

研究報告

AIの言語生成モデルを応用したライブコーディングの検討

A Survey of the LiveCoding with AI Language Model

奥田 知希

Tomoki OKUDA

九州大学大学院芸術工学府

Graduate School of Design, Kyushu University

城 一裕

Kazuhiro JO

九州大学芸術工学研究院

Faculty of Design, Kyushu University

概要

本研究では、GPT-3 に代表される AI の言語モデルを用いたコード生成技術の、ライブコーディングパフォーマンスへの応用を検討する。大規模なテキストデータを学習した GPT-3 やその後継モデルである Codex を用いることで、話し言葉のようなテキストからプログラムのソースコードを生成することができる。まず、GPT-3 を用いてコードを生成するサービス「AI Programmer」によるコード生成、次に Codex を用いてコードを生成する「GitHub Copilot」によるコード生成を調査した。その実践から、よりライブコーディングのパフォーマンスに適したコードの生成を行う方法を考察する。さらに、今後 AI のソースコード生成の精度を高めていくことで可能になる新しい表現方法についての展望を述べる。

1. 背景

人工知能 (Artificial Intelligence, AI) の社会実装が進む中で、芸術・デザイン分野での AI の活用が模索されている (徳井, 2021)。2022 年にはオープンソースの text-to-image モデルである Stable Diffusion が公開された (CompVis, 2022)。テキストを打ち込むだけで誰でも無償で簡単に、高品質な画像を生成できることから、世界中で瞬く間に利用が広がった。Stable Diffusion が大きな話題となったことで、現在盛り上がりを見せている画像生成の分野に続き、AI を用いた音声合成の分野も発展しつつある。Meta とヘブライ大学はテキストによる説明から音声を生成する AI モデル AudioGen を発表した (Kreuk et al., 2022)。この AI モデルを用いることで、「風が吹く中で口笛を吹く音」や「大勢の歓声の中で話す男性の声」といったテキストから、それらしい音を生成することができる。また、音楽を生成する AI モデルとして知られるのが、2020 年に OpenAI が発表した Jukebox (Dhariwal et al., 2020) である。歌手の名前とジャンル、歌詞を入力するだけでそれらしい

曲が生成され、時間と GPU さえあれば、歌手とジャンルを変えて無限に音楽を生成できる。その一方で、1 分の音楽を生成するのに 5 時間以上の計算が必要になるなど、リアルタイム性とは程遠いのが現状である。Jukebox は非常に高価な GPU を数百台何週間にもわたって動かして作られたことが論文内で明かされており (Qosmo, 2022)、そうしたリソースを持たない一般の研究者、ましてやアーティストにとって、こうしたアーキテクチャの実用性は低いと言える。そのため、現在行われているほとんどの音楽生成を目標とした AI 研究は直接生のオーディオを生成するのではなく、MIDI などの音楽を表すシンボルの生成を目的にしている。とはいえ、今後 AI による音楽生成分野の研究が進むと、Stable Diffusion のようにテキストを用いて行う音楽生成で、Jukebox に匹敵する高品質な音楽をリアルタイムで生み出せる AI モデルがいつかは登場するだろう。そうやって実現される、リアルタイムでのテキストの説明による音楽の生成を、今ある AI モデルやアプリケーションを組み合わせることで行えないだろうか考えた。そこで本稿で検討する組み合わせがライブコーディングと AI の言語生成モデルである。

1.1. ライブコーディング

ライブコーディングとは、プログラムを動作させた状態で即興的にコーディングを行う行為自体を表現形態とする、一種のパフォーミングアートであり (田所, 2018)、プログラムのソースコードを用いて音楽や映像を生成する。オーディオを生成するライブコーディングのための言語としては、python をベースとした言語 FoxDot、Ruby をベースとした言語 SonicPi などがあるが、本稿では Haskell というプログラミング言語のライブラリとして実装される TidalCycles (McLean, 2014) を用いたライブコーディングについて取り上げる。TidalCycles は、ライブコーディングの創始者の 1 人である Alex McLean を中心に開発された、ライブコーディング言

語である。テキストによって音楽的なパターンを生成したり、反復を記述したりして、それを組み合わせることで単純なパーツの相互作用による複雑な音楽の生成を可能とする。TidalCycles 自体には音を生成する機能はなく、音声合成エンジン SuperCollider に対して OCS(Open Sound Control) というネットワークコマンドを送ることで音を鳴らす。

1.2. GPT-3

AI の自然言語処理技術は近年飛躍的に向上しており、中でも 2020 年に OpenAI が発表した Transformer ベースの言語生成モデル GPT-3 は画期的なアイデアとして広く普及した (Brown et al., 2020)。この言語生成モデルは 45 テラバイトもの膨大なテキストデータを学習させて作られたもので、まるで人間が書いたかのような自然な文章を高精度で作り出すことができる。すでに多くの文章自動生成ツールやコード生成ツールに利用されており、その 1 つが、2022 年 7 月にサービスを開始した、AI ライティングサービス AI Programmer である (ASReal, 2022)。このサービスは GPT-3 の API を用いており、プログラミング言語の種類を選択し、どんなプログラムを書くかを日本語で入力することで、ソースコードを自動生成することができる。

1.3. Codex

OpenAI は 2021 年、GPT-3 をベースとしてプログラムのソースコードと自然言語の変換に特化した、言語生成モデル Codex を発表した (Chenn et al., 2021)。公開 GitHub リポジトリから数十億行のコードを収集して GPT-3 をトレーニングし、Codex モデルを開発した。Codex は Python の処理を最も得意としているが、JavaScript、Ruby、Haskell などを含む 12 以上の言語に対応している。GitHub Copilot はこの Codex を用いて開発された、コードの自動補完システムであり、関数名やコメントを入力するだけで、それに続くコードを自動生成できる。

2. 既存のサービスを使ったコード生成の実践

先述した 2 つの言語生成モデルを用いたコード生成ツールを用いて、ライブコーディング言語のソースコードを生成できるかどうか検証した。

2.1. AI Programmer によるコード生成

まず、AI Programmer を用いて、TidalCycles のコードを生成する。AI Programmer は言語を指定し、書かせたいコードの処理内容を日本語で記述することでコー

ドを自動生成する。言語を Haskell に設定し、処理内容を「TidalCycles を用いて、d1 に 4 つ打ちのバスドラムと、裏拍のハイハットによって構成されるドラムのリズムを生成する。」とすると、以下のコードが生成された。(図 1)

このコードは bd がバスドラムを、hh がハイハットを示している。”””で囲まれた部分が 1 小節を表し、最初の 2 拍でバスドラムを 4 回、後半の 2 拍でハイハットを 4 回再生するという意味のコードとなっている。指示通りに 4 つ打ちのドラムと裏拍でハイハットを鳴らすには、以下のようなコードになる。

AI Programmer は処理内容のテキストから、バスドラムとハイハットを 4 回ずつ鳴らすということを踏まえたコードを生成しているが、裏拍のような音楽の専門用語を理解してコードを書くことはできていない。

言語を選択し、AI Programmerに書かせたいコードの処理内容を入力してください。

Haskell

TidalCyclesを用いて、d1に4つ打ちのバスドラムと、裏拍のハイハットによって構成されるシンプルなドラムのリズムを生成する。

d1 \$ sound "bd bd bd bd hh hh hh hh"

コードを生成

クリア

図 1: AI Programmer を用いた TidalCycles のコード生成の例 1

次に、「TidalCycles を用いて、d1 でシンセサイザーの supersaw で Cmaj7 のコードを鳴らし、リバーブをかける。」という処理内容でコードを生成し、以下の出力を得た。(図 2)

このコードは文法的に間違っており、Cmaj のコードを指定するときには #note “c'maj7” のように表記する必要がある。また、リバーブのエフェクトを表す #room は 0.1 の値をとってリバーブの量が変わるので、正しくは以下のようなコードとなる。

```
d1 $ sound "supersaw" #note "c'maj7" #room 0.5
```

このように、AI Programmer を用いてコードを生成すると、指定した楽器の音やエフェクトなどはコードの記述に確実に含まれるものの、文法的な間違いが多く、リズムパターンの表記もうまくいかなかった。

2.2. GitHub Copilot によるコード生成

次に、GitHub Copilot を用いて TidalCycles のコードを生成する。GitHub Copilot はコメントアウトで書か

言語を選択し、AI Programmerに書かせたい

コードの処理内容を入力してください。

Haskell

TidalCyclesを用いて、d1でシンセサイザーのsupersawでCmaj7のコードを鳴らし、リバーブをかける。

d1 \$ sound "supersaw:Cmaj7" # room

コードを生成

クリア

図 2: AI Programmer を用いた TidalCycles のコード生成の例 2

せたいコードの処理内容を記述することで、コードを自動生成する。2.1と同様に、「d1に4つ打ちのバスドラムと、裏拍のハイハットによって構成されるドラムのリズムを生成する。」という指示でコードを生成すると、以下のコードが得られた。(図3)

このコードは最初の2拍でバスドラムを4回、後半の2拍でハイハットを3回再生するという意味のコードとなっている。AI Programmerと同様に、コメントで指示した内容通りの出力は得られなかったものの、バスドラムとハイハットを用いたリズムのコードの出力が得られた。また、同じ音の繰り返しを意味する「*」記号やユークリッドの互除法をリズムパターンの生成に応用したアルゴリズム、ユークリッド・シーケンスを表す()を用いた表記など、AI Programmerと比べてより実践的なコードが得られた。

次に、「d1でシンセサイザーのsupersawでCmaj7のコードを鳴らし、リバーブをかける。」という処理内容に対しては、以下のコードが得られた。(図3)

リバーブをかけるという指示に対して、# room "0.5"というコードを生成しており、値を取るエフェクトに対しては具体的な数値を指示していなくても勝手に適当な値を入れている。また、# s "midi" や # gain "0.5"などの指示を出していないコードまで生成されており、文法的なミスで実行できないコードになってしまっている。

```
test.tidal
1  --d1に4つ打ちのバスドラムと、裏拍のハイハットによって構成されるドラムのリズムを生成する。
2  d1 $ sound "bd*4 hh(3,8)"
3  --d1でシンセサイザーのsupersawでCmaj7のコードを鳴らし、リバーブをかける。
4  d1 $ sound "supersaw" # s "midi" # n "c:maj7" # gain "0.5" # room "0.5"
```

図 3: GitHub Copilot を用いた TidalCycles のコード生成の例

2.3. 考察

AI Programmer と GitHub Copilot の両方とも、TidalCycles のベースとなっている言語 Haskell のコードを学習させているので、TidalCycles のコードを生成させても基本的な文法は概ね正しく生成される。両方とも TidalCycles のコードはほとんど学習させていないと推測され、複雑なリズムや実用的なリズムが生成できない。GitHub Copilot は以前自分が記述したコードの影響を受けやすく、コメントアウトで記述した指示に忠実なコードが得られにくい。2.2で生成された# s "midi" や # gain "0.5"などの指示を出していないコードは、以前筆者が別のファイルで記述したコードであり、コメントアウトとコードの例文を増やしても生成されるコードの精度は上がらなかった。GPT-3の学習データに含まれる TidalCycles のコードの量を増やすことでよりコード生成の質は向上すると考えられる。

3. TIDALCYCLES の生成に特化した GPT-3

次に、TidalCycles のコードによる小規模なデータセットを用意して学習を行い、TidalCycles のコードの生成に特化した GPT-3 モデルを作成する。そのモデルを用いて日本語のテキストからソースコードを生成し、生成されるコードの精度を確認する。

3.1. finetuning

finetuning は既存の学習済みモデルをユーザーが用意した学習データでトレーニングし、特定のタスクに合わせて調整することである。GPT-3 は finetuning 無しで様々な用途の言語生成タスクに用いることができるが、finetuning を行うことで特定のタスクに特化し、より高い精度で言語を生成することができる。

3.2. 準備

学習データとして、bpm を指定し、ドラムとシンセサイザーの演奏を記述したソースコードと、それを説明する日本語のテキストの組み合わせを 406 組用意した。GPT-3 には Davinci, Curie, Babbage, Ada の 4 つのモデルがあるが、その中でも最も能力が高く、他のモデルが実行できるすべてのタスクを実行できる Davinci に対して finetuning を行った。

3.3. 結果

finetuning を行った GPT-3 に対して、「d1に4つ打ちのバスドラムと、裏拍のハイハットによって構成さ

れるドラムのリズムを生成する。」というテキストからコードを生成し、以下のコードが得られた。(図4)

setcps(60/60/4) は bpm を設定するコード、\$ sometimesBy 0.2 (jux (0.125 ~)) はランダムなタイミングでリズムパターンに変化を加えるコード、#shape 0.5 は音圧を上げるコードで、s “clap(1,8,3)” は3拍目にクラップを鳴らすコードである。意図していない記述が多く含まれているが、4つ打ちのバスドラムを示す s “bd(4,8,[0,3])” と裏拍のハイハットを示す s “808oh(4,8,1)” のコードは正しく生成された。

```
import openai
prompt="d1に4つ打ちのバスドラムと、裏拍のハイハットによって構成されるドラムのリズムを生成する。->"
response = openai.Completion.create(
  engine="davinci:ft-okudada-2022-12-06-13-10-12",
  prompt=prompt,
  max_tokens=100,
  stop="\n")
text2=response['choices'][0]['text']
text2_1=text2.replace("\n","\n")
print(text2_1)

setcps (60/60/4)

d1
$ sometimesBy 0.2 (jux (0.125 ~))
$ stack[
  $ bd(4, 8, [0, 3]),
  $ "808oh(4,8,1)",
  $ "clap(1,8,3)"
]
# shape 0.5
```

図4: finetuning した自作のモデルを用いたコード生成の例1

次に、「bpm180で、sine波でEsus4とEmin9のコード進行と、ドラムベースの激しいドラムを鳴らす。」というテキストから以下のコードを生成した。(図5)

指示通りにbpm180でブレイクビーツを鳴らすコードとなっているが、“clap mc125”という部分は存在しないサンプルを呼び出すコードとなっており、エラーとなってしまふ。また、テキストではEsus4とEmin9を使ったコード進行を指示したが、生成されたコードには反映されていない。

3.4. 考察

自分で finetuning を行った GPT-3 モデルの学習データは、コードに対して大雑把な日本語の説明を用意していたため、コードの生成を行う際には、日本語のテキストには含まれないコードまで多く生成された。これは実際にライブコーディングパフォーマンスを行う際に意外性のあるコードが生成されることを期待して、あえて大雑把な日本語のテキストを用意したためであった。しかし、ライブコーディングのスタイルによっては思った通りのコードが得られた方がいい場合もあるので、日本語のテキストの書き方については見直す余地があると考えられる。そして、生成されるコードは文法的なミスを含むことがあるので、学習データの量はさらに増やす必要がある。また、用意した学習データは1人が書いた数十曲分のコードを406曲まで傘増

```
import openai
prompt="bpm 180で、sine波でEsus4とEmin9のコード進行と、ドラムベースの激しいドラムを鳴らす。->"
response = openai.Completion.create(
  engine="davinci:ft-okudada-2022-12-06-13-10-12",
  prompt=prompt,
  max_tokens=400,
  stop="\n")
text2=response['choices'][0]['text']
text2_1=text2.replace("\n","\n")
print(text2_1)

setcps(180/60/4)

d1
$ stack[
  loopAt 1
  $ slice 8 "0 <2 0 3> 2 <3 0> <412 0> <1 5> <6 1 6> <6 7 0>"
  $ s "breaks 165"
  --, s "gabba"4" # gain 0.8 # lpf 200
  --, s "clap mc125" # gain 0.5 # pan (sine) # hpf 1000
]
# shape 0.5
# crush 4
# release(0.4+0.6*rand)
# lpf (range 1000 20000 $ slow 8 $ sine) # resonance 0.3

d3
$ slow 2
$ jux rev
$ (off (1/16) (!+ up "{0 3 0 4}%4"))
$ arp "<up>"
$ s "sine0"
>| n "<0 7 -9 12>/4"
# tremolrate (sine)
# room 0.4
# sz 0.8
# pan (sine)
# gain 0.9
```

図5: finetuning した自作のモデルを用いたコード生成の例2

したものであったため、コードを記述した人の癖が大きく反映されたものとなってしまった。GitHubに共有されているオープンソースのTidalCyclesのコードなど、より多様な学習データを集める必要がある。

4. おわりに

本稿では、AIの言語生成モデルを用いた2つのコード生成ツールとGPT-3を finetuning した自作のモデルを用いて、ライブコーディング言語TidalCyclesのコード生成を試みた。その結果、既存のコード生成ツールよりも自作のAIモデルの方が精度の高い生成結果が得られた。

ライブコーディングパフォーマンスのスタイルは大きく分けて2つある。1つは白紙のエディターから始めて、その場で1から全てのコードを書くスタイル。もう1つは事前に用意したコードのフラグメントをDJのようにその場で組み合わせていくスタイルである。後者のようなスタイルのライブコーダーのことをコードジョッキー(CJ)と呼ぶことがある(久保田, 2020)。CJは基本的に事前に自分で作った曲を繋ぎ合わせるが、DJは自分の曲をかける人、他人の曲だけをプレイする人がいて、むしろ後者の方が多いかもしれない。ライブコーダーにおいても、他人が作った曲だけをプレイするCJのスタイルがあってもいいのではないだろうか。本稿で検証したAIの言語生成モデルを用いることで、自分が作っていない曲を繋いで演奏する、新しいコードジョッキーのスタイルが実現できる。その演

奏の面白さを担保するためには、今後学習データを増やして、より多様な生成結果を得ることが望まれる。

5. 謝辞

本研究の一部は、日本学術振興会科研費 [21H03768] ならびに [19K21615] の支援を受け実施された。

6. 参考文献

- 徳井直生. 2021. 『創るための AI: 機械と創造性のはてしない物語』 ビー・エヌ・エヌ.
- CompVis. 2022. “stable-diffusion”GitHub Pages. <https://github.com/CompVis/stable-diffusion>
- Felix Kreuk et al. 2022. “Audiogen: Textually guided audio generation,” arXiv preprint arXiv:2209.15352.
- Dhariwal, P. et al. 202. Jukebox: A generative model for music. arXiv preprint arXiv:2005.00341.
- Qosmo. 2022. 「音楽生成 AI の現状と可能性 (2022 年版)」 <https://qosmo.jp/publication/musicai-whitepaper/>
- 田所淳. 2018. 『演奏するプログラミング、ライブコーディングの思想と実践』 ビー・エヌ・エヌ.
- Alex McLean. 2014. "Making programming languages to dance to: live coding with tidal". In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. pp.63–70.
- Tom B Brown et al. 2020. "Language Models are Few-Shot Learners", In *Proceedings of NeurIPS*, pp.1877-1901.
- 有限会社 ASReal 2022. "AI Programmer". <https://aiprogrammer.hashlab.jp/> (2022-12-06 参照)
- Mark Chenn et al. 2021. "Evaluating Large Language Models Trained on Code". arXiv preprint arXiv:2107.03374.
- 久保田晃弘. 2020. 「スペシフィック・コード」 <https://ekrits.jp/2020/01/3201/> (2022-12-06 参照)

7. 著者プロフィール

奥田知希 (Tomoki OKUDA)

2022年に九州大学工学部を卒業後、九州大学芸術工学府音響設計コースに進学。学部時代からサンプラーを用いた楽曲制作を開始し、現在は AI を用いた音楽表現について研究している。



この作品は、クリエイティブ・コモンズの表示 - 非営利 - 改変禁止 4.0 国際 ライセンスで提供されています。ライセンスの写しをご覧になるには、<http://creativecommons.org/licenses/by-nc-nd/4.0/> をご覧くださいか、Creative Commons, PO Box 1866, Mountain View, CA 94042, USA までお手紙をお送りください。