[Final Report]
RoboNotes: Reinforcement Learning for Music Composition

Eileen Li [chenyil@], Rutika Moharir [rmoharir@]

Abstract—**Reinforcement learning (RL) has proven to be effective for large variety of tasks, from optimizing ad placement to teaching AI to play Go. In this work, we apply techniques in RL to the problem of music composition. Creating inspiring music is a difficult challenge even for human composers. We limit our problem space by only considering two octaves of notes on a single track of composition, for a fixed length. We experimented with on-policy learning (PPO), off-policy learning (DQN), and model-based RL (CEM) and compare these methods with the performance of a random agent. We extend to multi-track composition (Modification #1) and use expert data with Behavior Cloning (BC) and GAIL (Modification #2). Our reward function is hand-crafted from a set of common music theory rules. Even in this limited setting, we show that the agent can indeed learn something meaningful and uploaded the finished compositions to Youtube for general viewing.**

I. Introduction

In this project, we will explore using reinforcement learning techniques for the purpose of music composition. The code can be reviewed at https://github.com/eileenforwhat/robonotes. This report describes our custom RoboNotes environment, the reward function, results showing the performance of PPO, DQN, CEM, vs. a random agent, and our two modifications: 1) multi-track composition and 2) BC and GAIL with expert data. We use MIDI, with is the widely used digital format for representing music data.

II. Terminology

Terminology used in this project:

- ‘note’ and ‘pitch’ are used interchangeably and refers to a single musical note.
- ‘beat’ is the length of one timestep. This is translated by the MIDI converter based on its ‘tempo’ setting to an actual BPM.
- ‘track‘ is the number of parallel notes we allow our agent to place at every beat. We explore multi-track composition in Modification #1.

III. Environment

The environment implementation can be reviewed at robonotes/blob/main/env.py.

A. Action Space

Our action space is defined as:

```
self.action_space = spaces.Discrete(38)
```

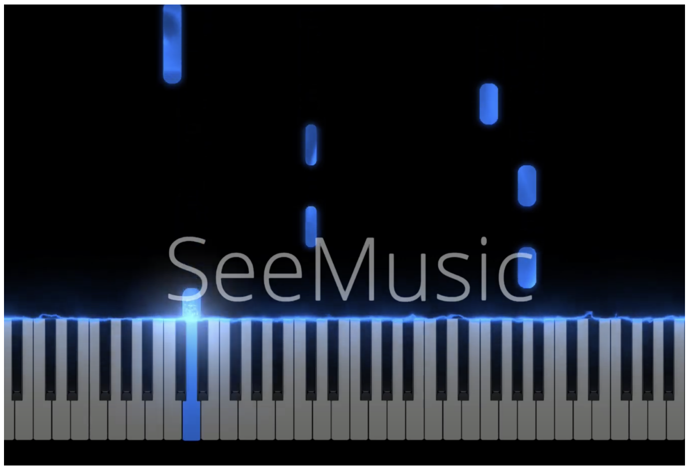


Fig. 1. Visualization of environment, by SeeMusic.

This definition represents the 36 notes + 2 special actions the agent can take at each timestep. The 36 notes span two musical octaves, mapping to MIDI pitches C3 to B5. The 2 special actions are note_off (0) and no_event (1). note_off releases the previous note and no_event produces no change to the observation (resulting in current notes held for longer duration). We can only have a single note at each timestep, so an action of any pitch (2 to 37) would also result in releasing the previous note. As an example, [3, 0, 4, 1] translates to [C3, Rest, C#3 for 2 beats].

B. Observation Space

We limit our problem space further by passing in a parameter *max_trajectory_len*, which determines the length (number of beats) of our composition. This is a parameter of our environment and determines the size of our observation space. Each observation consists of the sequence of actions (notes) that the agent has chosen so far, and we calculate a reward based on this sequence. The observation space is represented by:

```
self.observation_space =
    spaces.Box(low=0, high=37,
           shape=(max_trajectory_len,))
```

This gives us a fixed length vector initialized to (0 = note_off) at each environment reset.

C. Step

At each step, the agent takes the specified action by placing one more note in the composition. We keep track of the last index, and at each step, the state changes by adding the current action to the index that corresponds to this timestep.
As an example, for $(s_i, a) -> s_{i+1}$:

```
n = max_trajectory_len

t=0: (s0=[0, 0, 0, ... a_n], a1) -> s1
t=1: (s1=[a1, 0, 0, ... a_n], a2) -> s2
t=2: (s2=[a1, a2, 0, ... a_n], a3) -> s3
t=3: (s3=[a1, a2, a3, ... a_n], a3) -> s4
...
t=n: (s_n=[...], terminated)
```

We calculate the reward at each step by considering the sequence so far, and add it to the total rewards for that composition. Terminated is returned only if $t = max\_trajectory\_len$ has been met.

D. Reset

Rewards reset to zero. Observations reset to vector of zeros.

E. Render

Our environment supports $``render\_modes" : [``human"]$. While we may want to link this directly with a MIDI visualizer in the future, it currently calls our MIDI converter to save a MIDI file. This is called at the end of a trajectory.

F. MIDI Converter

robonotes/blob/main/midi/midi_writer.py contains the converter that translates our encoding of discrete actions to MIDI pitches and saves a MIDI file that can then be played on many existing tools. We use the SeeMusic App [6] for creating our videos.

IV. Reward Function

Crafting the reward function proved to be one of the major challenges of this project. Inspired by the work done by Jaques et al. [9], we based our reward function on music theory rules. Throughout the course of the semester, we've made efforts to revise and improve the reward function to encourage more sophisticated compositions. The implementation can be found at robonotes/blob/main/reward.py

We found that crafting a good reward function heavily guided the qualitative results of RoboNotes. Quantitative results were easier to obtain since RL algorithms are designed to optimized any reasonable reward function. However, this did not always translate to good qualitative results (subjective evaluation of how the music actually sounded), and only through cycles of trials and errors did we gain intuition for the effects of tweaking reward function parameters on the agent output. A more difficult failure mode was the problem of repeating and empty notes, despite penalties for both. It was only after adding an additional diversity reward that our agent broke out of this local minima.

Our total reward for the trajectory is a summation of the reward per step, a combination of five components:

$$R_{trajectory} = \sum_{t=0}^{T} R_{step_t}$$

$$R_{step_t} = R_k + R_o + R_r + R_e + R_d$$

- R_k: Reward for being in the same key
- R_o: Penalty for large jumps over an octave
- R_r: Penalty for repeating the same note
- R_e: Penalty for having empty (rest) beats
- R_d: Reward for entropy as a measure of note diversity

A. R_k: Key Reward

For simplicity, we set the first note of the sequence as the key of the composition. We give a reward of 5 if the current note is in the correct key. If we are at the terminal state, we give a large reward of 10 for ending the sequence on the tonic note (key note).

B. R_o: Octave Penalty

Return a penalty of -1 if current note is not within an octave of the previous note. We want to discourage large jumps.

C. R_r: Repeat Penalty

This penalty is important to discourage agent from selecting repeating notes. We scale the penalty based on the number of repeats in a row such that the longer the repeating sequence, the more severe the penalty: $R_r = -k$ where k is the number of consecutive repeats from the current note.

D. R_e: Empty Penalty

Similarly, we want to discourage the agent from selecting too many empty (rest) notes in a row. The penalty is scaled by length of empty notes, k: $R_e = -2 * k$, with a large penalty of -100 if the first note of sequence is empty.

E. R_d: Diversity Penalty

We want to encourage diversity in the music composition. We tried several methods such as auto correlation (with lags 1, 2, 3) and entropy, but found that the latter worked better. The entropy is greatest when the notes are all different and smallest when they are all the same. We give a reward of 5 if the calculated entropy is greater than 0.5 * the max possible entropy for the sequence.

V. Method

A. Modification #1: Multi-track Learning

The first modification increases the complexity of our problem space by expanding single-track composition to multi-track. This means that at each timestep, our agent is learning a vector of notes instead of a single note.

We modify our action space to support M notes per timestep:

```
self.action_space =
    spaces.Box(low=0, high=37, shape=(M,))
```

Similarly, we update our observation space:

```
self.observation_space =
    spaces.Box(low=0, high=37,
        shape=(max_trajectory_len, M))
```

We also make modifications to our rewards function to support multi-track outputs. For fair comparison, we keep the reward function as similar as possible, returning the average reward per track for each of the components. We run experiments using PPO on number of tracks = {2, 3}, but found that any higher numbers diverged in training, perhaps due to lack of robustness in the reward function for multi-track.

B. Modification #2: Imitation Learning with Expert Data

For our second modification, we experimented with imitation learning methods such as Behavior Cloning (BC) and Generative Adversial Imitaion Learning (GAIL) with expert data. To accomplish this, we downloaded 16K classical MIDI files from various composers from an online repository [1]. We pre-processed these files to align with the limits of our problem space (two octaves, single track, constant tempo). After pre-processing, we have left 10986 unique compositions. To generate our expert trajectories, we then sampled 20K sequences of length = $max\_trajectory\_len$ and used this dataset as the $expert\_rollouts$ for BC and GAIL. Note that each expert composition can yield many expert trajectories since the length of composition is often far greater than the $max\_trajectory\_len$. The pre-processed dataset can be loaded from robonotes/blob/main/data/midi_arrays.npy.

In BC, we simply use these expert trajectories to train a policy that, given input state outputs a distribution over actions, and is represented by a two-layer feed forward network. We evaluate this policy by taking the average reward per trajectory post-training.

In GAIL [8], we simultaneously train a discriminator (RewardNet) that aims to distinguish expert trajectories against trajectories from the learned policy and a generator (PPO) that aims to learn a policy to fool the discriminator. We alternate between training steps for the discriminator and generator until convergence, and then use the generator policy to sample trajectories for evaluation.

TABLE I
Comparing for each experiment: average reward per rollout, time to train until convergence, and number of steps until convergence. Regrettably, we did get steps[converge] info for BC and GAIL.

	avg_reward	time[train]	steps[converge]
random[b]	49.5	0min	NA
DQN	86.35	8min	1.5M
PPO	97.94	2min	200K
CEM	**100.9**	12s	200K
Multi-PPO	80.87	4min	600K
BC	58.5	5min	NA
GAIL	62	7min	NA

VI. Results

All the scripts used to run our experiments can be found at robonotes/blob/main/run_scripts/. We compare results from on-policy (PPO), off-policy (DQN) and model-based (CEM) RL. We also compare the results from our two modifications: multi-track (PPO) and imitation learning (BC and GAIL). We plot the performance of each method versus a random agent in Figure 2.

A. Quantitative

The plot shows that all our methods outperform the random agent, with CEM and PPO as the clear winners. The random agent fluctuates around 20-40 average reward per rollout without any improvement and with high variance. DQN converges after about 1.5 million timesteps at 86.35 average reward per rollout. PPO converges faster at about 200K timesteps at a higher reward of 97.94. While CEM converges at the same rate as PPO but with a slightly higher reward of 100.9. For our modification experiments, multi-track PPO (with two tracks) converges after about 600K timesteps at 80.87. BC and GAIL are trained with expert trajectories and their subsequent evaluation returns are plotted. GAIL has an average reward of 62 while BC is lower at 58.5. This is expected since BC is prone to generalization errors and does not perform well on unseen data while GAIL is much more robust.

From these results, it seems like CEM and PPO are the best choices among the 7 experiments. While PPO is learning directly from the sampled trajectories rather than trying to learn a Q-function to approximate rewards, CEM tries to improve the performance by training more on an elite set of states and actions. PPO has an advantage for our problem space in particular since sampling trajectories is very cheap.

1) DQN Training Setup: We train DQN with learning rate = 0.001 and batch size = 32. Learning starts after 100K timesteps. The train frequency = 4 (how often to update model) and the target update interval = 10K (how often to update target network). The number of gradient steps per rollout = 1.

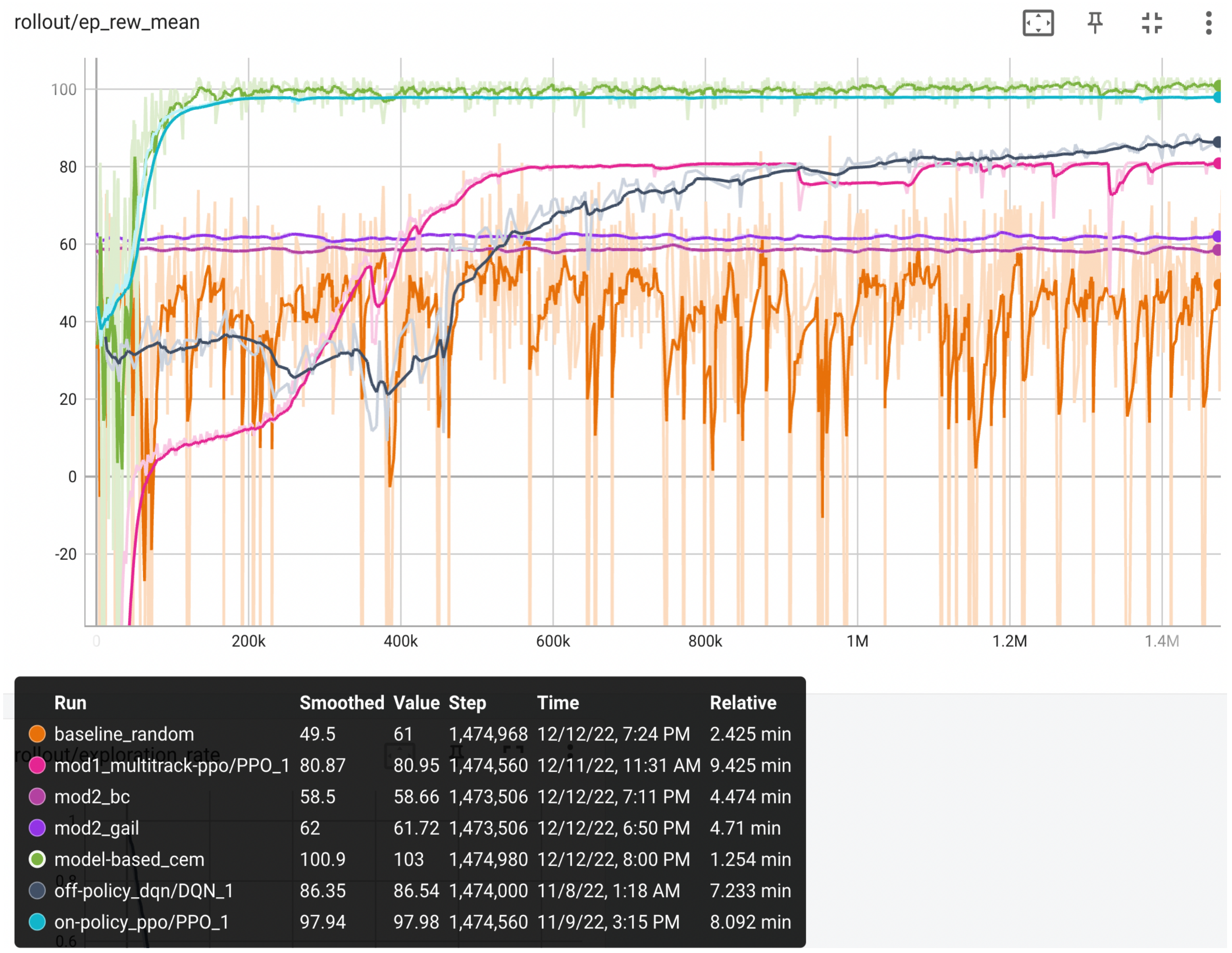
2) PPO Training Setup: We train PPO with learning rate = 0.003, batch size = 64, gamma = 0.99, gae_lambda = 0.95 and normalize_advantage = True. We did not perform an exhaustive hyperparameter search, but experimented with multiple settings to observe the best training curve.

3) CEM Training Setup: We train CEM with learning rate = 0.01, discount factor = 0.99, rollouts = 1000. We did hyperparameter search by modifying the architecture of the CNN. A simple network, with 2 hidden layers each having 64 units followed by 2 fully connected layers was finally used.

4) Multi-track PPO Training Setup: We train PPO with the same hyperparameters as single-track PPO above, but instead using our newly defined 'RoboNotesMultitrackEnv'.

5) BC Training Setup: We run BC using 20K sampled expert trajectories, for a total of 20 epochs and a batch size of 32. We use Adam optimizer and train until convergence. A 2-layer feed forward network is used to learn a policy network directly from the expert trajectories.

6) GAIL Training Setup: We run GAIL using 20K sampled expert trajectories, for a total of 500K timesteps. We alternate between training the generator (PPO) for 1024 updates and the discriminator (RewardNet) for 10 updates until both converge.



Run	Smoothed	Value	Step	Time	Relative
● baseline_random	49.5	61	1,474,968	12/12/22, 7:24 PM	2.425 min
● mod1_multitrack-ppo/PPO_1	80.87	80.95	1,474,560	12/11/22, 11:31 AM	9.425 min
● mod2_bc	58.5	58.66	1,473,506	12/12/22, 7:11 PM	4.474 min
● mod2_gail	62	61.72	1,473,506	12/12/22, 6:50 PM	4.71 min
● model-based_cem	100.9	103	1,474,980	12/12/22, 8:00 PM	1.254 min
● off-policy_dqn/DQN_1	86.35	86.54	1,474,000	11/8/22, 1:18 AM	7.233 min
● on-policy_ppo/PPO_1	97.94	97.98	1,474,560	11/9/22, 3:15 PM	8.092 min

Fig. 2. Plotting the average reward per episode for experiments random, PPO, DQN, CEM, BC, GAIL, and multi-track, with smoothing factor = 0.8. CEM outperforms the bunch with PPO close behind. All groups show significant improvement over the random agent.

There are a lot of hyperparameters for this method and we did not have time to do an exhaustive search. We used the same PPO hyperparameters as the experiments above.

B. *Qualitative*

We uploaded some videos of composition created by our RL agents to [https://tinyurl.com/52aj3brp]. The trajectory length (number of beats) for each composition is 20, with a tempo of 160 BPM.

The random samples are truly random, with notes jumping everywhere with no cohesion. CEM, PPO and DQN by comparison sound a lot more like music (though by no means master pieces of work). The compositions have notes in the same key and even some repeating motifs. Multi-track agent has more notes per beat, but due to lack of understanding of harmonies, it does not sound any better than the single-track options. In our opinion, the imitation learning experiments (BC and GAIL) sound the best qualitatively, with the least amount of repeating notes and more consistent composition.

VII. Conclusion and Future Work

We can use reinforcement learning techniques like PPO, DQN, and CEM to teach AI how to compose music from very basic music theory. In addition, we explored multi-track music generation and imitation methods (BC and GAIL) using expert trajectories generated from real compositions. Of the different methods, CEM and PPO are the easiest to train and fastest to converge to the highest return. Expert data with imitation techniques (BC and GAIL) greatly helps with the subjective quality of results, but since these methods are not optimizing our reward function directly, the quantitative return is not as good as CEM or PPO.

This points to a major limitation of our project: dependence on our custom reward function and its misalignment with qualitative results. While we can craft the reward function to avoid major failure modes (repeating notes, same key, etc.), it is much more difficult to teach the agent more nuanced musical notions (motifs, harmonies, etc.) directly from a rule-based reward function. To make further improvements, we must introduce some learned notion of musical quality. In GAIL, we learn a discriminator that is essentially a reward function (input transitions, output returns). With more time, we would like to explore using this learned reward for evaluation and see if it better correlates quantitative and qualitative results.

Another area for improvement is to better utilize our expert dataset. The pre-processing we current deploy greatly limits the potential of our raw MIDI dataset, since we are reducing it to single-track melodies. There is a lot more we can learn from this dataset if we could expand the scope of our problem space to incorporate different tempos, beats, note velocities and dynamics, harmonies, etc. An idea is to use our raw dataset to learn a feature encoder for representing music composition in latent space and a decoder to predict MIDI notes and dynamics (beats, velocities, etc.) from this latent representation. We can use the extracted features as a more rich observation space on which to run our RL algorithms.

With recent advances in AI, perceived creativity in music generation is improving at an alarming rate. While SOTA methods do not often use reinforcement learning, we had the opportunity to explore this topic and the efficacy of various RL techniques. The experience familiarized us with the world of RL, but more importantly sparked inspiration for future advancements in this area and beyond.

Third Party Libraries

We list the third party libraries used in this project:

- Gym [2]: environment API
- Stable-Baselines3 [5]: PPO and DQN experiments
- Skrl [7]: CEM experiments
- Imitation [3]: BC and GAIL experiments
- MIDIUtil and Mido [4]: reading/writing MIDI files
- SeeMusic [6]: visualizing MIDI files

References

[1] https://github.com/lucasnfe/adl-piano-midi.
[2] https://www.gymlibrary.dev/.
[3] https://imitation.readthedocs.io/en/latest/.
[4] https://pypi.org/project/MIDIUtil/.
[5] https://github.com/DLR-RM/stable-baselines3/blob/master/docs/index.rst.
[6] https://www.visualmusicdesign.com/.
[7] https://skrl.readthedocs.io/en/latest/intro/installation.html.
[8] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning, 2016. URL https://arxiv.org/abs/1606.03476.
[9] Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Generating music by fine-tuning recurrent neural networks with reinforcement learning. In *Deep Reinforcement Learning Workshop, NIPS*, 2016.