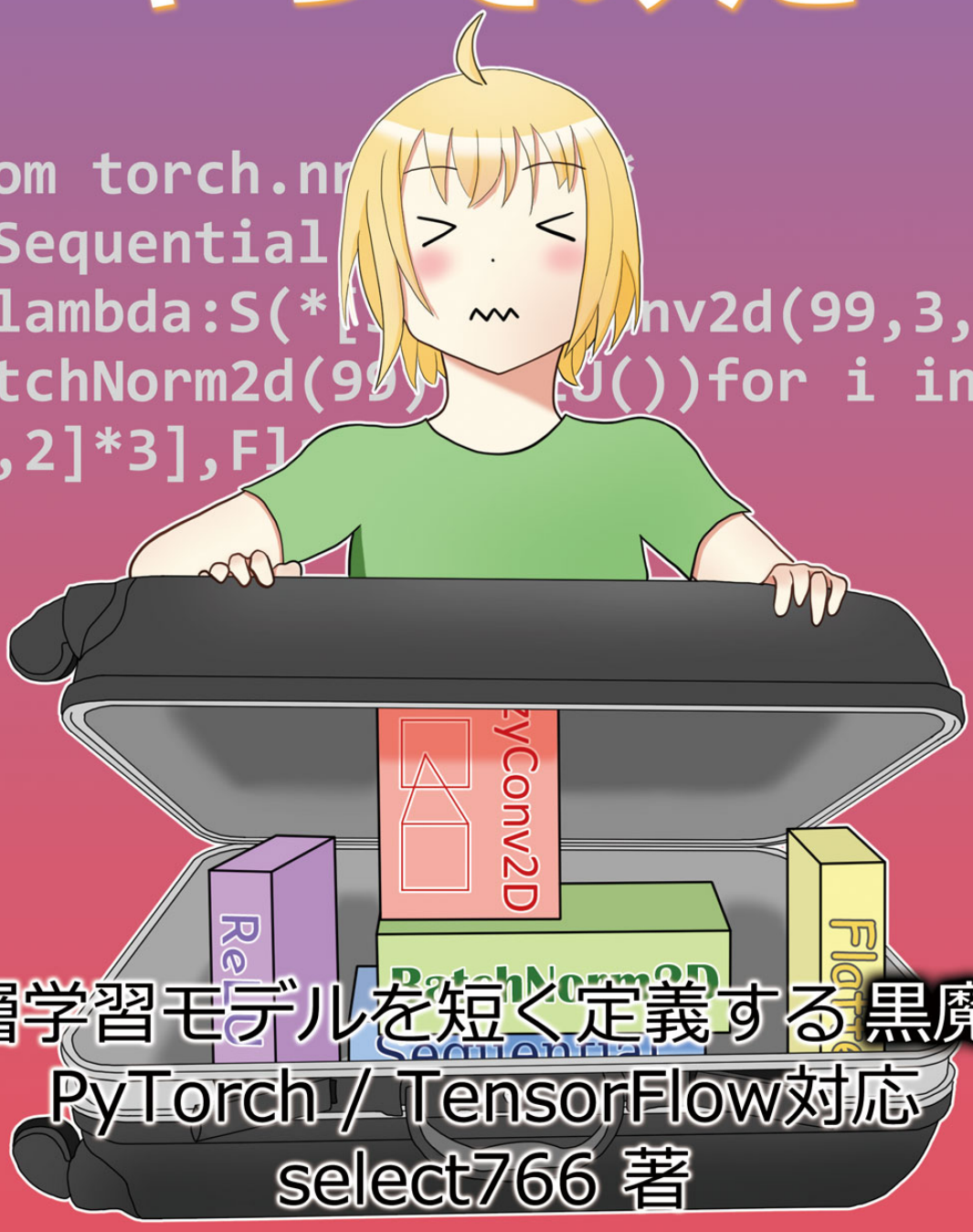
# Deep Learning Code Golf やってみた

```
from torch.nn
S=Sequential
m=lambda:S(*[
BatchNorm2d(99)
[1,2]*3],F]
```




深層学習モデルを短く定義する黒魔術

PyTorch / TensorFlow対応

select766 著

# まえがき

Deep Learning Code Golf は、筆者が作成した言葉で、「深層学習のモデル定義をできるだけ短いソースコードで表現する」というゲームです。様々なアルゴリズムを短いソースコードで表現する「コードゴルフ」という遊びがプログラマの間で従来から行われてきましたが、それを深層学習に応用したものとなります。本書は主に 2 つのパートからなります。1 つ目は、Deep Learning Code Golf という新しい遊びのルール化についてです。2 つ目は、そのルールに従って短いソースコードを書くテクニックの解説です。最終的に開発したコードは、リスト 1 のようになります。どうしてもライブラリの呼び出しが中心となるため、通常のコードゴルフほど奇怪なコードは記述できませんでしたが、深層学習に特化した黒魔術をお楽しみいただければ幸いです。

▼リスト 1　Deep Learning Code Golf の解答例

```
from torch.nn import*
S=Sequential
m=lambda:S(*[S(LazyConv2d(99,3,i),BatchNorm2d(99),ReLU())for i in[1,2]*3],
→Flatten())
```

本書では以下の知識を前提としており、発展的な事項のみを解説しておりますのでご了承ください。

- Python の基本文法
- 深層学習の基礎

## 免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、運用の結果について、筆者はいかなる責任も負いません。

# 目次

# 第 1 章

# イントロダクション

## 1.1 Deep Learning Code Golf とは

Deep Learning Code Golf は、筆者が作成した言葉で、「深層学習のモデル定義をできるだけ短いソースコードで表現する」というゲームです。このゲームを考えたきっかけは、深層学習の新しいモデル構造 ConvMixer を提案する論文[*1]で、手法の簡潔さを主張するためのおまけとして 280 文字以下で定義できるという参考コード（リスト 1.1）が Twitter 上で話題となっていたためです。論文自体はコードの長さ自体を短くすることを目標としているわけではありませんが、本書の筆者はコードを短くすること自体をゲームにしたらどうなるか興味を持ったので考察することにしました。

▼リスト 1.1 ConvMixer のコード（→は紙面上折り返しているが、コード上は前の行から継続していることを示す）

```
def ConvMixr(h,d,k,p,n):
 S,C,A=Sequential,Conv2d,lambda x:S(x,GELU(),BatchNorm2d(h))
 R=type('',(S,),{'forward':lambda s,x:s[0](x)+x})
 return S(A(C(3,h,p,p)),*[S(R(A(C(h,h,k,groups=h,padding=k//2))),A(C(h,h,1))) for
→ i in range(d)],AdaptiveAvgPool2d((1,1)),Flatten(),Linear(h,n))
```

Deep Learning Code Golf（以下 DLCG）の元となったのは、以前より知られているコードゴルフ（Code Golf）というゲームです。コードゴルフは、ショートコードとも呼ばれます。コードゴルフは、特定の課題を解くプログラムを、できるだけ短いソースコードで表現するゲームです。コードゴルフという名称は、スポーツのゴルフのように短い打数（文字数）でゴールすることからつけられたようです。コードゴルフで解く課題の例と

[*1] Patches Are All You Need? （執筆時点では査読中の論文で、匿名）
https://openreview.net/forum?id=TVHS5Y4dNvM

して有名なのは、FizzBuzz 問題です。FizzBuzz 問題とは、「1 から 100 までの数を出力せよ。ただし、3 の倍数の場合は Fizz、5 の倍数の場合は Buzz、3 の倍数かつ 5 の倍数の場合は FizzBuzz と出力せよ」という問題です。この問題を Python 言語で普通に解くコードは、リスト 1.2 のようになります。もちろん通常のプログラミングではこれで正解なのですが、コードゴルフでは、このコードをできるだけ短くする（コード自体の文字数を少なくする）ことを目指します。例えば Python ではインデントが必須であるもののスペース 1 個で十分で、% などの演算子の前後のスペースも不要ですので、元のコードと実行結果を変えずにリスト 1.3 のように書き換えることができます。これにより、コードが 196 文字から 143 文字に減少します。文法レベルの書き換えだけでなく、「3 の倍数かつ 5 の倍数」を「15 の倍数」と言い換えてロジックを組みなおせばリスト 1.4 のように少し短くなります。さらに多くのテクニックを用いると、リスト 1.5 のようなコードになります。ここまでくると、理解困難なコードとなるのが分かるでしょう。通常のプログラミングで重視される、コードの理解しやすさや変更のしやすさ、場合によっては実行速度を無視して、コードが短いということそれ自体を追求するのがコードゴルフの目的です。コードゴルフはプログラミング言語の仕様を最大限活用するゲームですので、言語によって使えるテクニックや実現できるコードの文字数が変わってきます。本書では、Python 言語（バージョン 3 系）を使用します。

従来のコードゴルフのほとんどでは、入力が決まれば出力が一意に定まるような課題が扱われてきました。アルゴリズムが違ったとしても、結果自体が変化しないことが条件となります。しかし本書で扱う DLCG では、若干異なります。深層学習は機械学習の一種で、画像などを入力し、その中に含まれる文字や物体を認識するというモデルを生成することが課題です。分類正解率が 100% であれば一意の出力となりますが、通常は 100% にはならず、ある程度の誤りが含まれます。多数の入力画像に対して統計的に正解率が一定以上であればよいと考え、具体的に各入力画像に対してどのような出力をするかは一意に定まらないのが普通です。正解率を決める要因は課題自体の難しさのほか、モデルの構造や学習時の手法などに影響されます。モデルの構造に自由度を持たせることが DLCG のユニークな点で、「正解率**% 以上のモデルを記述せよ」という課題になります。あまりに単純なモデルだと表現力が足りず正解率が低くなるという傾向がありますので、一定の正解率を担保しつつ、モデルの構造を簡略化したりコードレベルの実装方法を短くしたりすることに戦略性が生まれます。

▼リスト 1.2　FizzBuzz の標準的な実装

```
for i in range(1, 101):
    if i % 3 == 0 and i % 5 == 0:
        print("FizzBuzz")
    elif i % 3 == 0:
        print("Fizz")
    elif i % 5 == 0:
        print("Buzz")
    else:
        print(i)
```

▼リスト 1.3　不要なスペースを削除

```
for i in range(1,101):
 if i%3==0 and i%5==0:
  print("FizzBuzz")
 elif i%3==0:
  print("Fizz")
 elif i%5==0:
  print("Buzz")
 else:
  print(i)
```

▼リスト 1.4　ロジックを改良

```
for i in range(1,101):
 if i%15==0:
  print("FizzBuzz")
 elif i%3==0:
  print("Fizz")
 elif i%5==0:
  print("Buzz")
 else:
  print(i)
```

▼リスト 1.5　FizzBuzz の最短コード（59bytes）[*2]

```
for i in range(100):print(i%3//2*"Fizz"+i%5//4*"Buzz"or~i)
```

[*2] https://qiita.com/ymg_aq/items/b8e5d26035180bc8797e#python3-59-bytes

## 1.2 ルール設計と実行環境

Deep Learning Code Golf をゲームとして遊ぶにあたり、課題やその達成度を測る指標が必要です。そのためのルールを考えました。

### 1.2.1 深層学習を用いて解くべき課題の設定

深層学習を用いて解く課題としては、画像を入力とし、10 種類の物体のうちどれが写っているかを認識して分類する画像分類問題を扱うこととしました。より具体的には、CIFAR-10 データセット[*3]を用います。画像は 32 ピクセル四方のフルカラー（RGB の 3 チャンネル）で、分類すべき物体は 10 種類（飛行機、自動車、鳥…）です。学習用画像が 5 万枚、テスト用画像が 1 万枚あります。課題設定として重要なのは、要求されるモデルの複雑さと計算コストです。画像分類の入門課題としてよく用いられる MNIST データセットは簡単すぎて、極めて単純な構造のモデルでも正解率 99% が得られてしまい、モデル定義の工夫によって差がつきません。逆に、実用に近い課題として用いられる ImageNet データセットは複雑すぎて、学習初期の段階では全く正解できず、学習を数時間進めないと差がつきません。解き方によって短時間で差がつくという基準をもとにデータセットを選定しました。性質の異なる課題としては、ニュース記事（英単語の列）を入力して記事のジャンルのいずれに該当するかに分類するようなシーケンス処理問題なども考えられます。課題によって適切なモデルの構造が大きく異なるため、面白い課題を考えてみるのもよいでしょう。

次に、課題自体とは別に、ゲームとして成立させるための諸条件を考察します。

### 1.2.2 プレイヤーが記述するコードの範囲

深層学習を実行するには、モデルの定義だけでなく、学習データの前処理機構、オプティマイザなどいくつかの機構が必須となります。いずれの機構も正解率に影響を与えますし、様々な書き方が考えられます。ゲームとして注力すべき部分がブレないように、プレイヤーが記述すべき範囲と、課題として固定された機構を与える範囲を分ける必要があります。そのため、プレイヤーが記述するのは、モデル定義を行う関数 `m` だけとし、この関数を呼び出すとモデルオブジェクトを返すものとします。

コードの長さは、文字数で計測しました。アルファベットでも漢字でも 1 文字にカウントします。改行も 1 文字と数えます。コードの先頭・末尾の改行文字は無視します。

[*3] Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

### 1.2.3 実行時間

プレイヤーが書いたコードで生成されるモデルが所定の正解率を達成できるかどうか評価するには、実際に学習を行う必要があります。深層学習は計算コストが高く、高性能なGPU を搭載した計算機で何日もかかる場合もあります。計算時間がかかりすぎるとゲームとして遊べないため、数分以内に完了することが望ましいと考えられます。後述する実行環境において、合理的な複雑さのモデルが 1 分程度で学習できるようなデータセット及び学習ループ回数（エポック数）を目指しました。

これを満たす制約として、5 エポック（学習用画像全部をモデルに投入して、モデルパラメータを更新するサイクルを 5 回行う）学習させるという条件としました。ただし、モデルの複雑さやパラメータ数自体に制約を設けていないので、極端に複雑なものを定義すればいくらでも計算時間が長くなり得ます。現実的には、表現力の高い複雑なモデルは短いエポック数ではあまり正解率が高くならないことが多く、有力な解答にはなりづらいと考えられます。

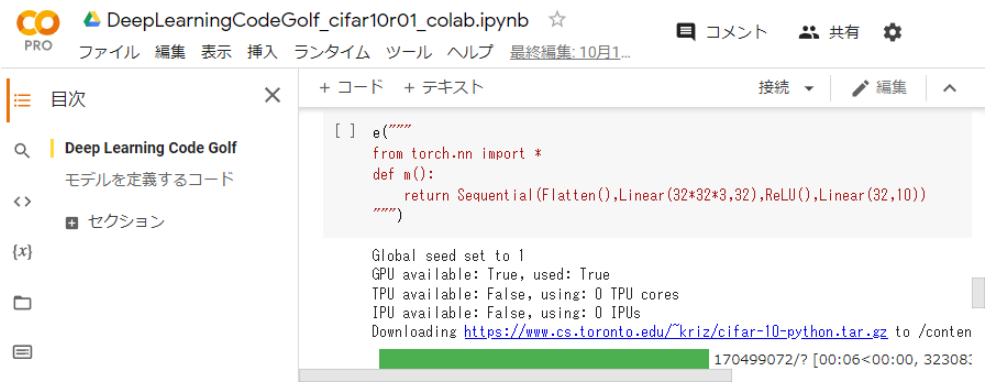
### 1.2.4 テスト用データに対する最適化

機械学習では本来、テスト用のデータは未知のデータであり、テスト用データでの正解率を最適化の指標として使うべきではありません。モデルの学習へフィードバックするための評価用データとテスト用データは分かれているのが正当です。しかしながらゲームとしてこれらの指標を明確に分離するのは難しいため、テスト用データでの正解率が高まるようにモデルのハイパーパラメータを最適化しても構わないものとします。

### 1.2.5 実行環境

環境構築および起動に手間がかかるとゲームとして遊びにくいです。本書では、深層学習向けに作られたクラウド上のインタラクティブな Python 実行環境である Google Colab を活用することとしました。Python の標準ライブラリだけで深層学習を行うのは事実上不可能なので、深層学習ライブラリとして PyTorch を用いました。Google Colab では初期状態でインストール済みです。さらに、学習ループの記述を簡略化するため PyTorch Lightning および Lightning Bolts を用いました。ただしこれらのライブラリの使用によってプレイヤーの書くべきコードに変化はありません。モデル定義のコードを文字列として評価用の関数に与えると、その文字数と学習結果得られたモデルの正解率を算出するような環境を整備しました。なお、深層学習ライブラリとして Tensorflow を用いた場合の検討を付録に掲載しています。実行環境のスクリーンショットを図 1.1 に示しま

す。e 関数に文字列としてモデル定義コードを与えると、学習を行ったうえで正解率と、コードの長さを表示してくれます。




▲図 1.1　Google Colab 環境でのコード記述例

### 1.2.6　ルールのまとめ

結論として、以下のようなルールとしました。

- 以下の条件設定で学習させた際に、テスト用データでの正解率 X% 以上となるコードを記述せよ。
  - X=10,20,...
- プレイヤーが記述するコードの条件
  - 引数のない関数 m を定義する。実行されると PyTorch のモジュールオブジェクトを返す。
  - m の内部で必要なモジュールは、コード内でインポートする。
- 学習条件
  - m() が返したモジュールオブジェクトに対し、評価システム側で学習用データによる学習を行なったのち、テスト用データにより正解率を計算し出力する。
  - データセット: CIFAR-10
  - データオーグメンテーション: 4px パディングしてランダムクロップ、ランダム左右反転（注：ゲームとしてはあまり重要ではないが、CIFAR10 を用いたサンプルコードの記述を継承）
  - オプティマイザ: `Adam(lr=1e-3)`

– バッチサイズ: 256
– エポック数: 5

### 1.2.7 再現性について

深層学習では、モデルパラメータを初期化する乱数や、モデルに与えるデータの順序などにより学習結果に違いが出ます。これらの数学的な違い以外にも、GPU 上での並列計算の実行順序に伴うハードウェアレベルの非決定的な挙動があり、完全な再現は難しいのが実情です。本書の実行環境ではリスト 1.6 により乱数の固定を試みました。同じハードウェア・ソフトウェアのバージョンで複数回実行したところ同じ結果が得られるようでした。一方で環境が違う場合までは保証できないため、賞金が出るようなコンペを実施するには難しい点となります。

▼リスト 1.6　再現性を担保する設定

```
pytorch_lightning.seed_everything(1, True)
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False
torch.use_deterministic_algorithms(True)
```

本書執筆時点でのライブラリのバージョンを表 1.1 に示します。第 2 章で掲載するコード例では実験的機能も利用しているため、将来のバージョンでは同じコードが使えない可能性があります。

▼表 1.1　ライブラリのバージョン

| | |
|---|---|
| Python | 3.7.12 |
| torch | 1.10.0+cu111 |
| torchvision | 0.11.1+cu111 |
| pytorch_lightning | 1.5.1 |
| pl_bolts （lightning-bolts） | 0.4.0 |
| tensorflow（付録で使用） | 2.7.0 |

実行環境の Colab notebook は `https://colab.research.google.com/drive/1KGZFdERNGVAofSCEBnEMfeRENOpWcCUA?usp=sharing` にて配布しています。また、書いたコードを投稿できるサイトを試作しました。`https://dlcodegolf.web.app/` からアクセスできます。

# 第 2 章

# コード解説

本章では、筆者が実際に DLCG に取り組んで記述したコードと、そこで用いたテクニックを解説します。

## 2.1 ベースライン

▼リスト 2.1　ベースラインコード: 43.41%, 104 文字

```
from torch.nn import *
def m():
    return Sequential(Flatten(),Linear(32*32*3,32),ReLU(),Linear(32,10))
```

まずはコードを短くするテクニックを用いる前の、最も基本となるコードをリスト 2.1 に示します。これは、全結合層 2 層からなるモデルです。本書のルールでは、コード中で関数 m を定義し、これを実行すると、ニューラルネットワークの実行順序を定めるモデル構造及び学習されるパラメータを包含するモジュールオブジェクト（torch.nn.Module のサブクラス）が生成されるように実装する必要があります。PyTorch でモジュールオブジェクトを生成するには、torch.nn 以下に定義されているクラスを使います。*でインポートを行うと何がインポートされたのか分かりづらいため通常の開発ではあまり用いませんが、毎回 torch.nn.Sequential のような記述をすると長くなるため、基本テクニックとして使用します。本書の範囲では、これ以外のインポートは行いません。Sequential は、引数で指定したレイヤー（本書ではモジュールオブジェクトとほぼ同じ意味で使います）全てを包含し、先頭から順に実行するようなレイヤーです。枝分かれのないモデルを定義するのに有用です。枝分かれがある場合も含めた一般的なモデル定義の方法はリスト 2.2 のようになりますが、明らかにコードが長くなります。モデルへの入力テ

ンソルは、32 × 32 ピクセルの RGB 画像なので [バッチサイズ，3，32，32] となっています。Linear（全結合層）レイヤーは 2 次元のテンソルしか受け付けないため、Flatten を用いて形状を [バッチサイズ，3*32*32] に変換します。Linear は、Linear(入力チャンネル数,出力チャンネル数) という引数を取ります。ここでは 3*32*32（=3072）チャンネル入力、32 チャンネル出力のレイヤーを生成します。次に活性化関数として ReLU を挿入します。PyTorch では、活性化関数は Linear に含まれていませんので別途挿入する必要があります。最後に、32 チャンネル入力、（10 クラス分類なので）10 チャンネル出力の Linear を挿入しています。

▼リスト 2.2　Sequential を使わないモデル定義の例

```
from torch.nn import *
class M(Module):
    def __init__(self):
        super().__init__()
        self.conv1 = Conv2d(3, 32, 3)
        self.relu1 = ReLU()
        self.conv2 = Conv2d(32, 64, 3)
    def forward(self, x):
        h = self.conv1(x)
        h = self.relu1(h)
        return self.conv2(h)
def m():
    return M()
```

## 2.2　正解率 30% 以上: 最短のコード

▼リスト 2.3　最短のコード: 32.65%, 67 文字

```
from torch.nn import*
m=lambda:Sequential(Flatten(),LazyLinear(10))
```

まずはできるだけモデルを単純化し、正解率を問わない、最短と思われるコードをリスト 2.3 に示します。モデルの構造は、全結合 1 層のみの線形モデルです。学習させてみたところ、正解率 32.65% で文字数 67 でした。まず、`import *`のスペースは不要で、`import*`とすることが可能です。これで 1 文字削減できます。改行を; に置換可能な場合がありますが、改行文字を 1 文字として数えているため効果はありません。ルール上、モジュールオブジェクトを返す関数 m を定義する必要がありますが、これを関数定義文 `def m():\n xxx` とする代わりにラムダ式を使用して `m=lambda:xxx` とすることができます。

1 文字の削減となります。ただし、ラムダ「式」のため、関数内で代入文を使えなくなります。そして、Linear(3072,10) の代わりに LazyLinear(10) を用います。LazyLinear は入力チャンネル数の記載を省略可能とする、実験的機能です。文字数としては、3027, の 5 文字が減って Lazy の 4 文字が増えます。合計で 1 文字の削減となります。もし入力チャンネル数が 3 桁（999 以下）ならこれで削減することはできないということになります。なお、Flatten は省略すると Linear に 4 次元のテンソルが入力されて実行時エラーとなりますので削減できません。何か抜け道がないか、気になるところです。

## 2.3 正解率 40% 以上: 複数レイヤーの導入

▼リスト 2.4 複数レイヤーコード: 44.43%, 83 文字

```
from torch.nn import*
L=LazyLinear
m=lambda:Sequential(Flatten(),L(99),ELU(),L(10))
```

次は、正解率 40% 以上の部門で最短のコードをリスト 2.4 に示します。線形モデルでは流石に精度が低いため、2 層以上のモデルが必要となります。中間層 99 チャンネルの 2 層モデルで 40% 以上を達成することができました。LazyLinear を 2 回使いたいので、変数 L に代入します。LazyLinearLazyLinear の 20 文字の代わりに、L=LazyLinear\nLL の 15 文字となり 5 文字の削減となります。2 つの全結合層の間には非線形の活性化関数が必要です。もし活性化関数がないと、2 連続の行列積となりますが、これは 1 つの行列積と等価となり線形モデルと変わりません。深層学習が最初に流行した際の活性化関数は ReLU($ReLU(x) = \max(x, 0)$) でしたが、さまざまな亜種が提案されています。そのうち PyTorch で文字数最短は ELU です。これは、$ELU(x) = x$ (if $x > 0$), $\exp(x) - 1$(otherwise) で表されます。中間層のチャンネル数が 99 なのは、2 文字でできるだけ大きなパラメータ数を得るためです。普通は 32、256 など 2 の冪乗を使いますが、DLCG 特有の制約で珍しい数値が出現しています。

## 2.4 正解率 50% 以上: 畳み込みの導入

▼リスト 2.5 畳み込みコード: 52.76%, 88 文字

```
from torch.nn import*
m=lambda:Sequential(Conv2d(3,9,3),ReLU(),Flatten(),LazyLinear(10))
```

正解率 50% 以上の部門で最短のコードをリスト 2.5 に示します。線形モデル 2 層では実現が困難なため、ここで畳み込み層を導入しました。画像に対する畳み込みは Conv2d クラスで実現されます。引数は、Conv2d(入力チャンネル,出力チャンネル,カーネル幅,ストライド=1,パディング=0,...) となります。ストライド以降の引数は省略可能です。正解率 50% 以上という条件であれば、畳み込み層の出力チャンネル数は 9 チャンネルで実現可能でした。60% 以上とするにはチャンネル数 2 桁が必要でした。その場合のコード例をリスト 2.6 に示します。1 文字の増加で精度が 8 ポイント改善するというのもコードゴルフ的には奇妙な感じですが、ハイパーパラメータの性質上仕方ありません。活性化関数では ELU が最短ではあるのですが、正解率が低いという問題があり、ReLU を使用しています。ELU では入力値が 0 付近の場合にほぼ恒等関数となるため、長い学習時間をとって非線形性が出るような領域まで使われるようにする必要があるのかもしれません。

▼リスト 2.6　畳み込みコードチャンネル数 99: 60.78%, 89 文字

```
from torch.nn import*
m=lambda:Sequential(Conv2d(3,99,3),ReLU(),Flatten(),LazyLinear(10))
```

## 2.5　正解率 70% 以上: ループの導入

▼リスト 2.7　ループのあるコード: 74.69%, 119 文字（→は紙面上折り返しているが、コード上は前の行から継続していることを示す）

```
from torch.nn import*
S=Sequential
m=lambda:S(*[S(LazyConv2d(99,3,i),BatchNorm2d(99),ReLU())for i in[1,2]*3],
→Flatten())
```

最後に、正解率 70% 以上の部門です。コードをリスト 2.7 に示します。2 層のモデルでは不十分なため、3 層以上のモデルをできるだけ短いコードで定義します。まずは文法を解説します。複雑になっているため、1 つずつ解きほぐしていきます。まず、Sequential が 2 回使われるため、変数に代入しています。[f(i) for i in X] という構文ですが、リスト内包表記です。X にはリスト等の列挙可能なオブジェクトを与えます。例えば、[f(i) for i in [1,3,5]] は、[f(1), f(3), f(5)] という記述と等価です。関数 f に対して異なる引数を与えて得た結果を、リストとして取得することができます。[

1,2]*3 は、リストに対する掛け算で、繰り返しを意味します。すなわち、[1,2]*3==[1,2,1,2,1,2] です。リスト内包表記では、通常 for の前、in の後ろにはスペースが必要ですが、変数名として使えない文字と連続する場合はスペースを省略してもエラーとなりません。これにより、...)for i in[... というようにスペースを 2 文字分削減できます。次に、リストの前の*ですが、リストの要素を関数の引数として展開する役割です。例えば、f(*[1,3,5]) は f(1,3,5) と等価となります。Sequential は、可変長の引数としてレイヤーを受け取るため、この構文を使用してリスト内包表記で動的生成したレイヤーの列を与えます。これらの文法を展開した結果はリスト 2.9 のようになります。結局のところ、畳み込み 6 層のモデルということになります。

次に、深層学習のモデルとしてのテクニックを解説します。ここでは新たに 2 つのクラス LazyConv2d と BatchNorm2d が登場しています。BatchNorm2d はバッチ正規化層で、入力の各チャンネルについて、ミニバッチ内の値の平均を 0、分散を 1 となるようアフィン変換することで学習を促進するものです。本書のルールではこれがないと学習イテレーション数に対する正解率の向上速度が低く、所定イテレーション数内に正解率 70% を達成できませんでした。文字数が長いですが採用せざるを得ません。チャンネル数指定不要の LazyBatchNorm2d もありますが、このコードでは文字数削減に寄与しません。LazyConv2d は、入力チャンネル数を省略できる畳み込み層です。畳み込み層の入力チャンネル数は 3 桁以下なので、Lazy よりもチャンネル数を明示的に記述して 999, とするほうが短いのですが、入力チャンネル数が最初の層では 3 なのに対し他の層では 99 となり、条件分岐が必要となります。ループ変数 j が仮に 0,1,2,3... となる場合、[3,99][j>0] のように場合分けを短く記述することはできなくはないですが、Lazy を用いる方がより短いです。最後に LazyConv2d の引数ですが、LazyConv2d(出力チャンネル,カーネル幅,ストライド) となります。ストライド指定の別の実装方法として、j%2+1 for j in range(6) も考えられますが、j for j in[1,2]*3 の方が短いです。ストライドが 1 と 2 に交互に設定されている点が重要です。ストライドは、出力のピクセルが 1 つ移動するたびに入力に対する畳み込みの窓の移動距離を表すものです。畳み込み層の画像の出力サイズの計算式は (入力ピクセル幅 $+ 2 \times$ パディング $-$ カーネル幅)/ストライド $+ 1$ のようになります。ストライドを $1 \to 2 \to 1 \to 2 \to 1 \to 2$ の順にすることで、画像サイズが $32 \to 30 \to 14 \to 12 \to 5 \to 3 \to 1$ と変化し、ちょうどピクセル数が 1 になるところがポイントです。最後の畳み込み層の出力テンソルの形状は [バッチサイズ, 99, 1, 1] となります。これを Flatten に入力すると、[バッチサイズ, 99] の出力が得られます。出力は 10 クラス分類なので [バッチサイズ, 10] となるのが正しいのですが、学習のためのコードはリスト 2.8 のようになっており、実は過剰なチャンネルがあってもエラーなく動きます。11 個目以降のクラスは学習データに出現しないので、負の無限大を出力するように学習が進んでいくと考えられます。逆にクラスが 10 個あるのに 9 チャンネル以下

の出力を与えると実行時エラーになりますので、精度不問のコードで 10 チャンネルの出力のところを 9 チャンネルに減らしてコードを短くすることには使えません。画像サイズが 1 になることは重要で、もしサイズが 2 となって `[バッチサイズ，99，2，2]` という出力となると、これを `Flatten` に与えた結果が `[バッチサイズ，396]` となります。チャンネル数が過剰という点だけでは画像サイズが 1 ピクセルの場合と変わりませんが、`[バッチサイズ，0，0，0]` がクラス 0 に対応、`[バッチサイズ，0，0，1]` がクラス 1 に対応ということになります。しかしこれはチャンネル方向ではなく空間方向の別ピクセルが別のクラスに対応するという正しくない構造となり、うまく学習できません。画像サイズをうまく 1 ピクセルになるように調節することにより、最後の全結合層を省略することができています。また、通常のモデルでは最後の畳み込み層や全結合層には活性化関数を用いませんが、リスト内包表記の都合で最終層だけ場合分けすることができません。これでも学習が進むので、DLCG 特有のテクニックとして使うことができます。簡潔さの裏に、深層学習初心者には決してお勧めできない黒魔術が入っておりますのでご注意ください。

なお、チャンネル数を 2 桁の 99 より増加させると、リスト 2.10 のように若干正解率が向上します。一方、他のモデル構造も検討しましたが、今回のルールで正解率 80% 以上を実現するモデルは残念ながら見つかりませんでした。分岐があるモデル構造として Residual 構造などを試しましたが、エポック数を 5 に固定した状態では正解率が高くなりませんでした。数十分かけて学習を進めれば本書で掲載したモデルでも 80% 以上の正解率は得られますし、より複雑なモデルで 90% 以上の正解率を目指すことも可能ではあります[*1]。しかし細かいテクニックを重ねてコードを短くするゲームとして遊ぶには計算時間がかかりすぎますので、ここまでとしました。筆者が開発したコードは以上となります。深層学習という課題上、ライブラリに定義されたクラス名の記述が避けられず、また大量のクラスを複雑に組み合わせることが最適解とはならなかったため、コードを見れば何をしているかはわかりやすい解答となりました。DLCG に興味を持ってくださった方が、従来のコードゴルフのような、コードを見ても何をやってるかわからないような奇怪なコードが最適解となるような面白い課題設定を開発してくれることを期待しております。

[*1] 99% 以上の正解率となるモデルが紹介されています。 https://paperswithcode.com/sota/image-classification-on-cifar-10

▼リスト 2.8　損失を計算するコード（self.model がプレイヤーが記述した関数で生成したモデル）

```
x, y = batch
logits = F.log_softmax(self.model(x), dim=1)
loss = F.nll_loss(logits, y)
```

▼リスト 2.9　文法上短縮したものを展開したバージョン

```
from torch.nn import*
def m():
    return Sequential(
        Sequential(LazyConv2d(99,3,1),BatchNorm2d(99),ReLU()),
        Sequential(LazyConv2d(99,3,2),BatchNorm2d(99),ReLU()),
        Sequential(LazyConv2d(99,3,1),BatchNorm2d(99),ReLU()),
        Sequential(LazyConv2d(99,3,2),BatchNorm2d(99),ReLU()),
        Sequential(LazyConv2d(99,3,1),BatchNorm2d(99),ReLU()),
        Sequential(LazyConv2d(99,3,2),BatchNorm2d(99),ReLU()),
        Flatten()
    )
```

▼リスト 2.10　ループのあるコード（チャンネル数 256）: 75.16%, 121 文字

```
from torch.nn import*
S=Sequential
m=lambda:S(*[S(LazyConv2d(256,3,i),BatchNorm2d(256),ReLU())for i in[1,2]*3],
→Flatten())
```

## 2.6　ConvMixer のテクニック

今回筆者が定義したルールに対してはモデルが複雑すぎて活用できませんでしたが、DLCG のアイデア元となった ConvMixer のコードを短くすることに使われたテクニックについて、筆者なりに解釈して解説します。

▼リスト 2.11　ConvMixer のコード（再掲）

```
def ConvMixr(h,d,k,p,n):
 S,C,A=Sequential,Conv2d,lambda x:S(x,GELU(),BatchNorm2d(h))
 R=type('',(S,),{'forward':lambda s,x:s[0](x)+x})
 return S(A(C(3,h,p,p)),*[S(R(A(C(h,h,k,groups=h,padding=k//2))),A(C(h,h,1))) for
→ i in range(d)],AdaptiveAvgPool2d((1,1)),Flatten(),Linear(h,n))
```

A=lambda x:S(x,GELU(),BatchNorm2d(h)) は、ラムダ式定義です。引数 x には例えば Conv2d() が入ります。S=Sequential なので、x に続いて、GELU()、BatchNorm2d() が実行されるようなモジュールオブジェクトが生成されることになります。つまり、畳み込みなどの処理に活性化関数を追加する作用を持ちます。

次の行は見慣れない構文が用いられています。R=type('',(S,),{'forward':lambda s,x:s[0](x)+x}) は、Residual 構造を作るためのクラス定義となります。組み込み関数 type は、引数が 1 つのときと 3 つ以上のときで全く役割が異なります。今回は、type(name, bases, dict, **kwds) という引数になり、クラス定義を動的に実行します。name はクラス名ですが空文字列でもエラーになりません。bases は、基底クラスをタプルで指定します。ここでは、S=Sequential が指定されています。dict は、クラスのメソッドを定義します。ここでは、forward メソッドをラムダ式の形で与えています。lambda s,x:s[0](x)+x を解説します。引数 s は、普通 self と書かれるもので、クラスのインスタンスを指します。x が通常の引数で、テンソルを受け取ります。テンソル x をあるレイヤーに与えた出力 s[0](x) と、元々の x を足したものを返すことで Residual 構造を実現します。s[0] はどこから来るのでしょうか。Sequential クラスは、コンストラクタの引数を、配列のインデックス指定のように取り出すことができます。例えば、m=Sequential(Conv2d(),ReLU()) と定義したとき、m[0]==Conv2d()、m[1]==ReLU() のようになります。クラス R は Sequential を継承したものであり、コンストラクタは上書きしていないため、R(Conv2d()) というコンストラクタ呼び出しに対して、そのメソッド内では self[0] で Conv2d() を取り出すことができます。結局、forward(x) メソッドでは、クラス R のコンストラクタに与えたレイヤーオブジェクトに x を与えて呼び出した結果と、x を足した結果を返すという処理が行われることになります。

最後の行は前の章で解説したテクニックと同じで、リスト内包表記による深いモデルの定義が行われています。なお、) for i in range(d)] は、)for i in[0]*d] と短縮できます。また、AdaptiveAvgPool2d((1,1)) は AdaptiveAvgPool2d(1) に短縮できます。

筆者はコードゴルフのために、通常あり得ない構造のモデルを定義しました。一方でこの論文のように、あくまで実用性のあるモデルを短く書き換えるという遊びも考えられますので、気が向いたら考えてみてはいかがでしょうか。

# 付録 A

# TensorFlow への移植

PyTorch と並び著名な深層学習フレームワークとして、TensorFlow があります。筆者は PyTorch を使うことがほとんどですが、TensorFlow でも DLCG を行うとどんな違いがあるか検証しました。

結論から言えば、PyTorch を利用する場合と解答となるモデルそのものは変わりませんでした。畳み込み層のほうが全結合層より短く書けるというような変化はないようです。

## A.1　環境構築

環境構築は PyTorch より簡単です。学習ループを自前で書いたりライブラリを入れなくても `model.fit()` を呼ぶだけで自動的にループを回してくれます。そのため、コードの長さの計測機能を除けば、リスト A.1 だけで学習・評価ができます。データオーグメンテーションがないなど、PyTorch と若干違うため同じモデル構造でも正解率に若干の差があります。

▼リスト A.1　TensorFlow での学習コード

```
import tensorflow as tf
tf.random.set_seed(1) # 乱数固定
# データセット読み込み
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
# モデル構築
model = m() # プレイヤーが定義するモデル
# 学習設定
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
model.compile(optimizer='adam',
              loss=loss_fn,
              metrics=['accuracy'])
model.fit(x_train, y_train, epochs=5, batch_size=256)
```

```
# 正解率の評価
test_loss, test_acc = model.evaluate(x_test, y_test)
```

## A.2 PyTorch との相違点

実際のコードを示しながら、PyTorch との違いを解説します。

最初に、TensorFlow での通常のモデル定義コードをリスト A.2 に示します。Sequential や Flatten は PyTorch とほぼ同じです。ただし、Sequential は可変長引数ではなくレイヤーのリストを引数にとります。PyTorch での Linear は Dense に変わり、入力チャンネル数は不要となります。すなわち Dense は PyTorch の LazyLinear 相当です。もちろん、ResNet のような分岐があるモデルを定義するための、クラスを用いたモデル定義方法もありますが明らかにコード量が増大します（A.3 節で触れます）。

画像データのデータ配置が、PyTorch では [バッチサイズ，チャンネル，幅，高さ] なのに対し、TensorFlow ではデフォルトでは [バッチサイズ，幅，高さ，チャンネル] となっています。この違いは学習済みモデルを移植したり Reshape を行ったりする場合に影響が出るものの、本書の範囲では影響ありませんでした。

▼リスト A.2　一般的なモデル定義

```
import tensorflow as tf
def m():
    return tf.keras.models.Sequential([
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(128, activation='relu'),
        tf.keras.layers.Dense(10)
    ])
```

正解率を問わない最短のコードをリスト A.3 に示します。PyTorch では Sequential と Flatten は同じモジュール（torch.nn）からインポートできたのですが、TensorFlow では、tf.keras.models.Sequential と tf.keras.layers.Flatten というように別々のモジュールからインポートする必要があります。ここで、tf.keras.models.Sequential は tf.keras.Sequential としてもインポートできるため、from tensorflow.keras import*で Sequential をインポートしつつ、同時に layers に tf.keras.layers がインポートされるようにするのが最短と考えれます。インポートの煩雑さが影響して、コードの長さは PyTorch の 67 文字から 85 文字に増加してしまいました。

▼リスト A.3　最短のコード: 38.08%, 85 文字

```
from tensorflow.keras import*
L=layers
m=lambda:Sequential([L.Flatten(),L.Dense(10)])
```

次に、複数の全結合層を利用するコードをリスト A.4 に示します。TensorFlow での活性化関数は、Dense へのパラメータ引数を用いて Dense(99,activation='elu') と記述することが可能ですが、それよりも単体の ELU クラスを用いるほうが短くなります。

▼リスト A.4　複数レイヤーコード: 43.41%, 103 文字

```
from tensorflow.keras import*
L=layers
D=L.Dense
m=lambda:Sequential([L.Flatten(),D(99),L.ELU(),D(10)])
```

畳み込みを用いたコードをリスト A.5、リスト A.6 に示します。畳み込みは Conv2D で、引数は Conv2D(filters, kernel_size, strides=(1,1), ...) です。入力チャンネル数は不要です。データオーグメンテーションの違い等により（過学習気味)、チャンネル数 99 では正解率 60% に達しなかったためチャンネル数が 3 桁になっています。

▼リスト A.5　畳み込みコード: 51.93%, 108 文字

```
from tensorflow.keras import*
L=layers
m=lambda:Sequential([L.Conv2D(9,3),L.Flatten(),L.ReLU(),L.Dense(10)])
```

▼リスト A.6　畳み込みコードチャンネル数 512: 60.22%, 110 文字

```
from tensorflow.keras import*
L=layers
m=lambda:Sequential([L.Conv2D(512,3),L.Flatten(),L.ReLU(),L.Dense(10)])
```

最後に、ループを導入した深いモデルをリスト A.7 に示します。PyTorch と同様、出力チャンネル数が 10 より大きくてもエラーになりません。チャンネル数 68 は、正解率 70% 以上となるようパラメータの探索を行った結果です。Sequential の引数は可変長引数ではなくリストなので、リストの結合を用いて最後の Flatten を付加しています。

▼リスト A.7　ループのあるコード: 70.26%, 148 文字

```
from tensorflow.keras import*
L=layers
S=Sequential
m=lambda:S([S([L.Conv2D(68,3,i),L.BatchNormalization(),L.ReLU()])for i in[1,2]*3]
→+[L.Flatten()])
```

## A.3　Residual 構造を記述する

PyTorch で書かれた ConvMixer の解説で登場した、Residual 構造 ($y = f(x) + x$) を TensorFlow でも短く書いてみます。TensorFlow で分岐があるモデルを記述する方法は 2 つあり、Functional API による方法と、サブクラス化による方法と呼ばれます。

まず、Functional API でのモデル記述例をリスト A.8 に示します。Functional API では、モデルの入力を表す変数を `Input` で生成し、これをレイヤーオブジェクトに渡してその出力を表す変数を順次生成していくことでモデルの構造を表します。最後に、モデル全体の入出力となる変数を `Model` 関数に渡すことで、モデルが完成します。変数は複数回使用できるので、分岐を表現することができます。`l=ReLU(k)+k` と記述すれば、非常にシンプルに Residual 構造が表現できます。Residual レイヤーを定義するのではなく、モデル定義文の中で 2 回同じ変数を使うことで表現します。レイヤー同士のつながりを表す変数を明示的に扱えることがシンプルさの要因です。ただし、変数を明示的に代入して取り回さないといけないため代入文が必須で、リスト内包表記（＝式）で深いモデルを表現することが難しくなります。DLCG 的には使いづらいでしょう。

▼リスト A.8　Functional API でのモデル記述例

```
from tensorflow.keras import *
from tensorflow.keras.layers import *
def m():
    i=Input([32,32,3])
    j=Flatten()(i)
    k=Dense(128)(j)
    l=ReLU()(k)
    m=Dense(10)(l)
    return Model(i,m)
```

最後に最も柔軟性の高いモデル定義方法であるサブクラス化による方法を示します。リスト A.9 に Residual レイヤーの記述例を示します。PyTorch のときと考え方は同じで、`Sequential` クラスを継承し、レイヤー実行時に呼び出される `call` メソッドをオーバー

ライドして Residual 構造を実装します。PyTorch における forward を call に、self[0] を self.layers[0] に置換した形となります。

▼リスト A.9　サブクラス化での Residual 記述例

```
from tensorflow.keras import *
from tensorflow.keras.layers import *
class Residual(Sequential):
    def call(self, inputs):
        return self.layers[0](inputs) + inputs

# 使用例
r = Residual([ReLU()])
x = tf.constant([1.0, -1.0])
r(x)
# => [ 2., -1.]
```

DLCG らしく、このコードを短く記述した結果をリスト A.10 に示します。type 関数を利用するのは PyTorch と同じなのですが、TensorFlow では余分な記述が必要でした。第 1 引数'x'ですが、空文字列にすると内部のコードでエラーが発生します。クラス名の 1 文字目が'_'かどうかで挙動を変える機構があり、1 文字目がないため IndexError が生じます。ラムダ式の引数には、キーワード引数を受け取る**d が必要でした。モデルの実行が学習中か推論中かを示す training キーワード引数（Dropout 等の挙動が変わる）が渡されるため、これを受け取らないとエラーになります。しかし、素直な定義であるリスト A.9 ではキーワード引数を定義しなくてもエラーになりません。この理由は、TensorFlow 内部の実装がメソッドが受け取る引数を列挙し、定義に合わせて適切な引数を渡すように実装されているためです。しかしながらラムダ式でメソッドを定義することが想定されていないのか、判定が間違ってしまい{'call':lambda s,x:s.layers[0](x)+x}という定義に対して training キーワード引数を与えてしまいエラーになっています。

▼リスト A.10　短縮されたサブクラス化での Residual 記述例

```
from tensorflow.keras import *
from tensorflow.keras.layers import *
R=type('x',(Sequential,),{'call':lambda s,x,**d:s.layers[0](x)+x})
```

以上のように、TensorFlow でも PyTorch とほぼ同じように DLCG を遊ぶことができることがわかりました。"tensorflow.keras"が"torch.nn"より長いことや、気を利かせて挙動を変える機構の誤判定等で、PyTorch よりコードが長くなってしまう傾向があることがわかりました。

# あとがき

当サークル初の、とても軽い読み物を刊行してみました。深層学習を実用目的でやる以外にも、知識を使っていろんな遊び方ができることを示したいと思って書きました。メインシリーズであるポケモンバトル AI 本が長らく出せておらず、技術書典 12 が開催されるということで軽いテーマで一冊本を書いてみようというモチベーションからテーマ選定を行いました。11 月には電子書籍用の原稿ができていたのですが、技術書典の「後から印刷」（会期後に注文があった冊数だけ物理本を印刷できる）の大幅割引キャンペーンが告知されたので急遽物理本も出すことにしました。元の原稿が印刷可能なページ数の下限に達しておらず、内容を増やす必要に迫られ思いのほか忙しくなってしまいました。

表紙のイラストは、ヤマブキちゃん（仮）が旅行帰りにお土産がスーツケースに入りきらなくて困っているというシチュエーションを描きました。深層学習のモデル定義を無理やり圧縮するという本書のテーマを表現しています。ヤマブキちゃんというのはヤマブキ計算所のオリジナルキャラで、筆者を代弁させる役割を期待しています。キャラクターデザインとかさっぱりわからないのですが…

私がイラストを描き始めたのは 2021 年に入ってからで、下手の横好きながら楽しんでいます。以下のようなきっかけがあって、自作の同人誌に自作のイラストを載せるに至りました。

- 200X 年 漫画「ひだまりスケッチ」の沙英というキャラクターに「美術を学んでいるのは自作小説に自分で挿絵を描くため」という設定があり、なんとなく気に留める
- 2018 年 初の同人誌を刊行。表紙は Word で適当に作ったが、周囲のサークルの本を見ていると綺麗な表紙が多くてうらやましくなる
- 2020 年 9 月発売の某恋愛アドベンチャーゲームで、神絵師であるヒロインを手伝って同人誌即売会に出るというシナリオを見て、イラストが描きたくなる
- 2021 年 親戚の子供からの年賀状に自作の漫画絵が描かれており、謎の対抗心を燃やし、イラストの勉強に着手

# Deep Learning Code Golf やってみた

2022 年 1 月 22 日　技術書典 12 版 v1.0.0
2022 年 1 月 23 日　内容修正 (「再現性について」を改良) v1.0.1

著　者　select766
発行所　ヤマブキ計算所

# ヤマブキ計算所