数字を学習させて数字を作る

Letting machines learn numbers to make up numbers.

鍛島 康裕
KAJIMA Yasuhiro

多くの機械学習の教科書には，MNIST を用いた数字判別が最初の練習問題として使われている．そして GAN(Generative Adversarial Network) においても，多くの本では数字を生成することから始め，最終的には人の顔などを生成する，というパターンで書かれている．しかしこれらの間には本質的な違いがある．人間の顔は二つの顔の中間的な顔というものが多くの場合ありうるが，数字にはありえない．例えば１と８の中間的（画像として）数字は存在しない．であるにも拘らず，自分の知る限りでは数字の場合も顔の生成と同様に一様分布の乱数を用いて画像を生成しようとしている．自分にはこれがいわゆる mode collapse の原因ではないかと思われた．そこでここでは，少し方法を変えて実験してみた．練習のような内容であるが，のちに何らかの発展があることを期待したい．

1.1 はじめに

MNIST とは，”THE MNIST DATABASE of handwritten digits” のことで，画像認識の実験のためによく使われる手書き数字のデータベースである．

MNIST Dataset の一部

このような手書き数字が 60,000 枚ある（これに加えて test set 10,000 枚，手書き数字画像認識の精度を判定するために用いられている）．元のデータは binary であるようで[1)]，少し取り扱いが面倒であるが，すぐ使えるようになったものが簡単に手に入る．３層程度の CNN (Convolutional Neural Network) を用いた機械学習で，99% 以上の精度を出すことができる[2)]．この Network で正解できなかった画像は，人間の目で見ても判別がつきにくいものがある．その例として，以前自分が電場の効率的計算方法である FMM(Fast Multipole Method) をさらに高速にしようとしていた時，それを画像認識に応用できるのではないかと考えて実験した時の画像を見てみる[3)]．自分がやったことなので簡単に説明すると，その方法は CNN と類似の新しい方法となるかと期待したものである．FMM が電場の情報を集約して扱っているため，遠い近いといった情報を連続的に扱っているが，それを応用すればピクセル間の画像の移動を連続的に捉えられるので，新しい方法となり得るのではないかと思ったのであった．その時，うまく判別できなかったものの例を挙げると次のようになる．

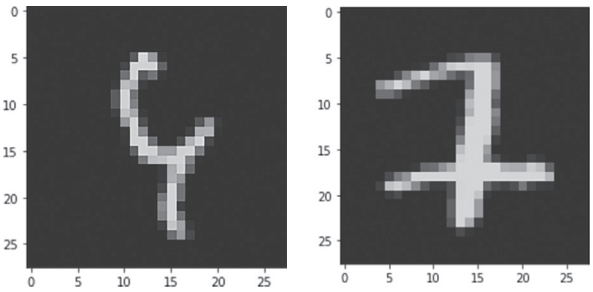
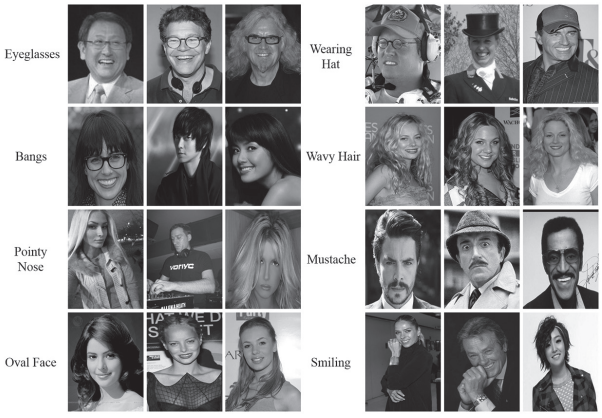


Figure1. 正解できなかった画像の一部
左の正解は９だが，コンピュータは４と判断した．右の正解は７だが，コンピュータは２と判断した． 左に関しては間違えても仕方ないようにも思える．MNIST にはこのような汚い手書き文字が含まれている．

なお，この方法では残念ながら精度は 98% くらいまでしか到達できなかった．

機械学習関係では，手書き数字以外にもこのようなデータベースがたくさん用意されている．例えば CelebA dataset (CelebFaces Attribute dataset) と呼ばれるものがあり，



CelebA Dataset の一部（実在の人物）

このように写真とともに属性 (attribute) が（別ファイルで）セットになっている．何と２０万人以上の有名人の顔が入っている．サミーデービス Jr とか，クルーゾー警部，その下はアランドロンでしょうか？

この画像を使って様々なことが試みられているが，その中でも GAN は最も面白いと思われる．GAN とは Generative Adversarial Network の略で[4]，敵対的生成ネットワークなどと訳されるもので，Turing 賞を受賞した Yann LeCun によって *"the most interesting idea in the last 10 years in MachineLearning"* と言われているそうである．Generative Adversarial Network という言葉では意味が分かりにくいが，この方法によって人の顔を作ったり，作曲したりなど様々なことをすることができる．下に転載したこの顔は NVIDIA の研究者たちによって作られたものである[5]．

GAN によって作られた顔[5]

よく見ると変なところもあるが（耳が左右異なるとか，髪や背景が変など），実在の人物にしか見えない．このような綺麗な画像は NVIDIA の研究者だからこそ，NVIDIA の GPU を思う存分使えるからこそ作れたものだと思われるが，これほど綺麗でない一応顔と思える程度のものならば家の GPU 入り PC でも作ることができる．

GAN の仕組みは先ほどの Yann LeCun の言うように非常に興味深いアイディアに基づいているが，それは次のようにまとめられる．

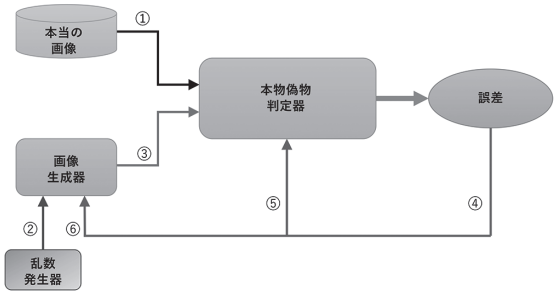


Figure2

ここで重要なのが左下の画像生成器と真ん中の本物偽物判定器である（この名前は適当につけました．Generator, Discriminator のつもりです）．この二つが，機械学習によって鍛えられる対象であり，これがどのように鍛えられるかで結果が決まります．図の意味を簡単に説明する；

①本当の画像を本物偽物判定器に流す．ここで画像は，例えば 100 × 100 ピクセルであれば 100 × 100 個の数字であり，これを判定器に流すと言うこと．

②乱数発生器が乱数を作る．numpy など python のライブラリにはそのような機能がある．多くの場合数十次元以上のベクトル．

③発生させられた乱数に応じて画像を作る．これは例えば乱数が M 個 r_m であり，画像が N 個の数 Im_n $(1 \le n \le N)$ であるならば，

$$\mathrm{Im}_n = \sum_{m=1}^{M} a_{m,n} r_m + c_n, \qquad (1)$$

と言う形のものを組み合わせたようなものである．

④これら①と③のデータの真偽を判定し，それぞれに対してその誤差（①であれば真正の画像に対して 70% 真正と判断したら 30% の誤差があり，③ であれば 70% 真性と判断したら 70% の誤差と言うような）を出力する．

⑤この出力により，本物偽物判別器は本物を本物と判断し，偽物を偽物と判断するように学習（訓練）させられる．この学習は，先ほどの式（１）のような式の係数 $a_{m,n}$ を正しい判断をする方向に修正するようなことを言う．

⑥それと同時に画像生成器の方では，本物偽物判定器を**騙すような**学習をする．作り出す画像が判別器によって真正（先ほどと逆）と判断されるような画像を作ろうとする．

つまり，学習する主体は判別器と生成器であり，判別器は騙されまいと努力し，生成器は判別器を騙そうと努力する．このように互いに切磋琢磨し，判別器と生成器が優れたものとなって行く．例えれば，贋金つくりとそれを見破ろうとする鑑定者が，それぞれ相手に負けまいと努力する．コンピュータにこのような学習をさせるのである．ただ，切磋琢磨という言葉から分かるように，両者の実力がある程度拮抗している必要がある．

1.2 問題の所在

（1）偽物を作り出す画像生成器は，勝手に画像を作るのではなく，入力された数によって画像を作る．機械に「適当に描け」と言っても無理なので，何らかの種となる入力が必要で，その入力として乱数が使われる．しかし勝手な乱数に対して真正の画像類似の画像を作り出すと言うのは，人間の顔ならばともかく，MNIST のようなものでは妥当ではない．
それは，生成器の訓練は，与えられた乱数全てに対して真正の数字を出力するように訓練されるので，つまり一定の空間の数全てに対して真正の数字が生成されるように訓練される．しかし顔と違い，二つの数の中間的なものは普通は数には見えないはずである．生成された乱数に対しては数字にならない画像を必ず出力するはずなのに，その画像に対しても真正と判断されるよう訓練させるというのはおかしな気がする． MNIST で画像を生成させようとすると，しつこい mode collapse（本来１０個の数が出て欲しいが，１つ，もしくは異常に少ない画像のみを出力する）が起こる原因の一つではないか？と自分には思えた． 画像生成器側だけから言えば，このような問題は常に同じ正しい画像を出力するような incentive となるからである．
MNIST のような単純なもの以外でも，いくつかのカテゴリーに別れる対象に対しては，先ほどのような単純な乱数で画像を作ると言う枠組みは適当ではないと思われる．つまり，全ての (乱) 数に対して数を生成させようとすると，ひとつの数しか生成できなくなる．これは生成器が連続関数（(1)）であることからそうならざるを得ない．

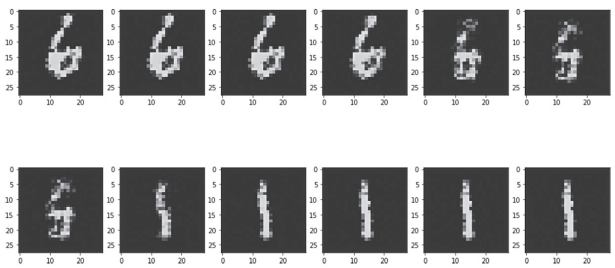


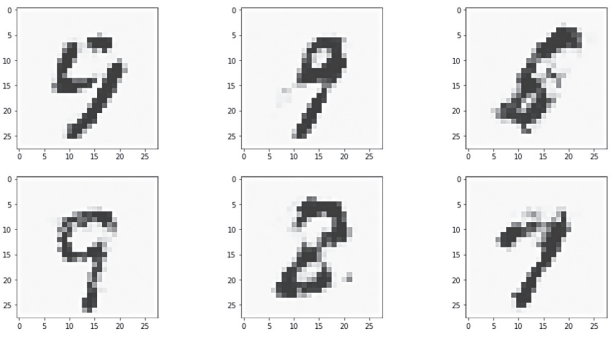
Figure3. ６と１を生成する乱数を等分して生成器で表示させたもの

上図は，６を表現した乱数（20 次元のベクトル）から１を表現した乱数までを 12 等分したベクトル（乱数を r_6 , r_1 とおくと，$r_1x + r_6(1-x),\ x = \{0, \frac{1}{11}, \frac{2}{11}, \cdots, \frac{11}{11}\}$ で決まるベクトル）に対応する数を記したものである．途中で数字でないものが現れているが，先ほどの GAN の方法によると，これらに対しても本物偽物判別器が正解となるように画像生成器をチューニングすることになる．これが可能になるのは常に同じ数字しか出力しない場合であるから，これはわざわざ mode collapse を起こすようにしているようなものである．
実際次にあるように様々な乱数に対して同じ数しか出てこないということが起こる．

そこでこのようにはっきりとしたカテゴリーに別れる対象に対しては単なる乱数以外で生成すべきであると考えられる．
（2）であるにも拘らず，LeakyRelu や LayerNorm を使うと意外なことに mode collapse がかなり抑えられる．ここで Relu という関数は，

$$\mathrm{Relu}(x)=\max(0, x)$$

として定義される非常に単純なもので，LeakyRelu は Relu を少し変えたものである．Relu は単純ではあるが，不思議なことに非常に有用である．以下の数はそれらを含んだ GAN によって生成された（乱数により）数である．



どうしてバラエティを持つのか，が問題である．本物偽物判別器の訓練によるものであるが，この点が分かりにくい（自分にはよく分からない）．おそらく統計的なことが関係していると思う．

2. 問題に関して

LeakyRelu と LayerNorm でうまくいくところの原理がよく分からないが，原理がはっきりしない物よりも，はっきり分かることを使いたい．そこで稚拙であっても理屈にかなった方法を考えることにする．

2.1（1）について

Figure3 にあるように，単純な乱数に真正な数が全て対応することはありえない．これが人の顔と違うところである．単純な乱数であることが問題であるとするなら，その代わりに，例えば

```
def random_generator(size):
    aa=np.random.randn(size)
    a=np.amax(aa)                    式（1）
    b=(aa/a)**pow
    random_data=torch.Tensor(b)*a+
    torch.randn(size)*const
```

のような関数を使えば良いのではないか？と思える．この関数は，size 次元の乱数を発生させ，最大数以外を小さめにするものである (pow によって)．この乱数発生関数を Generator に用い，何回か実験した．（パラメータは size=20, pow=3, const=0.02 とした．pow に関しては，最大で pow=7 までぐらいで，pow=10 にすると学習できなくなった．）

（2）について

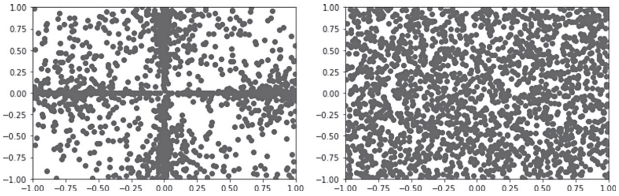
なぜ LeakyRelu や LayerNorm を使うことで幾種類もの数を出力するのであろうか．コンピュータが幅広い数を出力しようとするようなインセンティヴは何だろうか？　あまり分かっていないが，curse of dimensionality の原因となる高次元の球の体積のほとんどが表面近くにある，ということと関係している気もする．

そこで Discriminator の方では，少し異なる乱数を用いることにする（この乱数によって Generator によって画像が作られ，その画像を偽物と判断するように Discriminator を訓練する）．その乱数は以下のようなものである．

```
random=torch.rand(size)*2.0-torch.ones(size).  式（2）
```

これを Fig.2 の③のところの入力の乱数として用いた．なお，size は乱数の次元で，ここでは 20 であり，機械学習のライブラリとしては GPU も使いやすいので pytorch を使っている．

ここで二種類の乱数を使う理由は，先ほど書いたように，正しい数を発生させる乱数はおおよそ１０通りのパターンに分類されるようにし，もう一つの Discriminator を鍛える乱数は広く様々な数を出力するようにしたということである．ここで二つの乱数の２次元の場合のイメージを次に示す．



左：式（1）の乱数，右：式（2）の乱数

ここでこの pytorch によるプログラムコードの構成を以下に記す．ここでは多くの部分で文献 6) に従っている．最初に前の部分で本物偽物判定器と書いた Discriminator の主要部分は，

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model=nn.Sequential(
        nn.Linear(784,200),
        nnSigmoid(),
        nn.Linear(200,1),
        nnSigmoid(),
    )
```

とした．これは文献 6) では LeakyRelu や LayerNorm を用いていたが（少なくとも mode collapse を避ける段階では），ここでは一番原始的な上記の構成とした．これで mode collapse なしに動くのか？が一つの問題でもある．

次に，画像生成器と書いた Generator であるが，これは最初は

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model=nn.Sequential(
        nn.Linear(size,200),
        nn.LeakyRelu(0.02),
        nn.LayerNorm(200),
        nn.Linear(200,200),
        nn.LeakyRelu(0.02),
```

```
        nn.LayerNorm(200),
        nn.Linear(200,784),
        nn.Sigmoid(),
    )
    self.optimizer=torch.optim.Adam(self.parameters(), lr=0.0001)
```

とした．こちらは LeakyRelu も LayerNorm も用いた．その上文献 6) よりも層を厚くしたが，これは全く必要ない．厚くしないほうが綺麗な場合もあるし，そうでない場合もある．以下での実例の最初のいくつかは厚くした場合なので，それを挙げておいた． 薄くする，と書いた部分は，

```
nn.Linear(200,200),
nn.LeakyRelu(0.02),
nn.LayerNorm(200)
```

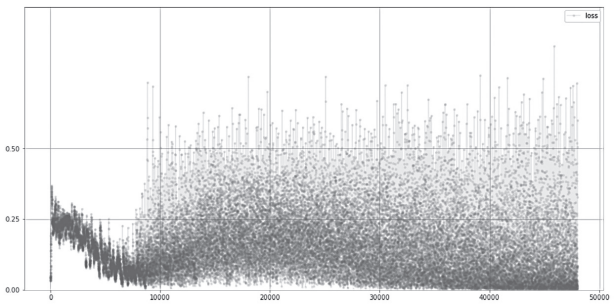
の３つを除いたものである．この点ももう少し調べるべきであるが，時間の都合で略した．

いずれにせよ，Generator と Discriminator は一般的には対称的に作られるが（切磋琢磨のために），ここでは非対称とした（非対称で動いた）．一方の方が能力が高すぎるとライバルにならない，ということであるが，自分の構成では問題なく動いた．

上記の通り Discriminator に関しては，文献 6) でギリギリ何とか mode collapse を防ぐ構成であったものを，少し弱くして，その分乱数発生部分を特殊にしたということである．なお，この辺りの様々なパターンでの比較（mode collapse に関して）は時間が足りず十分検討できていない．のちに実験してみたい．他の部分は典型的な構成と思われるので省略する．

2.2 結果

MNIST の数字６万文字を４回 (4 epochs) 繰り返し学習させた．Discriminator の学習具合を 6) の方法で表示すると



学習による loss の変化

このようになった．うまく行くときはこのような雲のようなグラフになることが多い．これで生成した数字が次の Fig.4 および Fig.3 で示したものである．

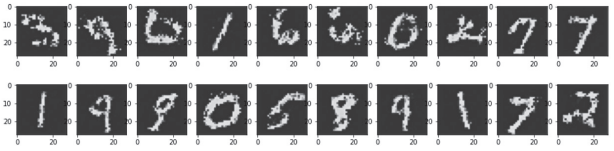


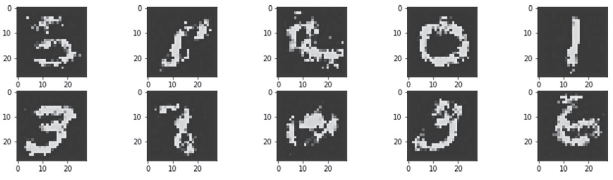
Figure4.

Figure4 は，

```
for i in range(2):
    for j in range(10):
        x=np.zeros(size)
        x[j+i*10]=1.0
        x=torch.FloatTensor(x) + torch.randn(size)*0.01
```

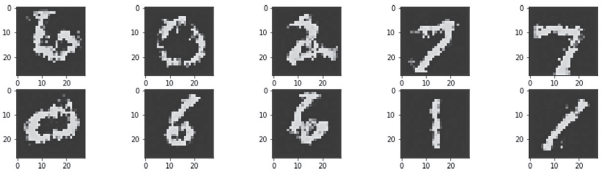
で作られた x に対して Generator が作った画像である．ほぼ，0 から 9 まで出揃っているが，4 が綺麗に出ていない．順番は任意であり，計算のたびに変わる（初期設定自体に乱数が使われているため）．もう少し訓練すれば（epoch を増やせば）解決するかもしれない．数字の数は 10 個なのに，20 次元なので当然のことながら重複がある．1 が 3 箇所に出てきて，それぞれ少し違うところは理解できる．

こうなると，次には次元を 10 にしたらどうなるか？が知りたいところである．20 であったのは，数回の計算で 20 が比較的綺麗な画像を出力したからである．改めて size=10 としてみると．

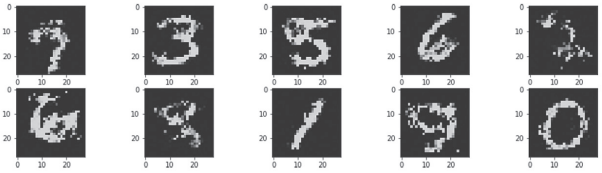


0,1,3,6,7 らしきものは見えるが，5 と 8 ははっきりしないし，2,4,9 は見当たらない．10 通りに分類されるものであっても，10 次元では狭いのだろうか．

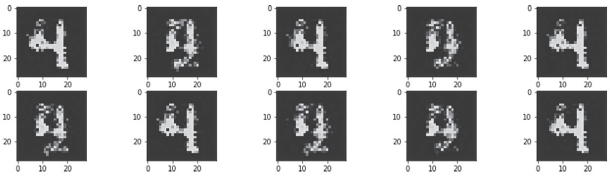
再度，Generator の層を厚くした部分（前のページ右，真ん中より少し上）を元に戻したところ，



となった．やはり全ての数は出ないが，少し綺麗になった．面白いのは，0,1,6,7 は共通して現れることである．認識しやすいのだろうか．6 は 3 回も出現している．そしてこちらには 2 も出ている．もう少し長時間の訓練をすれば結果が良くなるようにも思える．時間をかければ，現在縮退（というのか，特定の数字が出てこない＝どれかと重なっていて分離されない）している数字が分けられる可能性がある．そこで 10 次元でもう少し時間をかけて（6 epochs）やってみると，



となった．綺麗ではないが，0,1,2,3,5,6,7,8,9 が見て取れる．6 が重なっていて，その代わりに 4 がない．4 は今までも出にくかった．Figure4 では，9 と 4 が混ざったような数が出力されている．なお, これとは違う run の結果ではあるが, はっきりとした“ 4 ”が出力されないが，4 によく似た数を表示した時に，それを表示した乱数の前後の数を 10 個ほど出力し，それで描かれる数を出したところ

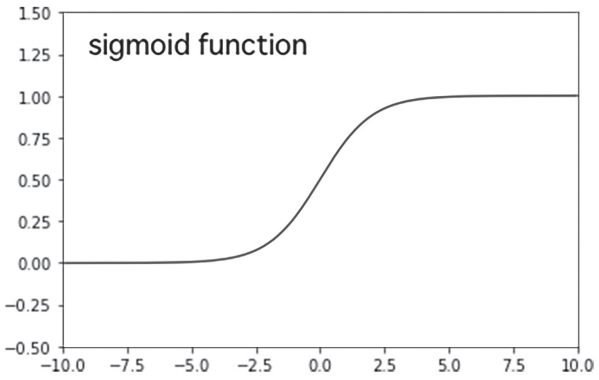


となった．9 と 4 の中間のように見えるところの周りに比較的はっきり 4 に見えるものがあったのだ．

2.3 数字生成のまとめ

GAN による数字を生成するという目的のために，乱数発生のところに少し手を加えた．その結果，Discriminator を少し簡単化しても mode collapse が起きないことと，次元を 10 にしたら, ほぼ 0 から 9 までの 10 個の数字が出力された．この場合，Generator はそうが浅い方が良いようであった．

しかし実は分類される理由がよくわからない．問題提起のところで書いたように，同じ画像を出し続けることになりそうにも思える．ここでそうならない理由を少し考えてみたい．なお，ここではもちろんオリジナルの model ではなく，乱数を変更して Discriminator を簡単にした以上で扱っていた model である．key は，Generator のところで sigmoid の代わりに LeakyRelu を使ったことではないだろうか？ sigmoid も LeakyRelu も共に線形ではないが（むしろ線形でなくするために存在しているが），sigmoid は線形とは似ても似つかぬものである．



これに対して LeakyRelu は，線形ではないが少なくとも次の性質は満たす；

$$L_R(x+x) = L_R(x) + L_R(x),$$

ここで$L_R()$はもちろん LeakyRelu のつもりで書いたものである．この性質を持つため，もし二つの乱数ベクトル$\mathfrak{a}$, $\mathfrak{b}$が Generator により同じ画像（例えば５のような）に対応する場合，つまり

$$G_e(\mathfrak{a}) = G_e(\mathfrak{b})$$

の時,

$$G_e(p\mathfrak{a} + q\mathfrak{b}) = G_e(\mathfrak{a})$$

となる（$p+q=1$ ）のではないかと想像される（ここで$G_e(\,)$は Generator の意味で用いた）．もう少し具体的に言えば，例えば$(1,0,0)$と$(0,1,0)$が同じ像に移された場合，その中点である$(\frac{1}{2},\frac{1}{2},0)$も同じ像に移されることになる．もちろん Generator には，明らかに線形でない関数 LayerNorm もあるので正確な話ではない．しかしある程度はそう言えるのではないかとも思われる．そこで LayerNorm を見てみると，これは pytorch の online manual によると[9)]

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \varepsilon}} * \gamma + \beta$$

という形の式であり，線形ではない．しかしこの式の入力の段階で同じ数値になっていたとしたなら，全体をみれば線形になる．つまりたった２段しかない Generator であるが，LeakyRelu のところまでが同じなら表示される画像は同じものになる．と，苦しい説明を考えたが，もしかすると無くても良いのでは？と考え取り去ってみた．予想通りちゃんと数字は出力された．文献６）によると，LeakyRelu と LayerNorm は共に必要で，一方がないと全くダメなのであったが，自分の乱数バージョンでは必要なかった．そうすると，Generator のクラスは

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model=nn.Sequential(
        nn.Linear(size,200),
        nn.LeakyRelu(0.02),
        nn.Linear(200,784),
        nn.Sigmoid(),
    )
```

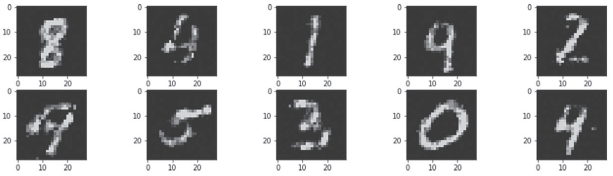
と簡単になってしまった．こうなると最後の sigmoid も線形っぽいものに変えるとどうだろうか？と思えてくる．

つまり，なるべく多くの要素を線形にしてしまえばうまく行きそうである．原理上全ては無理であるが，この最後の sigmoid も線形のものにできないだろうか．こう考えて sigmoid の代わりに次のようなものに取り替えてみた．

```
torch.reshape(torch.clip(a,min=0,max=1),(-1,))
```

である．ここで a はその前の段階で計算されたもので（このため sequential から関数方式？に改めた），clip は min と max の間に切り詰める関数．これが必要なのは画像の各ピクセルが０から１の間の数値で書かれているからで（0 ＝黒，1 ＝白），それゆえその間に制限した．

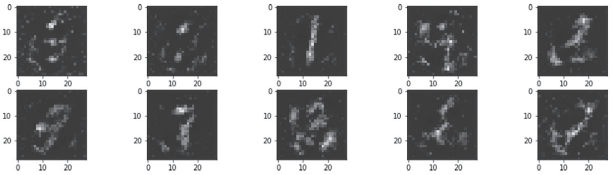
この時点で，GAN の構成は一番単純な，文献６ではおよそ mode collapse を避けられないものを基本にして，それに若干の工夫をした乱数と，上記の clip を適用したものとなった．こうして訓練した後，i 番目＝１，その他 ＝０という 10 次元ベクトルに対して表示された画像は



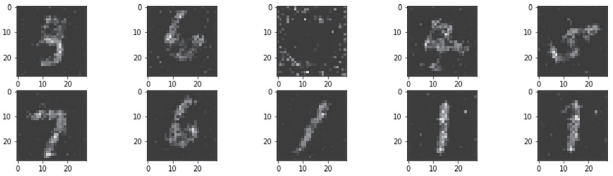
である．clip を使ったためか，全体として暗いが，mode collapse は起きていない（6 がないが）．比較的きれいに出来ているようにも思える．

３．ここまでの結果と議論

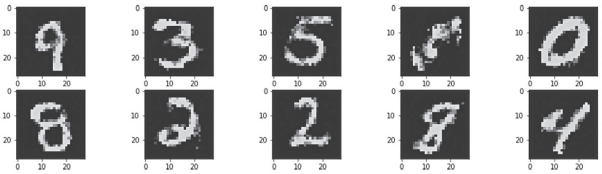
文献６は，GAN を面白く説明している本で，少しずつ手を加えながらどう変化していくか実験しながら進めていくというスタイルの本である．この本では色々手直ししながら最終的に LeakyReku と LayerNorm を使ってやっと数字らしい画像を得ていた．上に示したように，LayerNorm のような高級なものを用いなくても乱数を変え，sigmoid を少し触ったら（clip に変えたら）比較的きれない画像を作ることができた．乱数を変えない場合は



というような，およそ数字の画像とは思えないものになる．しかし乱数を変えるだけで（clip も加えたが出力の制限のため）数になった．一つ上の画像と比べると乱数の効果がよく分かる．ちなみに clip がないと（もちろん乱数は工夫した方を使って），



となり，やはり綺麗にはできない．clip も必要なようであるが，二つ前の画像で出てこなかった６らしきものが２箇所で出て来ていることが興味深い（偶然だろうが）．ついでに，全ての activation(sigmoid や LeakyRelu など) を単なる Relu に変えてみたところ，



このようになった．１と６が混ざったような画像があるが決して悪くはない．調べたところ５を生成した乱数の近くに６が出て来た．６と５の違いよりも上記の二つの２の違いの方が大きいのかもしれない．これらの画像は少し長い時間（8 epochs）学習させて作られた．この構成は本当に簡単で，主要部分は

nn.Linear(),
nn. Relu(),
nn.Linear(,),
nn.Relu()

これだけで（これは Discriminator のもの．Generator は最後の Relu の代わりに clip を用いた），非常に簡単である．ここまでの結論としては，構成自体は非常に簡単でも乱数を一様なものではないものに変えることによって mode collapse をかなり防ぐことができ，さらに比較的きれいな画像が得られるということである．

あっちに行ったりこっちに行ったりしていたので分かりにくいが，意外に簡単にそこそこ綺麗な像を作ることができた．

４. 発展

これまでの方法を最初に挙げたような人間の顔に適用したらどうなるであろうか？綺麗にカテゴライズできるものではないので，却って上手くいかないと思われるが，これまでの方法を画像に適用するのも意味があると思う．以下では文献６の画像の GAN のコードに手を加えて試してみることにする．

まずは文献６の p.139 までで，2 epochs 計算して出来た画像が以下のものである．

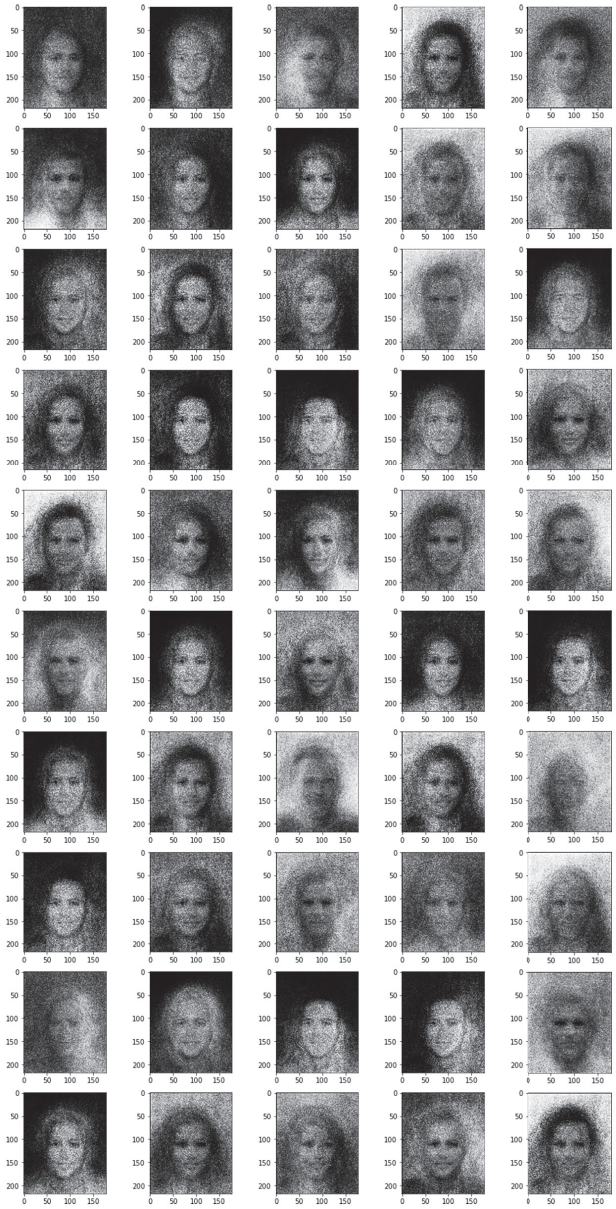
これらは，ランダムに５０個出力した．これを出力した Generator の主要部分は

nn.Linear(100,3*10*10)
nn.LeakyRelu()
nn.LayerNorm(3*10*10)
nn.Linear(3*10*10,3*218*178)
nn.Sigmoid()

である．まずこの部分を

nn.Linear(100,3*10*10)
nn.Relu()

nn.Linear(3*10*10,3*218*178)
nn.Relu()
という非常に原始的な構成にした．これは MNIST でもそうやったのであった．さらに，乱数を前述のように変えて 2 epochs 訓練し，同じ乱数発生器の乱数に対して得られた画像は以下のような画像である．



あまり大きな違いはないが，LayerNorm などを除いたにしては若干綺麗になった気もする．さらに，こちらの方が「明らかに男」と思える画像がある．そのような意味で若干は良くなったのかもしれないが，似た顔が多くなったようにも見える．この画像では clip を使っていないが，MNSIT の場合と異なり使わない方が綺麗なようである．ただ，乱数を単なる乱数（工夫していないもの）にしてもそこそこ綺麗な画像ができた．mode collapse もある程度避けられた．ただ，出力させるときに，つまり上記の様な画像を描かせるときには工夫した乱数の方が綺麗であった．いまひとつ良く分からないが，顔の場合はカテゴリー分け乱数の意味は MNIST の場合とはだいぶ違う様である．

条件をもっと色々と変えてみると面白いのだろうが，変えてばかりいると訳が分からなくなってくるので，この辺で一応終わりとする．

5. 本当の結論

本実験の結論としては，GAN において乱数をどう選ぶかということは非常に重要であるということの確認である．特に MNIST のようなカテゴリーに別れるものは乱数の選び方が重要なようで，はっきりとしたカテゴリーに別れない人の顔のような対象でも，乱数の選択は結果に随分大きな影響を与えることがわかる．文献 6 の中にももっと綺麗な GAN の例があり，さらに他の文献 7，8 にも色々ある．どこまで有効か分からないが，乱数を色々と工夫するというのは意外に面白いことかもしれない．自分の知る範囲では乱数をいじっている文献はない．

さらに，最初の方に出てきた本物偽物判別器を鍛えるための乱数（式（2））は，その出力範囲は　の範囲である．画像生成器の乱数は 1 以下の 0 以上の数なので，範囲が異なるが，この方が（一方のみを -1 まで広げたほうが）結果が良かった．理由はあまりはっきりとしない．

最後に一つ，この方法で分類ができないか？という気がした．MNIST で i 番目だけが 1 で他が 0 であるベクトルに対してほぼ異なる数が出力されたように，もしかすると一般的なデータに対してもなんらかの分類が行えるかもしれない．ただ，残念ながら顔の場合に次元を減らした場合（ex. dim=5），男女に分類されるというようなことを期待したが，そうはならなかった．特に次元を 2 次元にまで下げた時には，それぞれの単位ベクトルに対して顔と背景がそれぞれ現れた．

参考文献

1) http://yann.lecun.com/exdb/mnist/
2) Francois Chollet, Deep learning with python. Manning , 2018
3) Kajima Y., Summation of certain locally binary forms and

its applications to the Fast Multipole Method.
https://arXiv.org/pdf/2009.00767, (2020)
4) Goodfellow, I., J., et.al., Generative Adversarial Nets.
https://arxiv.org/pdf/1406.2661.
5) Karras, T., Aila, T., Laine, S., Lehtinen, J., Progressive growing of GANs for improved quality, stability, and variation.
https://arxiv.org/abs/1710.10196.
6) Tariq Rashid, Make your first GAN with pytorch.
ISBN 9798624728158
7) Jakub Langr, Vladimir Bok, GANs in action.
Manning 2019
8) David Foster, Generative Deep Learning.
O'Reilly, 2019
9) https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html