

4.6 Inference

PyTorch allows for easy saving and loading of trained models. Loading a saved checkpoint from the final epoch of training of the return-to-go conditioned model trained on $\sim 100,000$ ratings, we can perform inference on our model. This process is the same as the evaluation shown via pseudo-code in Figure 4.2, except with NDCG reported. At each timestep in the user rating sequence, we consider the top A actions not yet seen as a

slate recommendation (i.e. the top 10). It makes sense to use NDCG at each timestep to assess the quality of these recommendations as within the top actions, there should be a natural ranking placing the best items higher. Measuring NDCG upon inference for this model typically gives a result of the form shown in Figure 4.15.

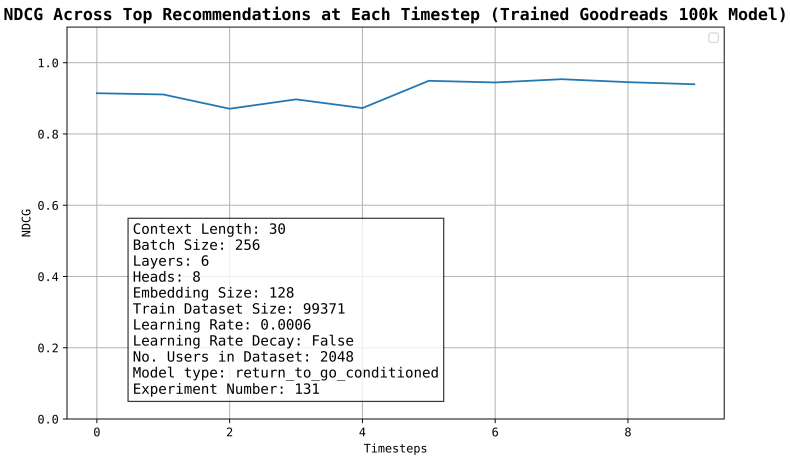


Figure 4.15: NDCG across the top A recommendations at each timestep, when performing inference on trained Goodreads model.

Intuitively, this means that the best items appear early on in the slate recommendations at each timestep of inference, and that this ranking improves moderately the longer the user rating sequence gets. Having high-NDCG slate recommendations as shown here drives many of the pleasing user experiences of RS, as it means a user is more likely to find items they are interested in at the top of the given list. Thus, less of the effort of discovery, such as scrolling and evaluating manually, is shifted to the user. Additionally, having a system where NDCG improves over a longer sequence indicates a model better able to increase user engagement with the system, as recommendations evolve and improve the more they interact with it.

Chapter 5

Conclusions & Future Work

This work successfully implements a sequence-aware RS based on Decision Transformer, despite the many difficulties encountered in converting an undoubtedly RL-oriented model to a novel task of RS. This research direction was navigated with extremely limited precedent or proof-of-concept from the literature, with the topic of Decision Transformer-based RS discussed in only a small number of papers released from 2023 onwards [31] [14] [15].

Our work involved the generation of suitable training data, closely following the statistical and schematic patterns of existing datasets, and the modification of the Decision Transformer architecture to allow for training on this data. The process of sampling from the generated probability distribution over all possible items, both during training and inference, was modified to better reflect the goals of RS. This sampling, via its temperature hyperparameter, introduced a level of deliberate randomness to address the common lack of an explore-exploit balance in RL-based RS.

The sampling process also achieved slate recommendation with high NDCG, showing the model's capability in generating recommendations of a form amenable to many of the desirable user experiences of an RS, giving the user the freedom to select among a ranked set of candidate items. Our work involved the creation of an evaluation process from scratch, designed to generate good recommendations for users in the system based on their grouping as well as the ideal sum of ratings desired over the course of their interaction. Work also included the carrying out of experiments on different dataset sizes, model hyperparameters, reward representations, token embeddings, and evaluation methods, in order to drive down loss and gain as stable and realistic a view of performance as possible.

Nevertheless, certain limitations in applying Decision Transformer to RS remain. Although addressed as best as possible within the scope of this work, these represent clear directions for future work in what is a highly active area of research.

5.1 Future Work

Some thoughts are given in this section on the shortcomings of the present research, as well as suggestions for future directions that would be of the greatest utility to the active research teams researching this problem worldwide.

5.1.1 Addressing Bias in Underlying Data

The MCTS-generated Goodreads grouping data was influential to our work in several ways. It allowed for the referencing of recommended items against a theoretical complete matrix of user ratings for each item in the dataset, in order to receive the rating the user is likely to have given the item. Without this feedback loop, and in the absence of a suitable simulation environment, it would have been more difficult to measure the performance of our model.

However, training on this synthetic data suffers from a similar flaw as was observed in the original Decision Transformer implementation, as discussed in Section 3.1, where the large offline datasets used as input are generated using the kind of RL agent the model purports to replace. In our case, during the design and implementation of our model, we sought to avoid relying on demographic information such as age, gender, occupation, etc. in describing the state of the user. While this information is often convenient to the developer of the RS, particularly in determining which items to recommend to a brand new user, the reduction of a person's complex interests and preferences to inferences about the existing social groups to which they belong can serve to impose algorithmic bias [32]. Unfortunately, removing any demographic identification from our schema does not address the fact that this very same information was used upstream in order for the MCTS model to successfully cluster users according to their preference groups. These existing forms of bias can thus not be ruled out entirely from our RS.

Another problem inherent to Decision Transformer is that it is unclear the extent to which expert-level datasets need to be used during the training of the model. More recent studies applying the architecture to RS unambiguously state their reliance on actor-critic, deep RL models such as Deep Deterministic Policy Gradient (DDPG) in creating expert demonstrations on which the transformer model can later be trained [31]. This risks introducing an expert bias into the RS, limiting the model's ability to generalise to more realistic input.

The introduction of these biases is the side-effect of some of the greater challenges facing offline-only RL, however it is clear that future work on this topic should focus on providing alternative methods of data collection better able to mitigate them.

5.1.2 Deployment to Online Environments

Another consequence of using the Goodreads synthetic datasets to train our models is the necessary limiting of our action space, or the number of items in the system available to be recommended to users. Although this is suitable for proof-of-concept research, most action spaces in production systems are 'generated dynamically and stochastically' [20] constantly changing to reflect added and deleted items.

Our models, such as that described in Section 4.6, can be loaded from a PyTorch saved checkpoint and integrated in the back-end of online platforms in evaluation mode in order to perform inference. However, as it stands, these models would have to be fully retrained in order to reflect changes in the action space, making them computationally infeasible. Any future work involving the deployment of our model to an online production environment would thus need to introduce a strategy for dealing with a rapidly changing action space, such as using concepts from content filtering.

5.1.3 Alternate State Representation

In the attention visualisation of our trained model seen in Figure 3.6, it appears that even with our one-hot encoding of state, the state tokens were not heavily weighted when generating new predictions. Although one-hot encoding was a practical solution to state representation in this work, where we only had to deal with 4 possible preference groups, one-hot encoding becomes harder to scale as the number of fine-grained preference groups increases.

When performing inference on our model, we need to manually specify the user's group via this one-hot encoding. Future work could explore how to more elegantly handle this user cold start situation, for example by performing real-time clustering of the user based on their interactions. More generally, a comprehensive index of state representations used across the literature, along with the merits and downfalls of each, would also be a direction of future work of great utility to researchers in the space.

5.1.4 Better Evaluation Environments

The evaluation method designed for this work, as described in Figure 4.2, allows us not only to train but also to evaluate our model based on offline data alone. However, it would be useful to spend less time developing custom evaluation methods when researching RS. Ideally, we would have access to the kind of standardised, off-the-shelf simulation environment familiar to RL researchers, such as the OpenAI Gym or Dopamine. Such an environment might not only support the type of explicit ratings used in our work, but

also implicit ratings, giving feedback according to whatever metric we are most interested in. Future work building upon the early equivalents as described in Section 2.4.2 would be a great contribution to the field.

5.2 Concluding Remarks

The application of recent advances in transformers to RL has triggered a strong side-effect of progress in RS, some early demonstrations of which are shown in this report. With the need for models to process ever larger and more complex datasets, this trend looks unlikely to be reversed. It will be exciting to follow its acceleration over the coming months and years, particularly as it contributes to more thoughtful user experiences through the tumult of modern platforms.

Bibliography

[1] Scott Reed, Konrad Zolna, Emilio Parisotto, Sergio Gomez Colmenarejo, Alexander Novikov, Gabriel Barth-Maron, Mai Gimenez, Yury Sulsky, Jackie Kay, Jost Tobias Springenberg, Tom Eccles, Jake Bruce, Ali Razavi, Ashley Edwards, Nicolas Heess, Yutian Chen, Raia Hadsell, Oriol Vinyals, Mahyar Bordbar, and Nando de Freitas. A generalist agent, 2022.

[2] Xiaoyuan Su and Taghi M. Khoshgoftaar. A survey of collaborative filtering techniques. *Adv. in Artif. Intell.*, 2009, jan 2009.

[3] Dorota Glowacka. Bandit algorithms in recommender systems. In *Proceedings of the 13th ACM Conference on Recommender Systems*, RecSys '19, page 574–575, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Djallel Bouneffouf, Amel Bouzeghoub, and Alda Lopes Gançarski. A contextual-bandit algorithm for mobile context-aware recommender system. In Tingwen Huang, Zhigang Zeng, Chuandong Li, and Chi Sing Leung, editors, *Neural Information Processing*, pages 324–331, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[5] Elena Gangan. Survey of multi-armed bandit algorithms applied to recommendation systems. *International Journal of Open Information Technologies*, 2021.

[6] Xiaocong Chen, Lina Yao, Julian McAuley, Guanglin Zhou, and Xianzhi Wang. Deep reinforcement learning in recommender systems: A survey and new perspectives. *Knowledge-Based Systems*, 264:110335, 2023.

[7] Juergen Schmidhuber. Reinforcement learning upside down: Don't predict rewards – just map them to actions, 2020.

[8] Lili Chen, Kevin Lu, Aravind Rajeswaran, Kimin Lee, Aditya Grover, Michael Laskin, Pieter Abbeel, Aravind Srinivas, and Igor Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021. cite arxiv:2106.01345Comment: First two authors contributed equally. Last two authors advised equally.

[9] Shoujin Wang, Liang Hu, Yan Wang, Longbing Cao, Quan Z. Sheng, and Mehmet Orgun. Sequential recommender systems: Challenges, progress and prospects. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, IJCAI-2019. International Joint Conferences on Artificial Intelligence Organization, August 2019.

[10] Fei Sun, Jun Liu, Jian Wu, Changhua Pei, Xiao Lin, Wenwu Ou, and Peng Jiang. Bert4rec: Sequential recommendation with bidirectional encoder representations from transformer. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, CIKM '19, page 1441–1450, New York, NY, USA, 2019. Association for Computing Machinery.

[11] M. Mehdi Afsar, Trafford Crump, and Behrouz Far. Reinforcement learning based recommender systems: A survey, 2022.

[12] Lixin Zou, Long Xia, Zhuoye Ding, Jiaxing Song, Weidong Liu, and Dawei Yin. Reinforcement learning to optimize long-term user engagement in recommender systems, 2019.

[13] Feng Liu, Ruiming Tang, Xutao Li, Weinan Zhang, Yunming Ye, Haokun Chen, Huifeng Guo, Yuzhou Zhang, and Xiuqiang He. State representation modeling for deep reinforcement learning based recommendation. *Knowledge-Based Systems*, 205:106170, 2020.

[14] Kesen Zhao, Lixin Zou, Xiangyu Zhao, Maolin Wang, and Dawei Yin. User retention-oriented recommendation with decision transformer. In *Proceedings of the ACM Web Conference 2023*, WWW '23, page 1141–1149, New York, NY, USA, 2023. Association for Computing Machinery.

[15] Siyu Wang, Xiaocong Chen, and Lina Yao. Retentive decision transformer with adaptive masking for reinforcement learning based recommendation systems, 2024.

[16] Jing-Cheng Shi, Yang Yu, Qing Da, Shi-Yong Chen, and An-Xiang Zeng. Virtual-taobao: Virtualizing real-world online retail environment for reinforcement learning, 2018.

[17] Xiaocong Chen, Chaoran Huang, Lina Yao, Xianzhi Wang, Wei liu, and Wenjie Zhang. Knowledge-guided deep reinforcement learning for interactive recommendation. In *2020 International Joint Conference on Neural Networks (IJCNN)*. IEEE, July 2020.

[18] Kai Wang, Zhene Zou, Minghao Zhao, Qilin Deng, Yue Shang, Yile Liang, Runze Wu, Xudong Shen, Tangjie Lyu, and Changjie Fan. Rl4rs: A real-world dataset for reinforcement learning based recommender system, 2023.

[19] Bichen Shi, Makbule Gulcin Ozsoy, Neil Hurley, Barry Smyth, Elias Z. Tragos, James Geraci, and Aonghus Lawlor. Pyrecgym: a reinforcement learning gym for recommender systems. In *Proceedings of the 13th ACM Conference on Recommender Systems*, RecSys '19, page 491–495, New York, NY, USA, 2019. Association for Computing Machinery.

[20] Eugene Ie, Chih wei Hsu, Martin Mladenov, Vihan Jain, Sanmit Narvekar, Jing Wang, Rui Wu, and Craig Boutilier. Recsim: A configurable simulation platform for recommender systems, 2019.

[21] Teresa Scheidt and Joeran Beel. Time-dependent evaluation of recommender systems. In *Perspectives@RecSys*, 2021.

[22] Andrej Karpathy. minGPT: Minimalist generative pre-trained transformer. `https://github.com/karpathy/minGPT`, 2020.

[23] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[24] The pandas development team. pandas-dev/pandas: Pandas, February 2020.

[25] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[26] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[27] Anaconda software distribution, 2020.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[29] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 111–125, 2008.

[30] Dilina Chandika Rajapakse and Douglas Leith. Fast and accurate user cold-start learning using monte carlo tree search. In *Sixteenth ACM Conference on Recommender Systems (RecSys '22)*, page 10, Seattle, WA, USA, September 18–23 2022. ACM.

[31] Siyu Wang, Xiaocong Chen, Dietmar Jannach, and Lina Yao. Causal decision transformer for recommender systems via offline reinforcement learning, 2023.

[32] Silvia Milano, Mariarosaria Taddeo, and Luciano Floridi. Recommender systems and their ethical challenges. *AI & SOCIETY*, 35(4):957–967, 2020.