Combining Reinforcement Learning and Constraint Programming for Sequence-Generation Tasks with Hard Constraints

Daphné Lafleur

Polytechnique Montréal, Canada
Quebec Artificial Intelligence Institute (Mila), Canada

Sarath Chandar

Polytechnique Montréal, Canada
Quebec Artificial Intelligence Institute (Mila), Canada
Canada CIFAR AI Chair, Toronto, Canada

Gilles Pesant

Polytechnique Montréal, Canada

Abstract

While Machine Learning (ML) techniques are good at generating data similar to a dataset, they lack the capacity to enforce constraints. On the other hand, any solution to a Constraint Programming (CP) model satisfies its constraints but has no obligation to imitate a dataset. Yet, we sometimes need both. In this paper we borrow RL-Tuner, a Reinforcement Learning (RL) algorithm introduced to tune neural networks, as our enabling architecture to exploit the respective strengths of ML and CP. RL-Tuner maximizes the sum of a pretrained network's learned probabilities and of manually-tuned penalties for each violated constraint. We replace the latter with outputs of a CP model representing the marginal probabilities of each value and the number of constraint violations. As was the case for the original RL-Tuner, we apply our algorithm to music generation since it is a highly-constrained domain for which CP is especially suited. We show that combining ML and CP, as opposed to using them individually, allows the agent to reflect the pretrained network while taking into account constraints, leading to melodic lines that respect both the corpus' style and the music theory constraints.

2012 ACM Subject Classification Software and its engineering → Constraint and logic languages; Theory of computation → Reinforcement learning; Applied computing → Sound and music computing

Keywords and phrases Constraint programming, reinforcement learning, RNN, music generation

Digital Object Identifier 10.4230/LIPIcs.CP.2022.30

Supplementary Material *Software (Source Code)*: `https://github.com/chandar-lab/RL-Tuner-CP`

Funding Financial support for this research was provided by an IVADO Fundamental Research grant.

Acknowledgements We would like to thank all the members of Chandar Lab, Laboratoire Quosséça, family and friends who gave very useful feedback and improvements before the submission. We would also like to thank the CP 2022 reviewers for their constructive criticism.

1 Introduction

Recurrent Neural Networks (RNNs) [14] are a class of Machine Learning (ML) algorithms renowned for their ability to extract structural information from a corpus in order to generate sequences that mimic said corpus' style. However, some sequence-generation tasks require the final output to respect a set of rules. Music generation, especially applied to Renaissance

© Daphné Lafleur, Sarath Chandar, and Gilles Pesant;
licensed under Creative Commons License CC-BY 4.0

music, is a perfect example of these kinds of tasks. Given a series of melodic lines, RNNs are able to produce sequences that resemble the style of a given composer. Nonetheless, RNNs are not easy to control and it is hard to make them enforce music rules without injecting domain knowledge. It is akin to asking someone with no musical training to extract rules just from analyzing a pile of scores.

RL-Tuner [7] is an algorithm that uses Reinforcement Learning (RL) to bridge the gap between RNNs and hard constraints, resulting in samples that both better respect the constraints and are representative of the data the algorithm was trained on. However, the constraints are enforced by manually tuning a reward for each rule. In this work we use Constraint Programming (CP) with Belief Propagation (BP) to learn better how to satisfy constraints. By using CP instead of checking each rule individually, we benefit from the interactions between all the constraints and may anticipate violations. Furthermore, the addition of BP is a recent advance in CP that provides marginal probabilities for every selected value [12]. These marginals take into account the entirety of the sequence and act as a metric to determine if choosing an action could potentially lead to a complete sequence with no added constraint violations. This information is crucial in increasing what we introduce as the constraint satisfaction of the generated sequences.

We illustrate our framework by applying it to contrapuntal music writing in the style of the Renaissance period. We design two closely-related CP models implementing the rules of counterpoint for melody writing according to a standard textbook [15]. We train our RNN model using a Bach chorale corpus, a widely available source that is compatible to some degree with the constraints we enforce.[1] We combine our CP and RNN models within RL-Tuner and analyze the evolution of the reward to show that our algorithm retains its acquired knowledge from the Bach corpus while more-closely following the rules of counterpoint we added. We also discuss why some constraints may be harder to learn than others.

Our main contribution with this paper is to show that RL can be used to combine RNNs and CP in order to fine-tune training so that it generates sequences that reflect the corpus while enforcing constraints. Even though we applied it here to music generation, such a combination can be useful for other sequence-generation tasks with hard constraints.

In the rest of the paper, Section 2 provides some necessary background on RNNs, RL, CP/BP, and music theory. Section 3 surveys related work. Section 4 describes the core of our contribution: each component, how they interact, their specialization for melody generation. Section 5 presents an empirical evaluation of our contribution. Section 6 discusses our present and future work. Finally we conclude with Section 7.

2 Background

2.1 Recurrent Neural Network

Recurrent Neural Networks (RNN) [5] are a family of ML algorithms able to generate sequences. At every iteration, a token is passed through the network and used to predict the next token. After each prediction, the history is updated to make sure that the next token takes into account the whole sequence.

$$h_t = f(W_h x_t + U_h h_{t-1} + b_h) \qquad \mathbf{\hat{y}_{t+1}} = g(W_y h_t + b_y)$$

[1] Even though Bach belongs to the Baroque period, he would have been (heavily) influenced by Renaissance music.

In the first equation, we obtain the current history h_t by updating the old history h_{t-1} with the current token x_t. Weights W_h and U_h and a bias b_h are applied to control the importance of x_t and h_{t-1} in the update. The activation function f makes the update process non-linear. In the second equation, we compute $\mathbf{\hat{y}_{t+1}}$, a vector indicating the probability distribution of each possible value for the next token using the current history h_t. Once again, weights, a bias and a non-linear activation function are used for the prediction. With this distribution, we can select the value with the highest probability as $\hat{x}_{t+1}$. All the weights and biases are adjusted and learned during the training of the RNN (see below) to produce the best tokens. To measure how good an RNN is, we use the cross-entropy loss function

$$L = -\sum_t \mathbf{y_t} log(\mathbf{\hat{y}_t}).$$

For each token x_t in the dataset, we feed the previous tokens to the RNN and obtain the *predicted* distribution $\mathbf{\hat{y}_t}$. $\mathbf{y_t}$ represents the *target* distribution. It is a one-hot vector where all the components are 0, except for the component representing x_t, which is equal to 1.

Training is done with gradient descent by modifying the weights and the biases to decrease the loss. Once training is over, we use the final weights and biases to predict new sequences.

2.2 Reinforcement Learning and DDQNs

The basic idea of Reinforcement Learning [8] is to learn through interaction with an environment, in order to maximize the sum of the *rewards*. The agent will sample from the *policy* (state-dependent probability distribution) to pick an *action*. Once an action has been picked, the environment will update its *state* based on said action and return a reward, indicating how good the action is. This reward will then be used to update the estimated value of that state-action pair. Once the value has been updated, the policy will be adjusted to encourage actions that have a higher value.

Different families of RL algorithms define ways to compute state-action values. For example, in Q-Learning, the following equation is used:

$$Q(s,a) = [1-\alpha]Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$$

In this equation, state-action value $Q(s,a)$ is updated by summing the reward r and the maximum value of the next state s'. Two hyper-parameters α and γ control the magnitude of the update.

In Deep Q-Networks (DQN) [9], a *neural network* is used to compute the state-action value from the current state. To update the state-action values, we can compute the loss of the network by using the difference between the current value $Q(s,a)$ and the update part of the Q-learning equation (see above):

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a)$$

We can then adjust the parameters by applying one iteration of gradient descent per update to decrease this loss. However, since Q-Learning uses the maximum value of the next state ($\max_{a'} Q(s',a')$), overestimated values will be encouraged. Double Deep Q-Networks (DDQN) [17] reduce this effect by adding a second network to compute the state-action values:

$$\delta^A = r + \gamma Q^B(s', \mathtt{argmax}_{a'} Q^A(s',a')) - Q^A(s,a)$$

Here we use network A to pick the best next action with $\mathtt{argmax}_{a'} Q^A(s',a')$. However, instead of using A to compute its value, we will use network B, mitigating A's possible overestimation. Once δ^A has been computed, only the parameters of network A will be updated. Note that these two roles (A and B) are assigned randomly every time.

2.3 CP-Based Belief Propagation

CP-based Belief Propagation, introduced in [12], offers a new way to propagate constraints. Instead of simply removing unsupported values from domains, the solver computes the probability that each value will respect a given constraint. These partial probabilities are then sent as messages to the other constraints through their shared variables, which makes them update and refine their own probabilities. Out of these interactions, marginal probabilities are derived for each variable-value pair. These marginals approximate the probability that, if a variable is assigned a given value, all the constraints will be satisfied. They also provide information as to how many possible values respect the constraints.

2.4 Music Basics

We provide a brief introduction to music concepts essential to the understanding of this paper. Generally, music is built from musical notes each having a *pitch* and a *duration*. Pitches an octave apart are grouped in a *pitch class*. A *scale* is an organized subset of pitch classes. There are many kinds of scales – we will consider *diatonic* scales and in particular the major scale featuring the seven natural pitch classes (C, D, E, F, G, A, B). We loosely use the term *melody* for a sequence of notes (pitch, duration) sounded consecutively. An *outline* in a melody is a maximal subsequence of notes between a temporary high point and the next temporary low point (in terms of pitch) or vice-versa. The distance between two pitches is called an *interval*, identified by the integral number of pitch steps from one to the other (e.g. a third between C and E) and by the quality of the interval based on the number of semitones between them (e.g. a major third between C and E; a minor third between D and F). The melodic *motion* between two consecutive notes can either be by *step*, if the notes are adjacent in the scale, or by *skip*.

3 Related Work

The abundance of previous work on music generation, both in ML and in CP, shows that the task we are tackling is especially well suited for both these approaches. Surveys from Briot et al. [1], Fernandez Rodriguez and Vico [13], and the book edited by Truchet and Assayag [16] provide an extensive view on ML and CP being applied to music. In this literature review, we focus on the most relevant ones to us.

RL-Tuner [7] has been the greatest source of inspiration for our work. In the paper, the authors present a Reinforcement Learning algorithm that combines probabilities from an RNN with a reward function measuring how much the generated sequence respects a set of rules. They apply it to music generation. The RL agent used is a Double Deep Q-Network (DDQN). The authors show that RL-Tuner enforces both the music theory laws and the similarity to the RNN. The main difference from our work is that they compute the music theory reward by checking each rule individually in an *ad hoc* fashion based only on the previous notes. We argue that we can improve this algorithm by using CP with belief propagation to compute marginals that take into account the whole sequence and not just the previous notes.

On the ML side, C-RNN-GAN [10] is a classical music generation system that uses a generative adversarial network (GAN). GANs work by having two networks: the generator that creates the piece and the discriminator that tries to distinguish between fake and real music. The goal of the generator is to fool the discriminator, so it learns to create music that is as similar as possible to the dataset.

DeepBach [4] combines two LSTMs (a type of RNN) and one neural network. The first LSTM picks a note based on the *previous* notes. The second one picks a note based on the *future* notes. The neural network picks a note based on the notes that will be played at the same time on other voices. Finally, these three note candidates are sent to another neural network, which will choose the note to be added to the piece.

Coconet [6] uses a Convolutional Neural Network (CNN) to generate Bach chorales. CNNs are usually used for image classification tasks because they are able to handle 2D patterns (whereas RNNs are limited to linear sequences). Combining all four voices of a chorale creates a 2D structure that is well suited for CNNs. Coconet's CNN is used to compute probabilities of a note with respect to its context. Once the probabilities have been computed, Gibb's sampling is applied to populate the music piece. To remove bad choices due to less context information, a window of notes is resampled, with the window's size decreasing over time.

Anticipation-RNN [3] combines two RNNs to enforce unary constraints. In this algorithm, the first RNN is used to predict a unary constraint based on the future notes. This constraint is then used to condition the output of the second RNN, which will pick the note based on the partial sequence and the future constraints.

Young et al. [19] introduce a generative model able to generate music with relational constraints. What is really interesting about this work is that the constraints are synthesized from the dataset instead of being hard-coded. These constraints include equality of different notes and transposition of a sequence of notes (repeating the same pattern with each note transposed by the same amount). They try three different techniques to incorporate the constraints: sampling from the model and rejecting if the token doesn't respect the constraints, representing the constraints in a graph convolutional network, and using MIP to maximize an objective function.

4 Sequence Generation with Hard Constraints

Figure 1 gives an overview of our architecture. We describe the training process:

(i) The partial sequence is sent to the DDQN.
(ii) The DDQN produces the next note.
(iii) The RNN computes $p(a|s)$ from both the partial sequence s and the next note a. The CP models do the same to obtain the marginals and the violations.
(iv) The output of the RNN and of the CP models are used to compute the reward.
(v) The DDQN's weights are updated based on that reward and the next note is added to the sequence.

The different components are detailed below. Our code is publicly available[2].

4.1 CP models

Among several textbooks that teach counterpoint, the relatively recent "Modal Counterpoint, Renaissance Style" by P. Schubert [15] is of particular interest to create our CP models. It gathers rules from several treatises, including the seminal Fux, draws from the works of many period composers, and pays much attention to the quality of melodic lines. But foremost it strongly appeals to the constraint programmer: rules are declarative and classified as

[2] `https://github.com/chandar-lab/RL-Tuner-CP`

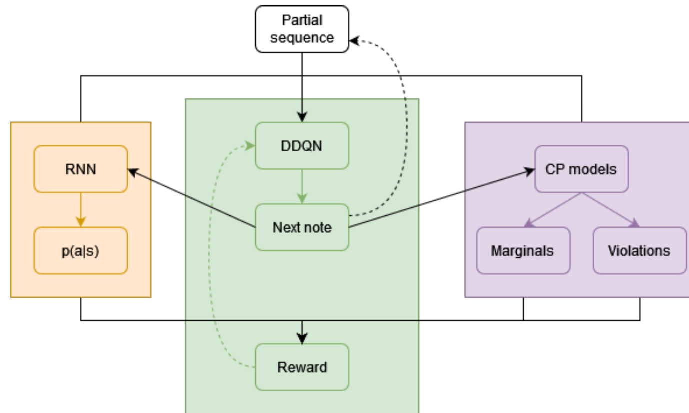

Figure 1 An overview of our training architecture.

hard and soft. The constraint models we built are based on the rules in this book. We implemented our models using the MiniCPBP solver[3] in order to have access to marginal probabilities.

4.1.1 Variables and domains

Even though rhythm is an important part of a melody, these rules (and those typical of the period, aside from the general recommendation that there should be rhythmic variety) do not take it into consideration. Accordingly, we represent a melody as a sequence of n pitches (notes). As is standard practice, we transpose the melodic lines from the corpus so that they are all in the same key (C major) and generate sequences in that same key.

Pitches should belong to the key, though we allow an accidental B♭ to avoid tritone intervals (i.e., augmented fourth or diminished fifth) from F. We also want a melody to stay within the octave range of its key, possibly extending it by one note below and above that range (resp. B♭ and D). The pitch values represent the number of semi-tones above the lowest pitch in the dataset (value 0 is G). Because of the range restriction explained above, the lowest value allowed in the CP domain is 3 (or B♭). We define our variables as:

$$\texttt{pitch}[i] \in \{3, 4, 5, 7, 9, 10, 12, 14, 15, 16, 17, 19\} \quad 1 \leq i \leq n$$

Schubert's book adds restrictions about the permitted values of the intervals. An interval can span no more than a sixth (9 semi-tones), except that an octave is also allowed and that a tritone is not. That interval can be sung up or down. We also want to avoid intervals of a sixth, except for an ascending minor sixth. Finally, two consecutive notes should not have the same pitch:

$$\texttt{interval}[i] \in \{\pm 1, \pm 2, \pm 3, \pm 4, \pm 5, \pm 7, +8, \pm 12\} \quad 1 \leq i \leq n-1$$

An interval is computed as the number of semi-tones between two consecutive notes. In other words:

$$\texttt{interval}[i] = \texttt{pitch}[i+1] - \texttt{pitch}[i] \quad 1 \leq i \leq n-1$$

[3] https://github.com/PesantGilles/MiniCPBP

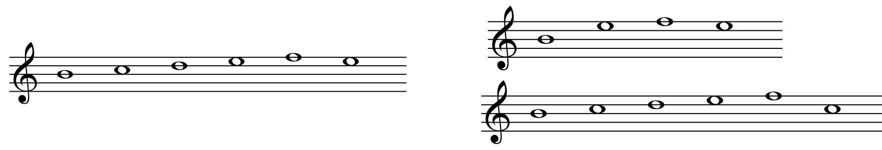

Figure 2 A legal augmented fifth outline (left) and two forbidden ones (right).

4.1.2 Constraints

Now that the main variables have been defined, we describe the melodic constraints we consider. Though some of them are considered soft by Schubert, here all are expressed as hard constraints for our system to learn.

(i) End on the tonic.

$$\texttt{pitch}[n] = \text{C}$$

(ii) End by stepwise descent.

$$-2 \leq \texttt{interval}[n-1] < 0$$

(iii) Use more steps than skips.

$$\sum_{i=1}^{n-1} (\texttt{interval}[i] < -2 \vee \texttt{interval}[i] > 2) < \frac{n-1}{2}$$

(iv) An accidental B♭ should be followed by a descending interval.

$$\texttt{pitch}[i] \neq \text{B}\flat \vee \texttt{interval}[i] < 0 \qquad 1 \leq i \leq n-1$$

(v) Tritone outlines. An outline of an augmented fourth is prohibited. An outline of a diminished fifth is allowed only if it is completely filled in by step (interval smaller than 2) and then followed by a step in the opposite direction. Fig. 2 gives examples of one legal and two forbidden tritone outlines. This rule is not as straightforward to express and requires that we consider several consecutive notes.
A useful observation is that the only allowed tritone outline spans four steps (which could correspond to more than five notes if we have repeated notes). We express this using a `regular` [11] constraint on `interval` variables with an automaton recording in its states the number of steps and semi-tones in a potential outline. Fig. 3 gives the automaton for ascending outlines – the case of a descending outline is similar. Though admittedly a little hard to decipher, it shows our ability to model complex rules.

(vi) A skip should be preceded or followed by a step in the opposite direction.

(vii) Avoid more than two successive skips.

(viii) Avoid skipping on both sides of a temporary high or low point.

(ix) Two successive skips in the same direction should be small. We consider a skip of a third or fourth to be small.

(x) "Pyramid" rule. An ascending outline should not have large skips following smaller skips or steps; a descending outline should not have large skips preceding smaller skips or steps. As when building a pyramid, larger blocks should be used at the bottom and smaller ones at the top. (Keep in mind that such rules are derived from practice at that time according to aesthetics and "singability".)
Because they are closely related, the previous five rules (**(vi)** to **(x)**) are handled together, through a single `cost-regular` [2] constraint on a characterization of individual intervals as ascending/descending steps, small skips (a third or a fourth), and skips (fifth, sixth,

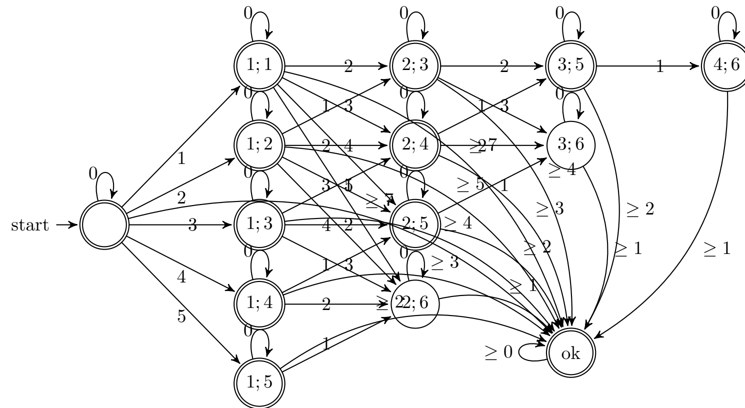

Figure 3 Automaton to check for ascending tritone outlines. Negative transitions to descending tritone states are not shown.

or octave). It is sufficient to create a state for each pair of characterizations of intervals. All states are accepting but each transition carries a cost corresponding to the sum of the penalties for every rule broken on that transition. An interval value of 0, corresponding to a repeated note, loops to the same state. In the interest of clarity, we do not show this automaton.

(xi) **Modal range.** A melody must cover (include notes from) the whole octave range. Longer melodies are encouraged to cover that range every so many notes. Through a single `gcc` constraint on `pitch` variables, we express the requirement that each pitch in that range should appear at least once (setting their minimum number of occurrences to 1).

(xii) **Minimum number of modal skips.** The final of the mode (C) and the pitch class four steps above it (G) are more important than the other pitch classes. Skips between these two pitches are characteristic of the mode and help establish it. We can enforce it by lower bounding to 3 the sum of reified constraints expressing modal skips.

$$\sum_{i=1}^{n-1}[(\texttt{pitch}[i] = \text{C} \land \texttt{pitch}[i+1] = \text{G}) \lor (\texttt{pitch}[i] = \text{G} \land \texttt{pitch}[i+1] = \text{C})] \geq 3$$

We thus have a total of 12 constraints $-$ 4 (because 5 constraints are handled together) + 4 constraints (the ones used to restrict the domains of our variables) = 12.

4.1.3 Marginals model

Our goal is to have a model that, given a sequence of notes, will compute the marginal probabilities of the next note. Since that sequence is provided by the RL-Tuner, it is possible that there have already been constraint violations. To clearly measure the impact of the next note, regardless of previous violations, we apply the constraints and domain restrictions only on the future notes. However, this requires some adjustments in our constraints. For instance, since our automata can no longer start at the beginning of our sequence, we need to modify the starting state according to the previous notes. Furthermore, if we want to respect a certain lower bound (for example, the minimum number of modal skips in constraint **(xii)**), we need to adjust that bound based on the input to the CP model. After the first propagation of the constraints, we compute the marginals using belief propagation and return the corresponding value for the chosen next note, provided by the RL agent.

4.1.4 Violations model

This model is very similar to the one presented above. The main difference is that, instead of enforcing the constraints, we use reified constraints to count the number of violations for each constraint. This causes a little bit of a challenge for some constraints. The tritone outline constraint **(v)** is limited to 25 different interval values (the domain of the intervals is from -12 to +12). However, if we allow the possibility of any interval, which we must since no constraints are enforced, we have a total of 57 different interval values (from -28 to +28). To deal with values not supported by the automaton, we use a soft version of that constraint [18]. If the interval is illegal, the tritone outline constraint will return one constraint violation.

Something similar is done for the other automaton (constraints **(vi)** to **(x)**). However, since this is a `cost-regular` constraint, we cannot assign to it a single violation. To compute the number of violations for this constraint, in case of an invalid interval, we decompose the cost in two sections. First, we compute the cost associated to the current transition by averaging the costs associated to transitions from the current state. Afterwards, we compute the cost of the future transitions with the same `cost-regular` constraint, but applying it only on the future transitions and modifying the current state.

For the rest of the constraints, we use reified constraints. For example, instead of restricting the domain of the notes, we compute the number of notes that are outside of this domain. The number of violations for each constraint is then a variable, and we return the smallest value in its domain.

This model receives as input the note sequence and the value of the chosen note. It will then only return the minimum number of violations for each of the twelve constraints, for that chosen note.

4.2 Training the RNN

We extracted melodic lines from the Boulanger-Lewandowski corpus of Bach chorales[4]. We used publicly-available code[5] to preprocess our midi files. During this preprocessing, we check every repeated note (midi files having a really small granularity, every repeated note has to be a note that is held) and replace the subsequent repetitions with a special token, indicating which notes are held. We then extracted melodic lines using a quarter-note granularity, where we ignored the hold token since we do not take into account the note duration. We transposed all sequences in the key of C and kept only the chorales in C major, for a total of 200 sequences. That dataset was split into 150 examples for training and 50 examples for validation. In order to have notes in similar ranges, we used only the soprano voice which usually represents the main melody. Since the RNN used in the RL-Tuner was trained with the Magenta library, we created our own set of inputs and outputs, using one-hot encoding of length 29 – two and a half octaves, similar to the vocal range – for the input representing the last note and a single value for the output representing the predicted note. We trained the RNN on 1000 iterations using the basic-rnn configuration provided by Magenta[6] with a batch size of 64 and two layers of 64. These hyperparameters were tailored to our dataset. The loss function used is the softmax cross-entropy loss (softmax meaning that the activation function used to compute the probability distribution is the softmax function).

[4] `http://www-ens.iro.umontreal.ca/~boulanni/icml2012`
[5] `https://medium.com/analytics-vidhya/convert-midi-file-to-numpy-array-in-python-7d00531890c`
[6] `https://github.com/magenta/magenta/tree/main/magenta/models/melody_rnn`

Though we were able to reach a training accuracy of 90%, the final validation accuracy obtained was 50% (compared to 92% in the original paper). This is likely due to the size of our dataset (200 sequences vs 30 000), which is substantially smaller than the one used in the RL-Tuner (more on that in Section 7).

4.3 RL-Tuner

We copy the weights of the RNN to initialize the DDQN. Before training, the value of each action (or note) is equal to the probability returned by the RNN since it is exactly the same network. During training, the next note, sampled from this distribution by the DDQN, is used to compute the reward, by using the probability returned by the RNN and evaluating with respect to each constraint applied. With the final reward, the weights (and the policy) of the DDQN are updated, without changing the original RNN, thus making the DDQN able to learn independently from the RNN.

The architecture briefly described above can also be found in the original RL-Tuner paper [7]. The sections below explain the changes we made to the RL-Tuner in order to include our CP models.

4.3.1 Restricting sampling to the feasible domain

In the original paper, a note is chosen either randomly or sampled from the DDQN probabilities. This leads to a lot of bad choices with respect to the constraints. That is why we considered an additional option, where the output of the DDQN is restricted according to the number of violations, at training time only. In other words, the algorithm can only sample from notes that have 0 violations. We called this option “restrict domain”.

4.3.2 Reward functions

We implemented seven different reward functions. Here v represents the number of violations for each constraint, m is the marginal for the chosen note and $p(a|s)$ is the probability of the chosen note a according to the RNN, based on s, the partial note sequence. We added two constants, k and c to weigh the different parts of the reward and bring them to a similar range [7].

$$\begin{aligned}
\texttt{violations} &= -\sum v \\
\texttt{marginals} &= km \\
\texttt{marginals_violations} &= km - \sum v \\
\texttt{rnn} &= log(p(a|s)) \\
\texttt{rnn_violations} &= log(p(a|s)) - c\sum v \\
\texttt{rnn_marginals} &= log(p(a|s)) + ckm \\
\texttt{rnn_marginals_violations} &= log(p(a|s)) + c(km - \sum v)
\end{aligned}$$

5 Experiments

We ran experiments to answer the following research questions:

1. Can CP reduce the number of constraint violations?
2. Can this be done without forgetting stylistic knowledge acquired by the RNN?
3. What is the impact of restricting the domain while sampling notes?
4. Are all constraints as easy to learn?

[7] Unlike for the RNN probabilities we do not use log-marginals because several marginals are null, which would generate $-\infty$ rewards.

Table 1 Hyper-parameters of our RL-Tuner.

Parameter	Value	Parameter	Value
random_action_probability	0.1	one_hot_length	29
store_every_nth	1	algorithm	q
train_every_nth	5	reward_scaler (c)	2
minibatch_size	32	cp_reward_scaler (k)	40
discount_rate	0.5	output_every_nth	5000
max_experience	100000	num_notes_in_melody	32
target_network_update_rate	0.01	num_steps	50000
rnn_layer_sizes	[64, 64]	exploration_period	25000

5.1 Experimental Setup

We performed experiments for each reward function twice, one where we restricted the domain and one where we did not. We chose the same constant c as in the original paper [7], since it allegedly produced better samples. The value for k was set to 40, converting our marginals to values between 0 and 40. Table 1 presents the hyper-parameters chosen to train our model as named in the RL-Tuner configuration.

Each experiment was averaged across 10 different seeds for a total of 50 000 iterations. Every 5000 iterations, we generated 10 sequences of 32 notes to evaluate our model and measure the evolution of our metrics. We focused mainly on two performance metrics. To answer Question 1, we measure the `constraint satisfaction`, a value between 0 and 1 indicating how good the algorithm is at following constraints (1 being perfection):

$$\texttt{constraint_satisfaction} = \frac{\sum_{i=1}^{n}(1 - \frac{v_i}{v_i^{\max}})}{n}$$

where n is the number of constraints, v_i is the number of violations for constraint i, summed over all 10 sequences, and $v_i^{\max}$ is the maximum number of potential violations for that constraint during all the 10 sequences.

The second metric used to answer Question 2 is $log(p(a|s))$, the `rnn reward`. For each chosen note, we check the RNN probability and compute an average reward over every note and every sequence. The higher it is, the closer the generated sequence is to the trained model. This metric was also used in the RL-Tuner paper to show that the model does not "forget" the stylistic knowledge learnt from the corpus while enforcing the constraints.

Question 3 can be answered by comparing our two metrics, while restricting the domain or not. To answer Question 4, we analyze the number of violations for each constraint separately by plotting the values returned by the violations model.

Table 2 presents the final constraint satisfaction and average rnn reward for each of our reward functions.

5.2 Comparing metrics to the RL-Tuner

Since we have a different corpus (Bach instead of pop music) and different constraints, we cannot compare our results directly with those in the RL-Tuner paper. However, line 4 without domain restriction allows us to compare to the RL-Tuner approach because we only count the number of violations. We can see here that line 4 is better than line 7. This is similar as what was showed in the paper. However, we can also see that adding the marginals in the mix offers another improvement of 4%.

Table 2 Constraint satisfaction (sat) and average rnn reward ($\log(p(a|s))$) with different reward functions. *Takeaway:* rnn + marginals + violations yields the best constraint satisfaction and its rnn reward is still higher than without using CP.

	Restrict domain			
	No		Yes	
Reward function	sat (%)	$\log(p(a\|s))$	sat (%)	$\log(p(a\|s))$
1. violations	60.1	-4.9	62.1	-4.5
2. marginals	75.1	-2.1	75.8	-2.2
3. marginals + violations	76.9	-1.8	74.3	-2.3
4. rnn + violations (similar to RL-Tuner)	77.2	-1.8	74.7	-2.4
5. rnn + marginals	77.4	**-1.4**	76.6	**-2.0**
6. rnn + marginals + violations	**81.6**	-1.9	75.4	-2.3
7. rnn	74.5	-2.2	**76.9**	**-2.0**

5.3 Without domain restrictions

Without domain restrictions, as we expected, most of our CP models yield a better constraint satisfaction than just using the output from the RNN (lines 2 to 6 vs 7). This shows that, overall, CP helps the agent to respect the constraints. However, it is not the case for the violations reward function, which does a lot worse than the RNN alone (line 1 vs 7). Further experimentations would be required to understand if it simply takes more iterations to see an improvement in that case.

We wondered if CP and the RNN would counteract each other, the RNN wanting to stay close to the corpus and CP aiming to constrain the algorithm. On the contrary, combining them yields better constraint satisfaction than using them separately (lines 4 to 6 vs 1 to 3). This could be due to our choice of corpus: it is likely that Bach generally followed the music rules we imposed and so the RNN learnt them to some degree. However, adding the marginals and the violations gave the model a second push towards a better constraint satisfaction. The same can be said for the rnn reward. Yet again, it seems like the CP model generally helps the RNN to stylistically represent the corpus (lines 4 to 6 vs 1 to 3). This is once again due to concordance between our corpus and our constraints.

5.3.1 Combining violations and marginals

The reward function yielding the best constraint satisfaction is the `rnn + marginals + violations` function (line 6). Both CP models are necessary because the marginals provide information about how good the chosen note is to respect the constraints in the long run and the violations provide information about how big the constraint violation is. If a note does not break any constraint, the violations will be 0 but the marginals will give new information (it could return 100% if the chosen note is the only good choice). If a note breaks some constraints, the marginals will be 0 but the violations will provide new information on the number of constraints that were violated, in other words, how bad this choice is. That's why the best performance is obtained by combining marginals and violations, both providing information in different contexts.

The algorithm with the highest rnn reward is the `rnn + marginals` (line 5). It appears that adding the violations to this function increases the constraint satisfaction at the expense of the rnn reward. However, even if the rnn reward of line 6 (best constraint satisfaction) is lower than line 5, it is still an improvement over the initial `rnn` (line 7).

Figure 4 Number of violations for each constraint for different reward functions without domain restriction (red = rnn + violations, blue = rnn + marginals, green = rnn + marginals + violations, gray = rnn). *Takeaway:* For most of the constraints, the green curve quickly learns to produce close to no violations, better than the other curves.

5.4 With domain restriction

Now if we compare with the right part of Table 2, we see that adding domain restriction seems to have a mixed effect on both performance metrics. Only for lines 1, 2 and 7, we can see some improvement. What is interesting is that these three reward functions are the functions with only one of our three components (rnn, marginals and violations). It seems as if domain restriction has a good impact when the reward function is simple, but is harmful when the complexity increases.

Another interesting fact is that generally both metrics evolve in the same direction. Indeed, when we compare both halves of Table 2 on the same line, there is only one occurrence of the constraint satisfaction increasing and the rnn reward decreasing. This shows an alignment between the corpus and the constraints.

5.5 Comparing constraints

Figure 4 shows the number of violations during all 50 000 iterations, for each constraint enforced, averaged over 10 seeds. We are mostly interested in the green curve, representing the `rnn + marginals + violations` reward function. As we can see, the first seven constraints and the last constraint have a very similar behavior. Indeed, the number of violations drops close to 0 at around step 100 (20 000th iteration). However, constraints 8 to 11 seem to be a lot harder to learn. This can be explained by the fact that these constraints create violations only at the end of the sequence. For two of these four constraints, the best reward function

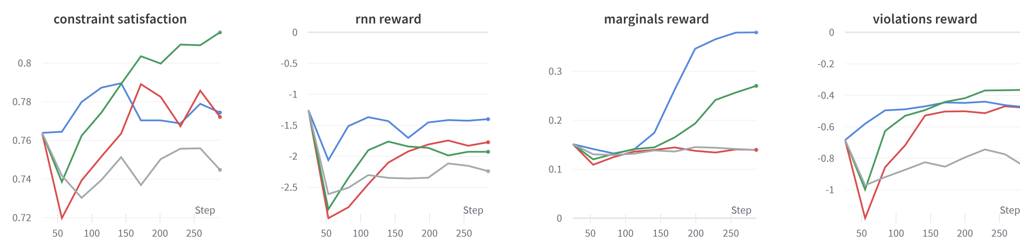

Figure 5 Evolution of our four metrics with different reward functions (red = rnn + violations, blue = rnn + marginals, green = rnn + marginals + violations, gray = rnn). The domains were not restricted. The origin of each curve shows the initial value before training the DDQN. *Takeaway:* All of the CP curves improve on the gray curve for all the metrics. The green curve achieves the best constraint satisfaction and the best violations reward.

is the `rnn` itself, which could mean that constraints only having an impact at the end are easier to learn from data than by using CP. We can also see that the green curve yields fewer violations for 9 out of 12 constraints.

The unexpected increase of the blue curve for constraint avoidSixths could be a sign that an agent keeping track only of the marginals doesn't hesitate to break more constraints when the marginals are already 0. That could explain why this behavior is not seen for reward functions that take into account the number of violations.

5.6 Comparing rewards

Figure 5 shows the evolution of the constraint satisfaction and the three rewards averaged over 10 seeds. By looking at these plots, it is obvious that adding CP outputs improves all four metrics. The great improvement for the constraint satisfaction, the marginals reward and the violations rewards is not surprising since the CP outputs are added to enforce constraints. However, even the rnn reward is increasing, showing that satisfying constraints also improves the capacity of the RL agent to reflect what the RNN learned from the corpus. We can also see that the `rnn + marginals + violations` function (green curve) is the winner for half of the metrics.

6 Discussion and future work

Even though our algorithm is an extension of the RL-Tuner, our results cannot be compared directly with those in the original paper. This is mainly due to our choice of corpus and constraints. However, the violations returned by the CP model are similar in spirit to evaluating each constraint one by one as is done in the original paper. We can then consider the `rnn + violations` reward function as reflecting what the original paper would have obtained, had the authors used the same RNN and constraints. If we use this reward function as a baseline for comparison, we see that the marginals provide an advantage and that the best performance is achieved by combining them with violations. Furthermore, we see that most of our constraints reach 0 violations in the first 20 000 iterations which decreases the number of steps required for convergence (more than 50 000 iterations for the red curve).

Experiments on the number of BP iterations required for convergence of the marginals could be useful to increase the speed of the CP model. As of now, it takes around 500 ms to compute either the marginals or the violations for each partial note sequence. The full training process for all 50 000 iterations takes a few hours. To increase efficiency, we

store a list of marginals and violations to reuse them. With the marginals and violations already computed, the training process takes a few minutes. Unfortunately the length of the sequences prevents us from precomputing them for each possible note sequence.

It would also be relevant to evaluate our algorithm with a human study (e.g. Turing test or Likert scale). Given that our task was to generate melodic lines, we could not create samples without rhythmic information. Ongoing work adds rhythmic constraints to convert melodic lines to note sequences with pitches and durations. This will allow us to generate samples and conduct human studies.

We are aware that the performance of our RNN is not nearly as high as that of the one used in the RL-Tuner paper. However, our goal here was to show that the RL agent is able to combine the RNN's predictions with the constraints, and this goal is not affected by the initial RNN performance. To evaluate the aesthetic quality of generated sequences, we would need to have an RNN with a higher accuracy.

We also plan to investigate such a combination of ML and CP for sequence generation tasks in other application domains.

7 Conclusion

In this work, we presented an extension of the RL-Tuner algorithm: adding the output of CP models to the reward function in order to learn hard constraints. We applied our algorithm to the generation of melodic lines. We showed that combining the pretrained RNN's probability with the marginals and the number of constraint violations yields the best constraint satisfaction, while increasing the rnn reward obtained by the model without CP. This shows that combining ML and CP yields generated sequences that are better at both reflecting the corpus and respecting constraints than using only ML (or CP). We also studied each constraint individually and showed that for most of the constraints, the best reward function quickly converged to no violations. Our work allowed us to generate melodic lines that respect constraints without forgetting structural knowledge obtained from the Bach corpus.

References

1 Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation - A survey. *CoRR*, abs/1709.01620, 2017. arXiv:1709.01620.

2 Sophie Demassey, Gilles Pesant, and Louis-Martin Rousseau. A Cost-Regular based Hybrid Column Generation Approach. *Constraints*, 11(4):315–333, December 2006. doi:10.1007/s10601-006-9003-7.

3 Gaëtan Hadjeres and Frank Nielsen. Anticipation-rnn: enforcing unary constraints in sequence generation, with application to interactive music generation. *Neural Comput. Appl.*, 32(4):995–1005, 2020. doi:10.1007/s00521-018-3868-4.

4 Gaëtan Hadjeres, François Pachet, and Frank Nielsen. Deepbach: a steerable model for bach chorales generation. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 1362–1371. PMLR, 2017. URL: http://proceedings.mlr.press/v70/hadjeres17a.html.

5 Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, December 1997. doi:10.1162/neco.1997.9.8.1735.

6 Cheng-Zhi Anna Huang, Tim Cooijmans, Adam Roberts, Aaron C. Courville, and Douglas Eck. Counterpoint by convolution. *CoRR*, abs/1903.07227, 2019. arXiv:1903.07227.

7 Natasha Jaques, Shixiang Gu, Richard E. Turner, and Douglas Eck. Tuning recurrent neural networks with reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017. URL: `https://openreview.net/forum?id=Syyv2e-Kx`.

8 Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *CoRR*, cs.AI/9605103, 1996. `arXiv:9605103`.

9 Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

10 Olof Mogren. C-RNN-GAN: continuous recurrent neural networks with adversarial training. *CoRR*, abs/1611.09904, 2016. `arXiv:1611.09904`.

11 Gilles Pesant. A regular language membership constraint for finite sequences of variables. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 482–495, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

12 Gilles Pesant. From support propagation to belief propagation in constraint programming. *J. Artif. Intell. Res.*, 66:123–150, 2019. `doi:10.1613/jair.1.11487`.

13 Jose David Fernández Rodriguez and Francisco J. Vico. AI methods in algorithmic composition: A comprehensive survey. *CoRR*, abs/1402.0585, 2014. `arXiv:1402.0585`.

14 Robin M. Schmidt. Recurrent neural networks (rnns): A gentle introduction and overview. *CoRR*, abs/1912.05911, 2019. `arXiv:1912.05911`.

15 Peter Schubert. *Modal Counterpoint, Renaissance Style*. Oxford University Press, 2nd edition, 2008.

16 Charlotte Truchet and Gérard Assayag, editors. *Constraint Programming in Music*. Wiley, 2011.

17 Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015. `arXiv:1509.06461`.

18 Willem Jan van Hoeve, Gilles Pesant, and Louis-Martin Rousseau. On global warming: Flow-based soft global constraints. *J. Heuristics*, 12(4-5):347–373, 2006. `doi:10.1007/s10732-006-6550-4`.

19 Halley Young, Maxwell Du, and Osbert Bastani. Neurosymbolic deep generative models for sequence data with relational constraints, 2021. URL: `https://openreview.net/forum?id=Y5TgO3J_Glc`.