Co-creative Music Synthesizer Patch Exploration

Sean Luke and **Victoria Hoyle**
Department of Computer Science
George Mason University
Washington, DC, USA
sean@cs.gmu.edu
victoria.hoyle@protonmail.com

Abstract

We present an interactive evolutionary approach to exploring the space of synthesizer patches which combines an evolutionary optimizer with a variational autoencoder neural network. The objective is to work with musicians to explore the complex space of patches rather than program patches themselves, an often tedious and difficult task. The technique uses an algorithm to wander through the parameter space, while engaging the musician in assessing the quality of discoveries and providing real-time feedback to the algorithm. We describe the method and argue for it as a co-creative system.

Introduction

A *patch* is a program for a music synthesizer which directs it to produce a given kind of sound when played by a musician. The term dates from early synthesizers, which were programmed by connecting various modules with *patch cables* to control the flow of audio and modulation signals. Nowadays a patch is typically a fixed-length array of parameter values which together specify the nature of the sound generation elements used, their settings, and their connections.

Programming synthesizer patches can be daunting. While earlier synthesizers had relatively few parameters, modern synthesizers can have many hundreds of them. Indeed, some *additive synthesizers* have several thousand parameters, presenting a difficult high-dimensional design space. Still other synthesizers, such as *romplers*, have parameters consisting of many hundreds of unordered options. Critically, the relationships between parameters may be nontrivial. Some kinds of synthesizers, such as *subtractive synthesizers*, have parameters which are relatively independent of one another, and so their effect on the overall sound can be predicted and tuned independently. But other synthesizers, such as *frequency modulation (FM) synthesizers,* have parameters with strong and nonlinear relationships, and the impact of changing one parameter will strongly depend on the settings of others.[1]

[1]Some FM synthesizers were so difficult to program that musicians resigned themselves to playing only the factory patches which came on the units: these patches have since become famous. For example the Yamaha DX7's *E. Piano 1* and *Bass 1* factory patches were used on numerous pop songs, as was the Yamaha TX81Z's *LatelyBass* patch.

Finally many synthesizers, particularly those from the 1980s and 1990s, have very poor interfaces, making programming them from their front panels tedious.

But synthesizers do not have to be programmed only from their front panels: they can also be programmed via MIDI, a standardized serial port interface with a packet protocol. This makes it possible to design software tools called *patch editors* which allow the musician to program the synthesizer remotely using a better quality interface on a computer screen. However even with an improved interface, the number and complexity of synthesizer patch parameters can still make programming them a very difficult challenge.

An alternative is for the musician to collaborate directly with the patch editor in exploring the patch space. As it turns out, no less than Brian Eno proposed exactly this idea in a 1995 letter to Stewart Brand. He wrote:

> But what if the synthesizer just "grew" programs? If you pressed a "randomize" button which then set any of the several thousand "black-box" parameters to various values, and gave you sixteen variations. You listen to each of those, and then press on one or two of them — your favourite choices. Immediately the machine generates 16 more variations based on the "parents" you've selected. You choose again. And so on . . . . The attraction of this idea is that one could navigate through very large design spaces without necessarily having any idea at all of how any of these things were being made. I want to get some synth manufacturer interested in this. They are not too bright, in my opinion, so this might take a long time . . . . (Eno 1996) [p. 190].

In Luke (2019) we developed a method for doing this via interactive evolutionary optimization in Edisyn, a popular patch editor of our own design. Using this method, the editor wanders through the space of patches, discovering, proposing and auditioning ones to the musician, who assesses them. These assessments guide the editor in its search for new and better patches. In this paper we present an extension to this method which employs a combination of evolutionary optimization with a variational autoencoder trained on a large number of patches developed by the synthesizer community. In short, the method wanders not through the space of *all* patches, but through a manifold or subspace of patches which resemble, to some degree, the community patches themselves. We then discuss how and whether this back-and-forth between the program and musician is co-creative.

Figure 1: Edisyn's Yamaha DX7 patch editor, showing the "Global" and "Operators 1–2" panes.

Evolutionary Computation Evolutionary computation (or EC) is a family of stochastic optimization algorithms of which probably the most famous example is the Genetic Algorithm. An EC algorithm starts with a sample of randomly-generated candidate solutions (a *population* of *individuals*). It tests each individual according to some objective (or *fitness*) function. It then *breeds* a next-generation population by iteratively selecting and copying individuals from the previous population (the *parents*), recombining (mixing and matching) elements of the copies, and mutating the recombined copies with some degree of noise, producing their *children*. The selection procedure is biased to tend to select fitter individuals. Ideally over successive generations the current population improves in fitness. See Luke (2013) for more on EC.

Usually the fitness function is an automated procedure, but in our approach, the fitness of an individual is computed by auditioning the individual (the patch) in front of a human (a musician or sound designer), who offers an assessment. This approach is commonly known as *interactive evolution* and has been applied to a very wide range of fields ranging from art to robotics to industrial design (Takagi 2001).

Previous Work

The seminal paper in evolutionary patch optimization was Horner, Beauchamp, and Haken (1993), in which patches were proposed, played on the synthesizer, and then automatically compared for error against a target sound. Thus the fitness function was an automated procedure. This approach is known as *evolutionary resynthesis*.

Some later work has focused on interactive evolution, using a human to assess patch individuals. An early and well-known implementation of Brian Eno's original idea is *MutaSynth* (Dahlstedt 2001), a manual patch-recombination method which eventually found its way onto the commercial editor for the Nord Modular G2 synthesizer. The interactive evolution literature has considered different ways to deal with the difficulties inherent in auditioning patches, which take up time, for humans, who are easily bored. McDermott, O'Neill, and Griffith (2010) focused on interfaces designed to speed the assessment and selection of solutions. Seago (2013) simplified the search space by updating a parameterized model instead of a sample (essentially a form of *estimation of distribution algorithm*, see Luke (2019)). Suzuki et al. (2011) also simplified the search space by restricting candidate solutions to those drawn from an existing corpus of patches.

Rather than use neural networks in conjunction with evolutionary optimization as we have, some work has applied evolutionary computation in the *development* of neural-network-based synthesis methods (Ianigro and Bown 2016; Jónsson, Hoover, and Risi 2015).

Edisyn

Edisyn is a popular open source patch editor library of our design written in Java.[2] Edisyn has 76 patch editors supporting 139 synthesizers from 39 different families, plus editors for microtonal scales and for general MIDI parameter editing. These editors cover a wide range of synthesizer types: additive synthesizers, subtractive, rompler, drum, FM, and hybrid synthesizers; plus samplers, MIDI routers, and controllers. It attempts to present these using a unified and consistent interface. Figure 1 shows two of four panes from Edisyn's patch editor for the Yamaha DX7, a famous FM synthesizer.

Edisyn allows the musician to connect to a remote synthesizer over MIDI, then play notes on the synthesizer, change parameters in real-time, upload and download patches from the synthesizer's current working memory (the patch it is presently playing), read and write patches to the synthesizer's long-term patch storage, and load and save patches to disk on the musician's laptop. Edisyn also offers a *librarian*, essentially a spreadsheet of all patches on the synthesizer for bulk modification and organization.

Edisyn is distinguished among patch editors by its extensive set of automated patch exploration tools. This includes patch mutating, recombining two or many patches to form a child, "nudging" patches towards or away from other patches, and real-time morphing of patches as interpolations of up to four other patches. These features can be constrained in several ways, notably by restricting the parameters permitted to be mutated or recombined, and by specifying the degree of mutation or recombination involved. Prominent among the patch exploration tools is Edisyn's *Hill-Climber.*

The Hill-Climber

Edisyn's Hill-Climber is a patch space exploration tool using interactive evolution. It employs a variation of a so-called (μ, λ) Evolution Strategy algorithm with a highly customized

[2]Edisyn may be downloaded at https://github.com/eclab/edisyn

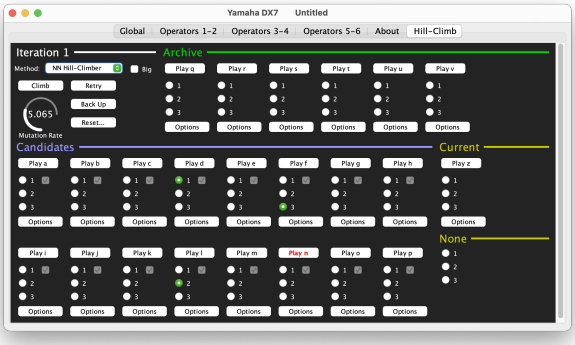

Figure 2: Edisyn's Hill-Climber in 16-Candidate mode (Variational Autoencoder turned on).

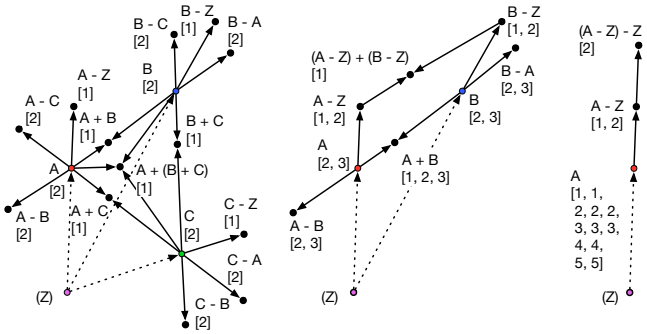

Figure 3: Breeding mechanism for the Hill-Climber when the musician has selected three parents (left), two (center), or one (right). A, B, and C represent these parents ranked best to worst by the musician, and Z is the previous generation's parent A. Children are produced near locations represented by nodes in the graph (other than Z). A child is produced by combining parents as shown ($+$ denotes standard recombination and $-$ represents opposite-recombination), then applying mutation. The notation $[a, b, ..., n]$ indicates the number of children produced at a location and their mutation counts. For example, $A + B$ [1, 2, 4] means that three children are produced by recombining A with B: one child is mutated once, one is mutated twice, and one is mutated four times.

recombination and mutation method. For the Yamaha DX7 family of synthesizers this facility is further augmented with a deep-learned neural network (a variational autoencoder), as discussed later. The Hill-Climber, set up with the variational autoencoder, is shown in Figure 2.

The musician initializes the Hill-Climber by selecting a patch as a starting point. The Hill-Climber then seeds itself with 16 or 32 patches randomly selected from the vicinity of the initial patch. These patches are sent to the synthesizer to be auditioned to the musician one by one; the musician can request to re-audition a patch one at any time. The musician selects and ranks up to three patches as favorites. The Hill-Climber then *breeds* these patches to produce a new generation of 16 or 32 new patches in their vicinity. The new patches are auditioned to the musician and the process repeats.

At any time the musician can edit patches under consideration, save them, move them to other patch exploration tools, or back up to or build a new set of patches. The musician can designate a patch to be one of six "hall of champions" patches: any time later they may select and rank any "champion", as well as the current patch being edited, instead of an individual from the current generation. Finally, the musician can restrict the parameters that the algorithm is permitted to modify during optimization, and has control over the degree of mutation and noise applied at any time (and thus the balance between exploration and exploitation of the space).

The Hill-Climber employs an elaborate breeding mechanism which provides diversity and novelty while also offering patches which resemble ones preferred by the musician, as shown in Figure 3. The breeder relies on three mechanisms: mutation, recombination, and opposite-recombination. These algorithms are discussed in detail in Luke (2019), but we may summarize them here. Mutation adds noise to every parameter in a patch individual. If the parameter is metric, the noise is added by uniformly selecting from a range centered on the parameter value sized according to the musician's chosen mutation weight. If the parameter is categorical then its value is randomized with a certain probability again chosen according to the mutation weight.

Recombination takes two parent individuals and produces a child individual as follows. For each parameter in the first parent, with some probability the parameter will deviate from the first parent's value. If the parameter is metric, the new value will be randomly selected from the range between the two parents. If the parameter is categorical, the new value will be, with 0.5 probability, set to the value of the second parent. Recombination is meant to "mix and match" features of two fit parents, ideally to produce yet fitter offspring.

Opposite-Recombination is a variant of Recombination meant to add diversity or act as an inertia procedure to push in the direction indicated by the musician's selections. For each parameter, it produces a value on the "other side" of the first parent from the second parent with some probability. If the parameter is metric, this is done by subtracting the first parent from the second. If the parameter is categorical, then the new value is set to the second parent unless they are the same, in which case it is set to some random different value.

Humans are a Problem The primary challenge in interactive evolution is the low number of individuals (patches) presented to the musician. It is common for an evolutionary optimization algorithm to require tens or hundreds of thousands of presentations before it has adequately optimized. This is not possible in interactive evolution, as the fitness function is a human, and humans are fickle, are easily distracted, and get bored quickly. It is not reasonable to expect a human to sit through more than a few hundred patch auditions before they give up. This difficulty is known as interactive evolution's *fitness bottleneck* problem (Biles 1994).

Because it has so few auditions available, the Hill-Climber must resort to tricks to maximize the value of each audition. The parameter space of patches is sparsely populated with "good" patches, and filled with garbage or silent ones, and the Hill-Climber must avoid these garbage patches. For example,

the Hill-Climber's careful delineation of metric and categorical parameters, with custom mutation and recombination operators for each, avoids jumping into garbage space caused by treating all parameters as metric (as is commonly done).

This is also a reason for the unusual breeding mechanism: it does not deviate too far from the patches selected, but still enforces diversity and can provide an inertia mechanism: if the musician has moved from a previous patch Z to a new patch A, perhaps they would prefer a patch even further in that direction (see $A-Z$, $B-Z$, and $C-Z$ in Figure 3).

A version of the Hill-Climber called the *Constrictor*, employs a different garbage-avoidance strategy: starting from N well-vetted patches, it allows the user to iteratively remove patches, replacing them with recombined versions of the remainder. The idea is that by staying in the middle of a cloud of well-vetted patches, we are less likely to find garbage.

However perhaps the most aggressive approach to avoiding "bad" patches is the Hill-Climber's new, optional variational autoencoder, discussed next.

Variational Autoencoding

An *autoencoder* (Hinton and Salakhutdinov 2006) is a feed-forward neural network that takes an incoming vector and must output exactly the same vector. However, in the middle of the neural network there is a narrow neck through which data must pass. For example, the autoencoder might input vectors of length 100, but in order to output them must pass their information through a space of size 45. Obviously this cannot achieve an identity function in general: but it may be able to achieve the identity function on a finite training set of vectors. To do this it would learn a smooth, 45-dimensional latent space (a manifold) which passes through all of the training set vectors in the higher (100) dimensional space. It would then map incoming vectors to this latent space (or *encode* them) and then unmap them back again on the other side of the narrow neck (or *decode* them).

We employ a version called a *variational autoencoder* or VAE (Kingma and Welling 2014), which learns a distribution over the latent space instead of a direct mapping into it. This is achieved by having the middle layer of the autoencoder encode a collection of parameters that describe this distribution. As Gaussian distributions are commonly used and well understood, what is learned in the model for our method is a collection of means and standard deviations which describe separate Gaussian functions for each dimension. While learning, the network is penalized for deviating from standard normal distributions (in order to avoid collapsing into degenerate zero-deviation distributions) by using a weighted Kullback-Leibler divergence. The process during training is to encode the vector into the parameters for the distribution, sample from the Gaussian distribution as the latent vector, then decode this sampled latent vector. The variance inherent in this process helps map similar regions of the latent space to similar patches: if multiple vectors near each other in the latent space are sampled, and both are supposed to decode to the same final vector — as often happens during training as we will input the same vector many times — the network will ideally learn something about *the region of the latent space surrounding those vectors* in the decoder, and not just the vectors themselves. This architecture, known as β-VAE (Higgins et al. 2017), ideally restricts similar sounding patches, when manipulated in the latent space, to map to smooth, reasonable, and nearby patches in the final space.

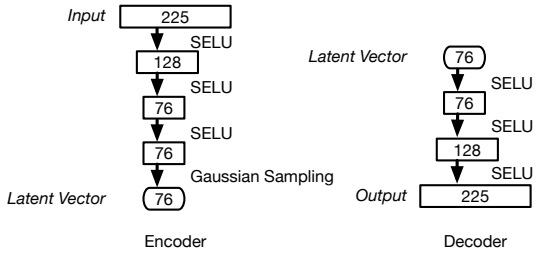

Figure 4: Encoder and Decoder Networks. Note that each block in the encoder and decoder describe their input size.

After training, the VAE is broken into the *encoder*, which maps the full space into the latent space, and the *decoder*, which does the opposite. The encoder's Gaussian sampling layer is then replaced with the identity function. We can then, for example, input a random vector to the Decoder, and it would output a vector along the manifold defined by the original samples.

Improving Patch Optimization We train a VAE on a large corpus of human-designed patches. After training, we then separate it into the encoder and decoder. We primarily use the decoder as follows. The Hill-Climber is no longer maintaining a population of 16 or 32 patches: rather it is maintaining a population of *vectors in the latent space*. To assess a vector, it decodes it to produce the patch, then auditions it. To seed the initial generation, it simply uses the encoder to encode patch seeds into latent-space vectors for the population.

How does this help us? We first train the autoencoder on a large corpus of open, human-designed patches, and so the latent space only passes through the parameter space in the vicinity of these patches. Thus arbitrary vectors on the latent space will generally map to vectors in the regions populated by "good patches", avoiding garbage. Unfortunately, there are few synthesizers with an online corpus of enough patches to successfully train an autoencoder. The Yamaha DX7 is one: we have successfully trained an autoencoder using nearly 27K unique patches. The Yamaha TX81Z synthesizer is possible target candidate for the future, with approximately 8K patches available. The DX7 has 145 parameters, some categorical, and so when one-hot-encoded it comes to 225 parameters. The trained latent space was 76 parameters.

The specific architecture used can be seen in Figure 4. The *SELU* is an activation function which behaves identically to the well known *Exponential Linear Unit*, but has chosen scaling parameters which cause the weights to self-normalize over many training iterations (Klambauer et al. 2017).

We also use the autoencoder for simple patch mutation: rather than mutate a patch directly by some amount, we encode the patch into its latent vector, mutate the vector by that amount, then decode the result into a new patch. The goal, once again, is to mutate the patch but keep it near "reasonable" patches even with significant mutation weights.

Is this Computationally Co-Creative?

It seems clear that the Hill-Climber is at least a *creative support tool* in the sense of Shneiderman et al. (2005): it supports exploration, is forgiving of error, has a low threshold to entry, and is capable of exploring any part of the space. But this is a very low bar: it's the same for many very rudimentary tools. Instead, we argue that this tool is in fact co-creative.

Karimi et al. (2018) define computational co-creativity as "interaction between at least one AI agent and at least one human where they take action based on the response of their partner and their own conceptualization of creativity during the co-creative task." We think that the Hill-Climber easily achieves this: it is taking action (proposing new patches) based on the response of the human, and the human is taking action (criticism) based on the proposals.

The Hill-Climber is an asymmetric collaboration: it is proposing new patches, and while the musician *can* propose patches to consider, they are primarily the fitness function or critic. Thus we may view the Hill-Climber as a DIFI (or Domain Individual Field Interaction) system (Feldman, Csikszentmihalyi, and Gardner 1994). From a DIFI perspective, the Hill-Climbing algorithm is the Individual, and the musician is the Field (and, if you like, the Domain).

However, a creative system must typically optimize for two criteria at once: novelty and some notion of value (Boden 1992; Wiggins 2006). What is the Hill-Climber really optimizing against? After all, the fitness function is being entirely determined by a human being. It's true that the system is emphasizing both diversity (if not novelty) and quality when breeding, but ultimately it ought to be considered co-creative only if the *human,* in collaboration, is also aiming for novelty and value when assessing fitness. We imagine that this is the case in many situations: but humans are fickle. The system as a whole is co-creative, in some sense, only if the human is doing their part.

Conclusion and Future Work

We presented a system which combines interactive evolution and a variational autoencoder to help explore the space of synthesizer patches. We think this back-and-forth qualifies it as a co-creative system: or certainly something rather more than just a creative support tool.

The biggest challenge in interactive evolution still remains the fitness bottleneck. To progress even faster, we'd need to allow supervisory feedback: that is, allowing "I'd like the sound brighter" or "more like a cello" rather than just "I like this one better". This would make it easier to argue in favor of co-creativity as well: as the musician would be able to contribute more to the system than mere criticism.

References

Biles, J. A. 1994. GenJam: a genetic algorithm for generating jazz solos. In *International Computer Music Conference*, 131–137.

Boden, M. 1992. *The Creative Mind*. Abacus.

Dahlstedt, P. 2001. A MutaSynth in parameter space: interactive composition through evolution. *Organized Sound* 6(2):121–124.

Eno, B. 1996. *A Year With Swollen Appendices: Brian Eno's Diary*. Faber & Faber.

Feldman, D. H.; Csikszentmihalyi, M.; and Gardner, H. 1994. *Changing the World: A Framework for the Study of Creativity*. Praeger.

Higgins, I.; Matthey, L.; Pal, A.; Burgess, C.; Glorot, X.; Botvinick, M.; Mohamed, S.; and Lerchner, A. 2017. β-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representations*.

Hinton, G. E., and Salakhutdinov, R. R. 2006. Reducing the dimensionality of data with neural networks. *Science* 303:504–507.

Horner, A.; Beauchamp, J.; and Haken, L. 1993. Musical tongues XVI: Genetic algorithms and their application to FM matching synthesis. *Computer Music Journal* 17(4):17–29.

Ianigro, S., and Bown, O. 2016. Plecto: a low-level interactive genetic algorithm for the evolution of audio. In *EvoMUSART*, 63–78.

Jónsson, B.; Hoover, A. K.; and Risi, S. 2015. Interactively evolving compositional sound synthesis networks. In *Genetic and Evolutionary Computation Conference*, 321–328.

Karimi, P.; Grace, K.; Maher, M. L.; and Davis, N. M. 2018. Evaluating creativity in computational co-creative systems. In *International Conference on Innovative Computing and Cloud Computing*.

Kingma, D. P., and Welling, M. 2014. Auto-encoding variational bayes. In *International Conference on Learning Representations*.

Klambauer, G.; Unterthiner, T.; Mayr, A.; and Hochreiter, S. 2017. Self-normalizing neural networks. In *Neural Information Processing Systems*, 972–981.

Luke, S. 2013. *Essentials of Metaheuristics*. Lulu, 2nd edition. http://cs.gmu.edu/∼sean/book/metaheuristics/.

Luke, S. 2019. Stochastic synthesizer patch exploration in Edisyn. In *EvoMUSART*.

McDermott, J.; O'Neill, M.; and Griffith, N. J. L. 2010. Interactive EC control of synthesized timbre. *Evolutionary Computation* 18(2):277–303.

Seago, A. 2013. A new interaction strategy for musical timbre design. In *Music and Human-Computer Interaction*. Springer. 153–169.

Shneiderman, B.; Fischer, G.; Czerwinski, M.; Myers, B.; and Resnick, M. 2005. *NSF Workshop Report on Creativity Support Tools*. National Science Foundation.

Suzuki, R.; Yamaguchi, S.; Cody, M. L.; Taylor, C. E.; and Arita, T. 2011. iSoundScape: Adaptive walk on a fitness soundscape. In *EvoApplications*, 404–413. Springer.

Takagi, H. 2001. Interactive evolutionary computation: fusion of the capabilities of EC optimization and human evaluation. *Proceedings of the IEEE* 89(9):1275–1296.

Wiggins, G. A. 2006. A preliminary framework for description, analysis and comparison of creative systems. *Knowledge-Based Systems* 19:449–458.