Digit Recognizer

https://www.kaggle.com/competitions/digit-recognizer

庭師

Digit Recognizer

- …の場合はここから始めてください。
- RまたはPythonと機械学習の基礎知識はある程度あるけれど、コンピュータービジョンは初めてという方。このコンテストは、事前に抽出された特徴を含む古典的なデータセットを用いて、ニューラルネットワークなどの技術を学ぶのに最適な入門コースです。
- コンテストの説明
- MNIST(「修正米国標準技術研究所(Modified National Institute of Standards and Technology)」)は、コンピュータビジョンにおける事実上の「Hello World」データセットです。1999年のリリース以来、手書き画像からなるこの古典的なデータセットは、分類アルゴリズムのベンチマークの基盤として機能してきました。新たな機械学習技術が登場する中でも、MNISTは研究者と学習者の両方にとって信頼できるリソースであり続けています。
- このコンテストでは、数万枚の手書き画像のデータセットから数字を正しく識別することが目標です。回帰分析からニューラルネットワークまで、あらゆる要素を網羅したチュートリアル形式のカーネルセットをご用意しました。様々なアルゴリズムを試して、どのアルゴリズムが効果的か、また各手法を比較検討して、実際に体験してみることをお勧めします。
- 練習スキル
  - シンプルなニューラルネットワークを含むコンピュータビジョンの基礎
  - SVMやK近傍法などの分類手法

データセット（1）

- データ ファイル train.csv および test.csv には、0 から 9 までの手描きの数字のグレースケール画像が含まれています。
- 各画像は高さ28ピクセル、幅28ピクセルで、合計784ピクセルです。各ピクセルには、そのピクセルの明るさまたは暗さを表す単一のピクセル値が関連付けられています。数値が大きいほど暗くなります。このピクセル値は0から255までの整数です。
- トレーニングデータセット (train.csv) には785列あります。最初の列「ラベル」は、ユーザーが描画した数字です。残りの列には、関連付けられた画像のピクセル値が格納されています。
- トレーニングセット内の各ピクセル列には、pixelxのような名前が付けられます。ここで、xは0から783までの整数です。画像上でこのピクセルを見つけるには、xをx = i * 28 + jと分解したと仮定します。ここで、iとjは0から27までの整数です。そうすると、pixelxは28 x 28行列のi行j列に配置されます（インデックスは0）。
- たとえば、pixel31 は、以下の ASCII 図のように、左から 4 列目、上から 2 行目にあるピクセルを示します。
- 視覚的に言えば、「ピクセル」という接頭辞を省略すると、ピクセルは次のように画像を構成します。

```
000 001 002 003 ... 026 027
028 029 030 031 ... 054 055
056 057 058 059 ... 082 083
 | | | | ... | |
728 729 730 731 ... 754 755
756 757 758 759 ... 782 783
```

	A	B	C	D	E	F	G	H	I	J
1	label	pixel0	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8
2	1	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	0	0	0	0	0
5	4	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	7	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0
10	5	0	0	0	0	0	0	0	0	0
11	3	0	0	0	0	0	0	0	0	0
12	8	0	0	0	0	0	0	0	0	0

データセット（2）

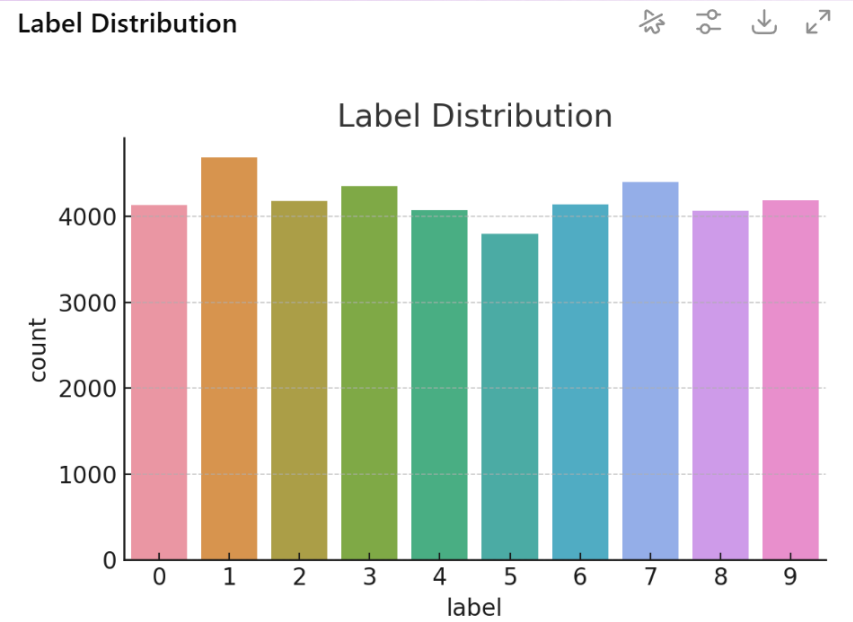
- テスト データ セット (test.csv) は、「ラベル」列が含まれていないことを除いて、トレーニング セットと同じです。
- 提出ファイルは以下の形式にしてください。テストセット内の28000枚の画像それぞれについて、ImageIdと予測する数字を含む1行を出力してください。例えば、1枚目の画像が3、2枚目の画像が7、3枚目の画像が8だと予測した場合、提出ファイルは以下のようになります。
- このコンテストの評価基準は分類精度、つまり正しく分類されたテスト画像の割合です。例えば、分類精度が0.97の場合、3%を除くすべての画像を正しく分類できたことを意味します。

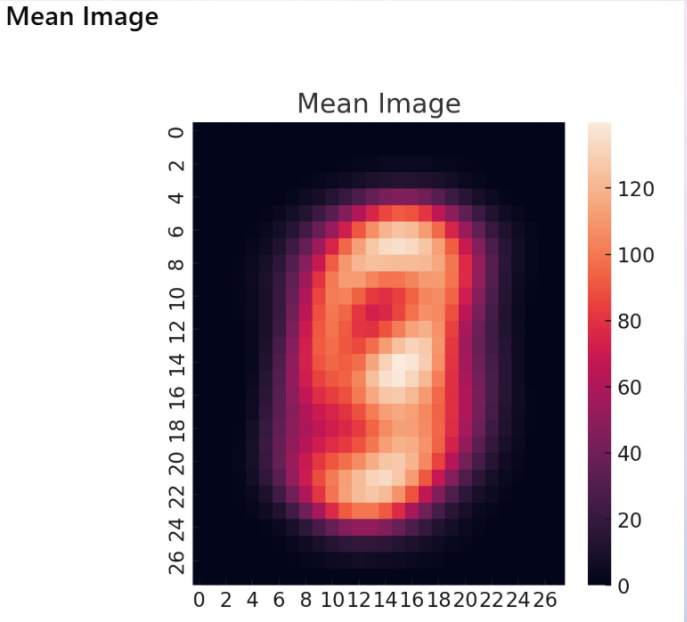
```
画像ID、ラベル
1,3
2,7
3,8
（さらに27997行）
```

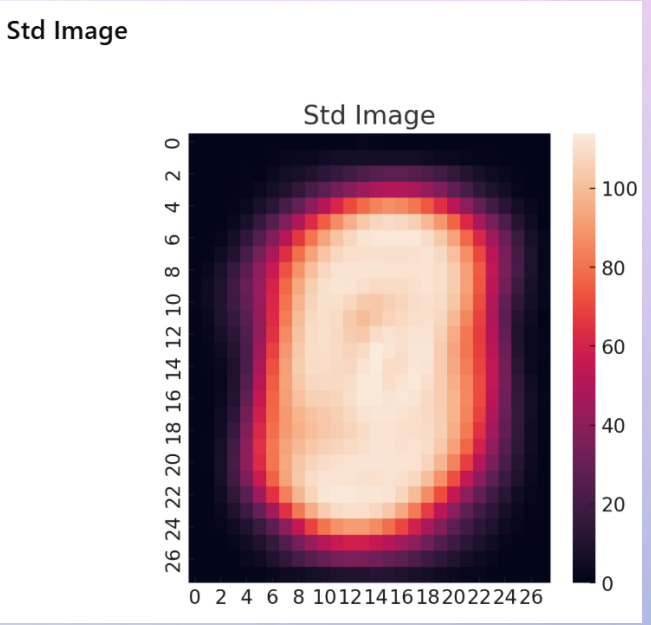
digit recognizerをEDAして

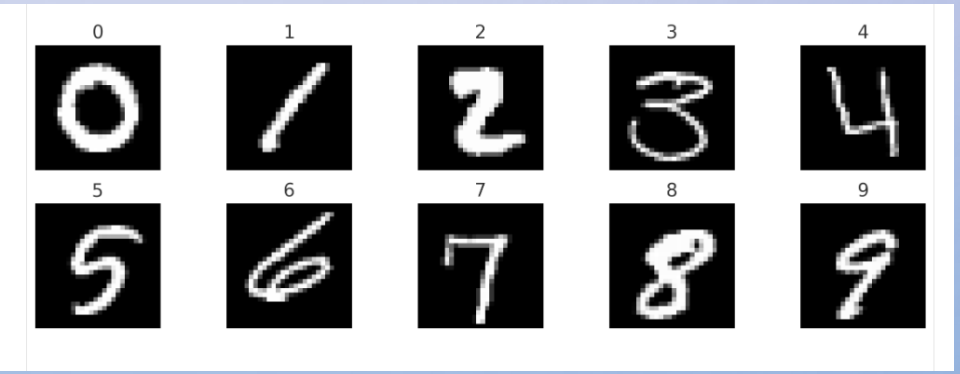
- データ概要
- 競技概要：手書き数字（0〜9）を分類する課題。 Kaggle.com+1
- データ構成：
  - train.csv：ラベル付き画像（例：42,000枚）など。 Medium+1
  - test.csv：ラベルなし画像。予測して提出。 Medium
- 画像仕様：28×28ピクセル、グレースケール（1チャネル）。各ピクセルは0〜255の整数。 GitHub+1
- 目的：入力画像から数字ラベル（0〜9）を正しく予測し、分類精度を高める（公開リーダーボード精度など）。 Medium+1
- つまり、「入門〜中級の画像分類タスクとして、CNN／MLP／データ強化等が学べる」データセットです。

train.csv, test.csv を渡してEDAして





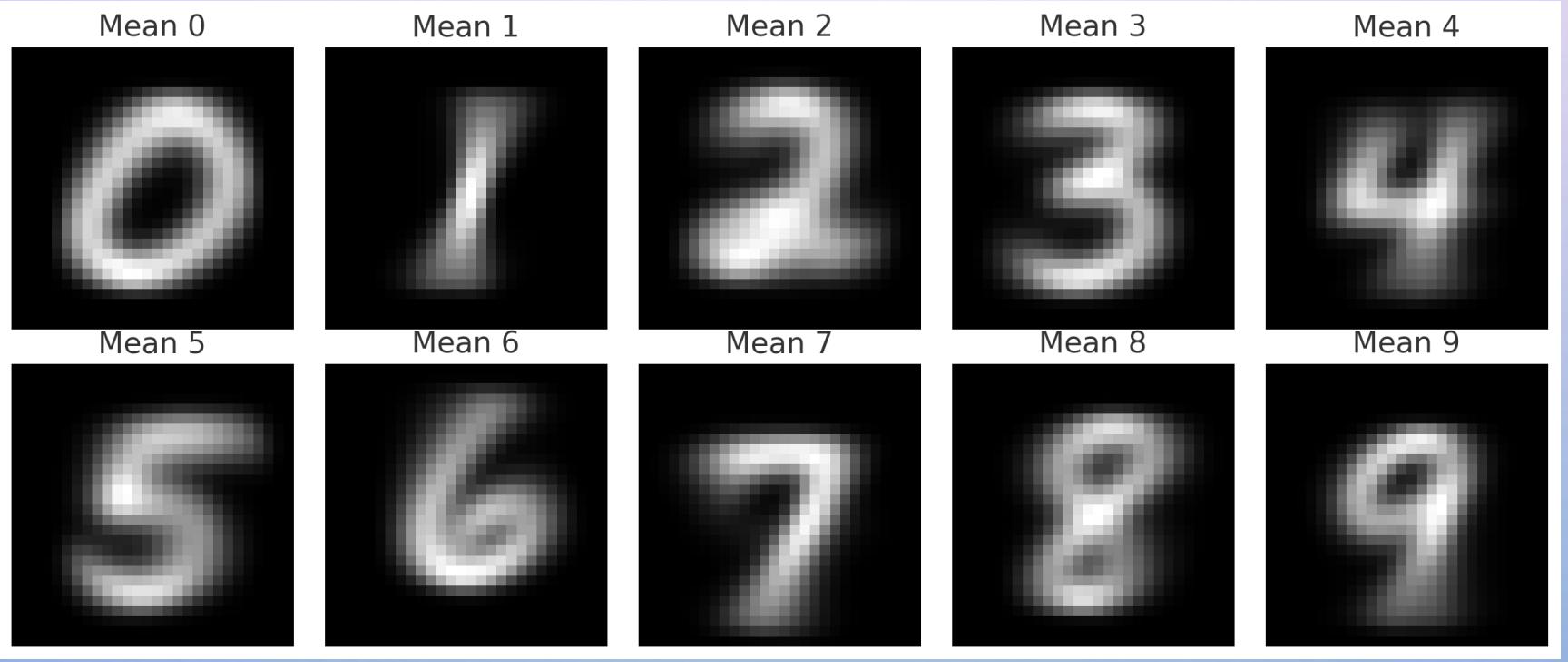




それぞれの画像の最初に1枚

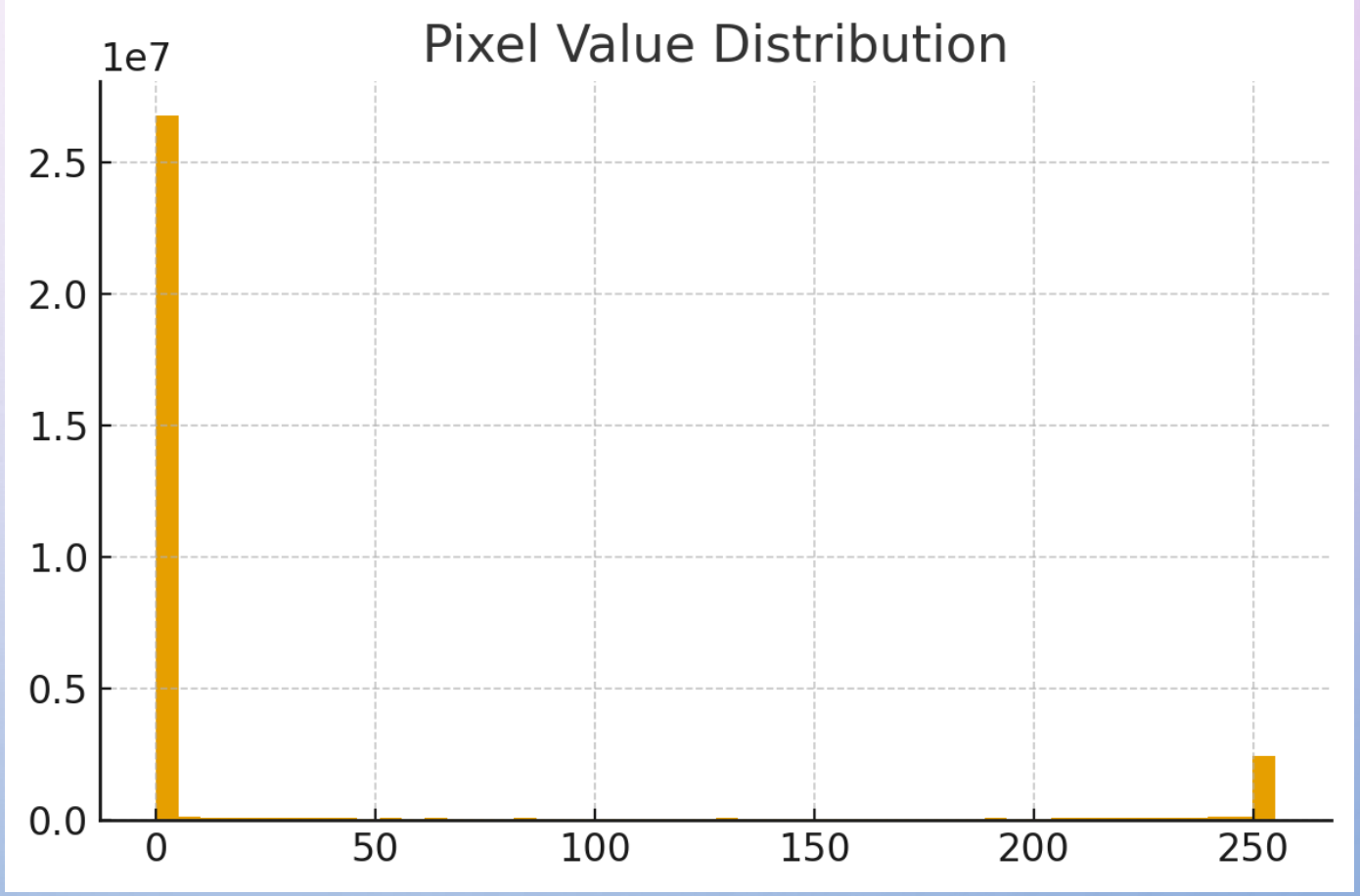
各ラベルの平均画像(0-9)

- 0：綺麗な丸
- 1：非常に細く縦線
- 8：他に比べ複雑で濃い
- 4 と 9：類似性が高い



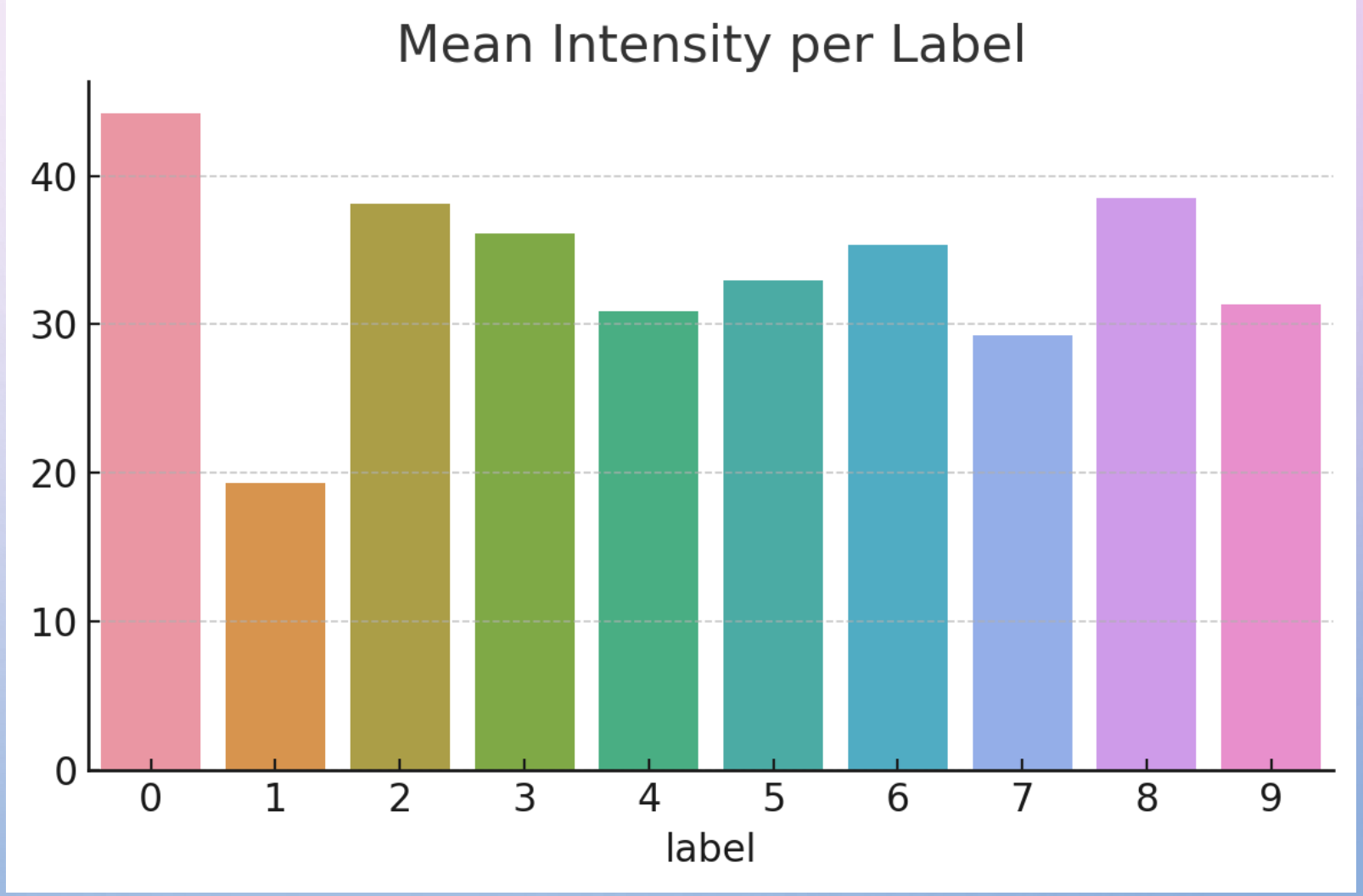
ピクセル値ヒストグラム

- ほとんどのピクセル値が 0(背景)に集中。
- 一部に 200+ の強い筆圧の白が存在。
- → これは画像分類モデル(特にCNN)で
  正規化(0〜1) or 標準化 が効果的な理由。



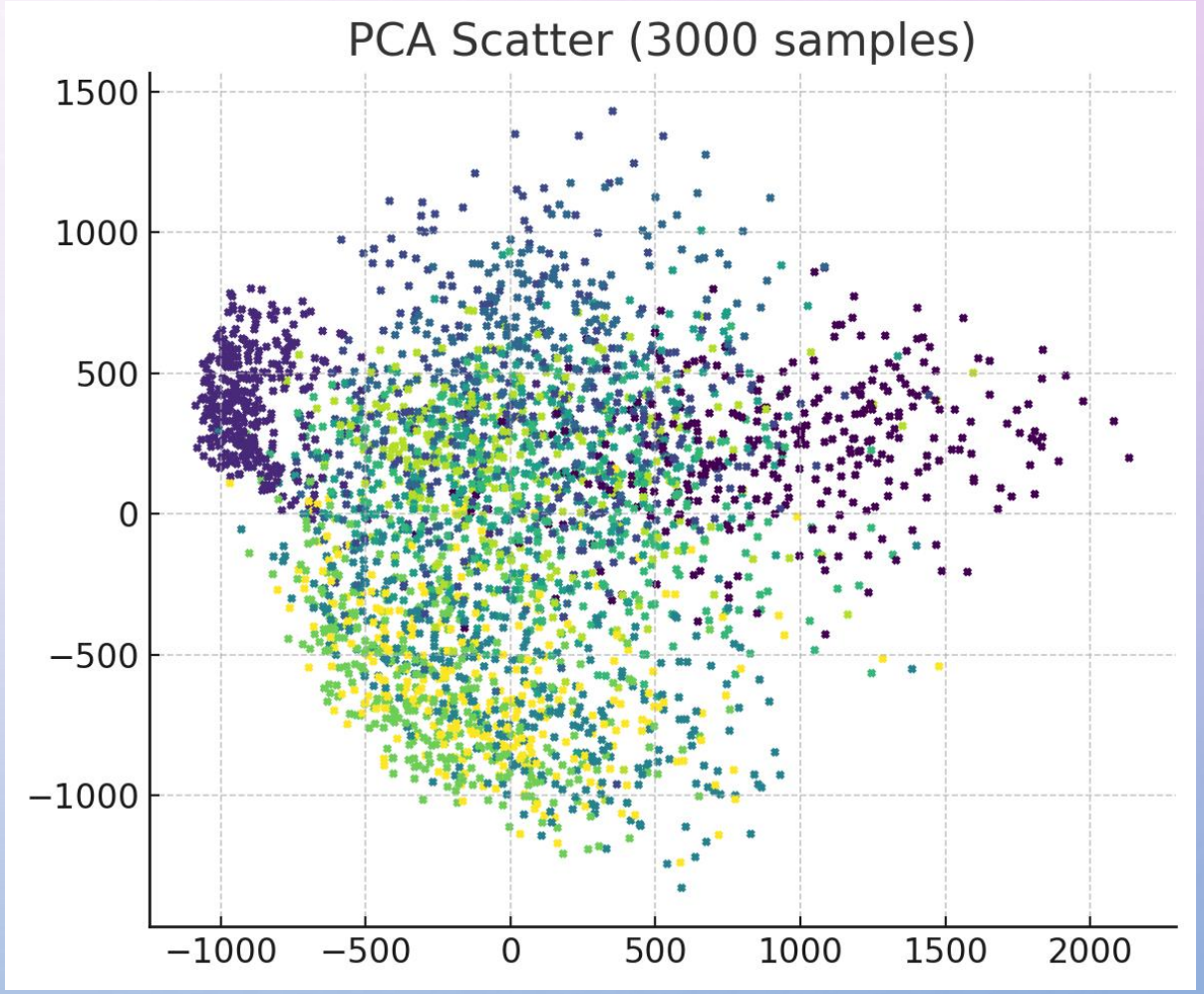
ラベルごとの平均筆圧（Mean Pixel Intensity）

- 最も“濃い”数字：0, 8
- 最も“薄い”数字：1
- → 1は誤分類されやすい数字であることを示唆。



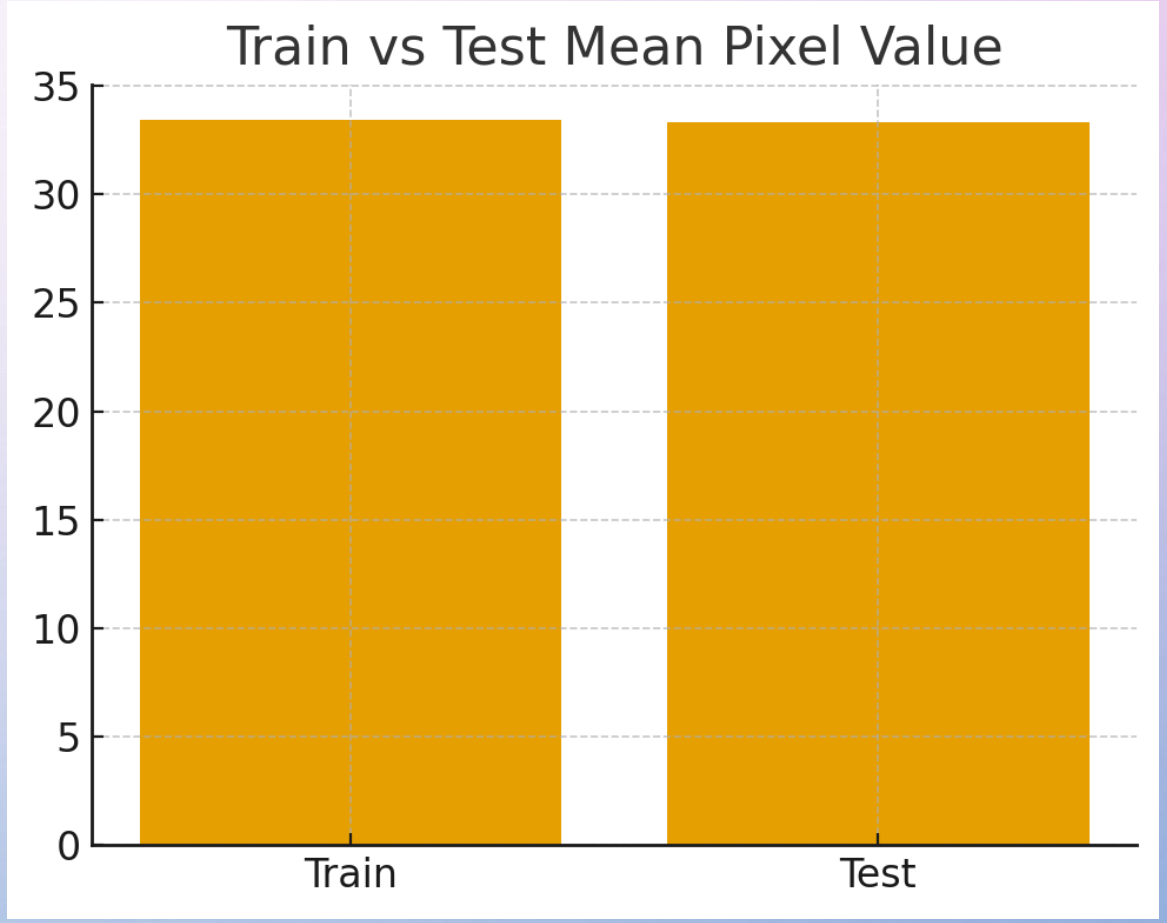
PCA 2D 可視化（3000枚）

- 次元削減で2次元にして散布図表示
- 明確にクラスごとのクラスター分離はされていない
  → 手書き文字のばらつきの大きさがわかる
- → CNNのような空間特徴を扱うモデルが必要な理由。



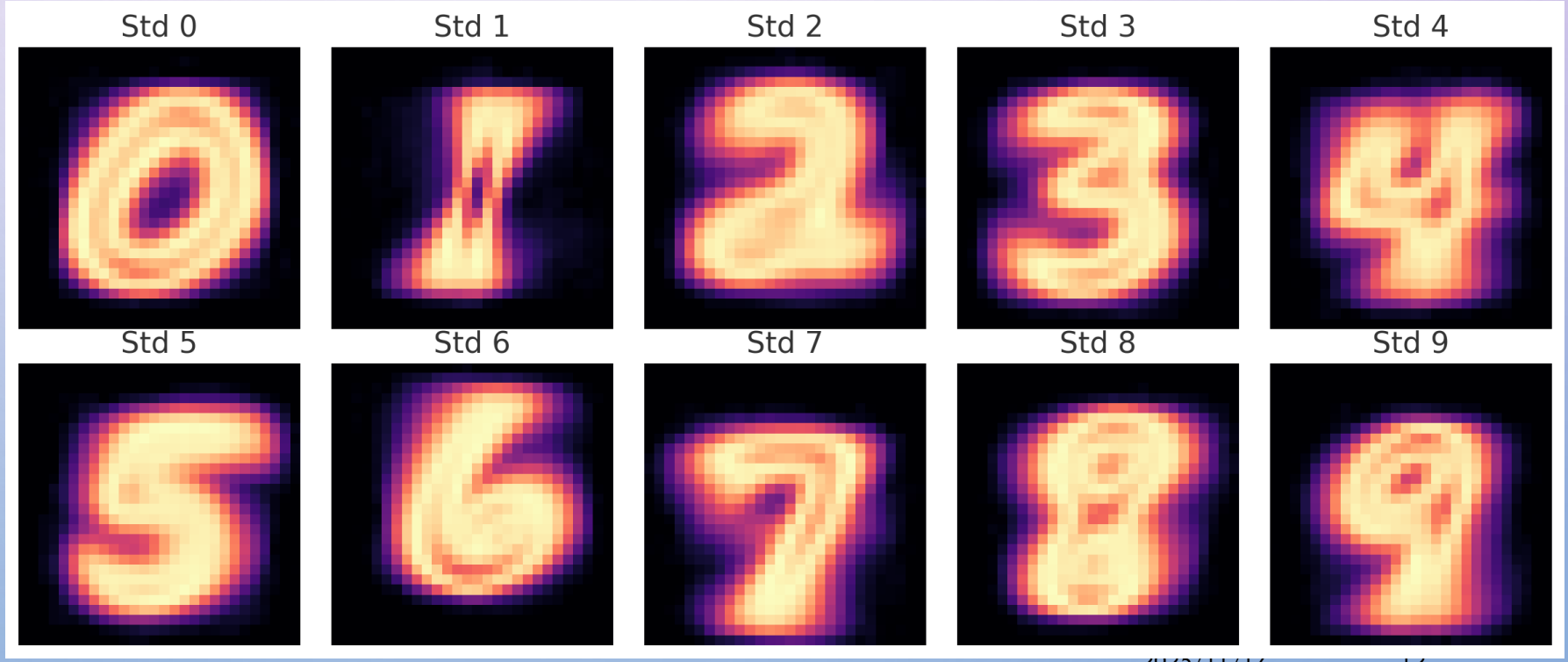
Train vs Test の分布比較

- 全ピクセル平均
  - Train ≒ Test
- 大きなデータシフトなし
- → モデルのスコアが安定しやすい良データセット。



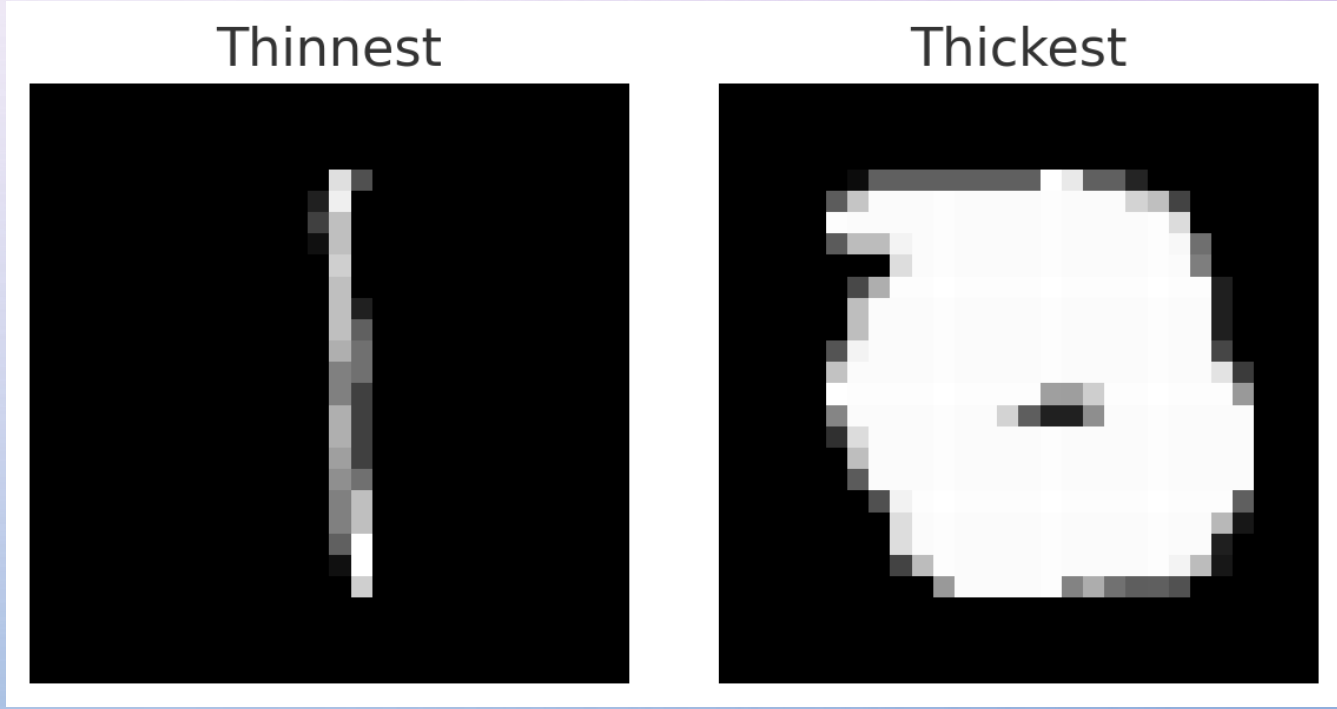
ラベル別 標準偏差画像(Std Image for each label)

- “数字ごとのバラつき(個人差)が大きい領域”が赤く表示される
- 0:右上の開き方が人によりバラつく
- 4:上部と右側の書き方が大きく変動
- 8:ほぼ全域で標準偏差が高い(複雑な書き方のため)



最も細く書かれた数字 / 最も太く書かれた数字

- → ピクセル平均値が最も低い(薄い)画像
  → ピクセル平均値が最も高い(濃い)画像
- 多くの場合:
- Thinnest(最薄) → 「1」寄りの細い数字
- Thickest(最濃)」 → 「8」や太字の「0」が多い

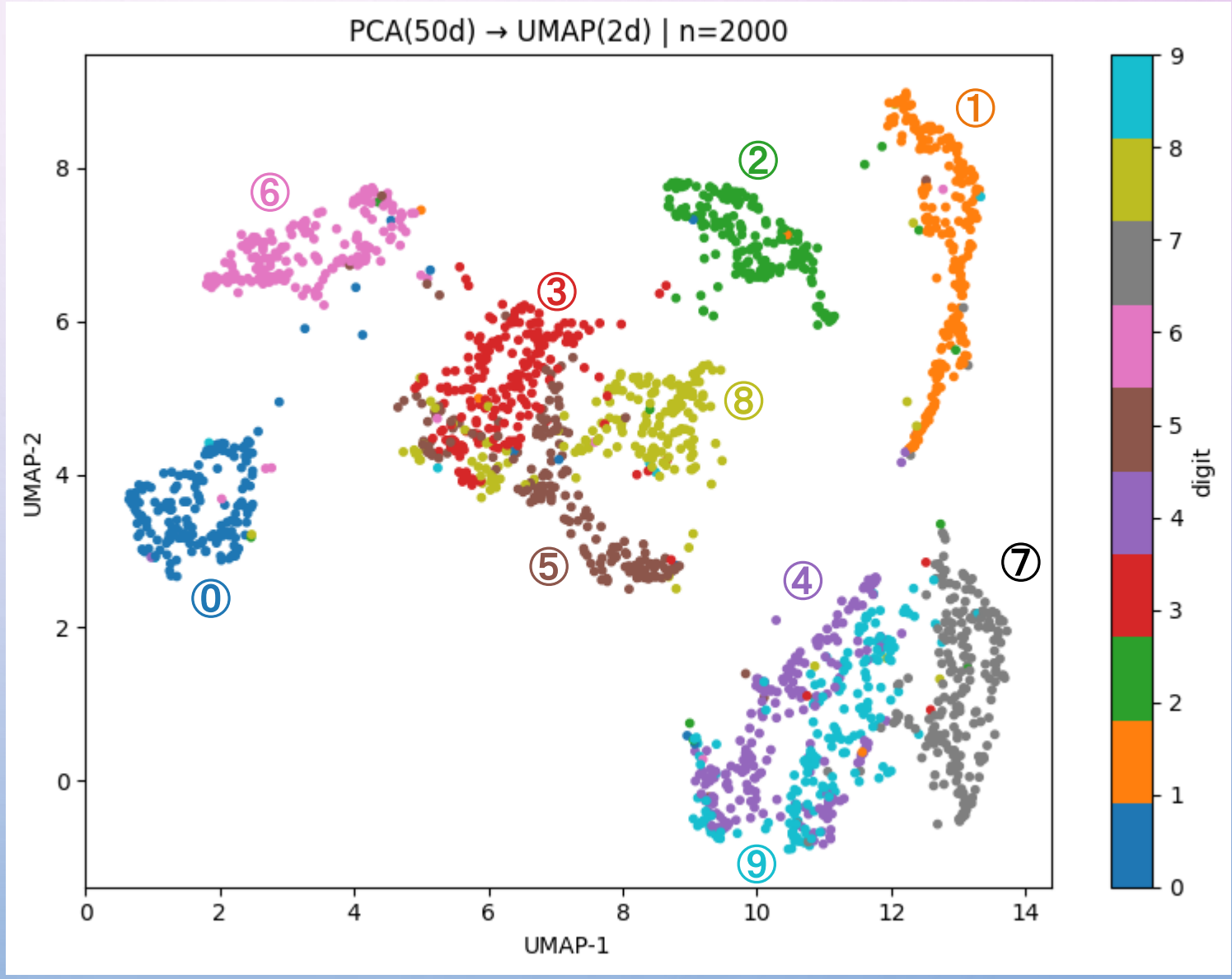


PCA → UMAP（Kaggle用）

Kaggle notebookの結果画像をpngファイルにして(CLIP STUDIO使用)、ChatGPTへupload

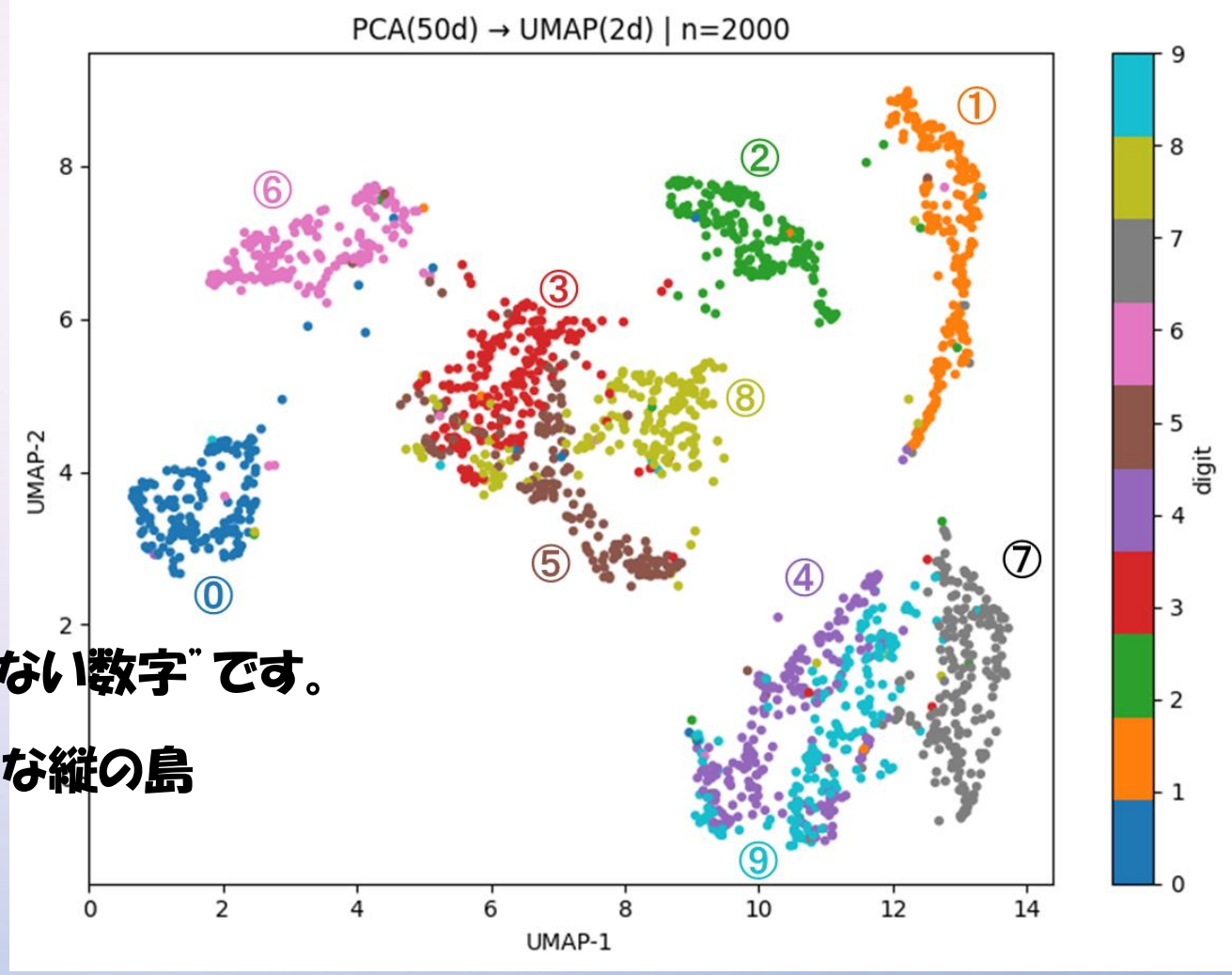
- 全体俯瞰：きれいにクラスタ形成ができている
- UMAP は MNIST で“きれいにクラスター分離できる”のが強みですが、
  今回のプロットは MNIST として理想的な分離になっていると言えます。
- 各ラベルは 独立した島（cluster）を形成しており、“中間にある混合領域”もとても自然です。

- 1.「距離が近い数字」は誤分類リスクが高い
- → 3 と 5、9 と 4 が典型
- 2.「大きく分離している数字」は学習しやすい
- → 1, 0, 6
- 3.「内部が広がるクラスタは書き方の個人差が大きい」
- → 8、5



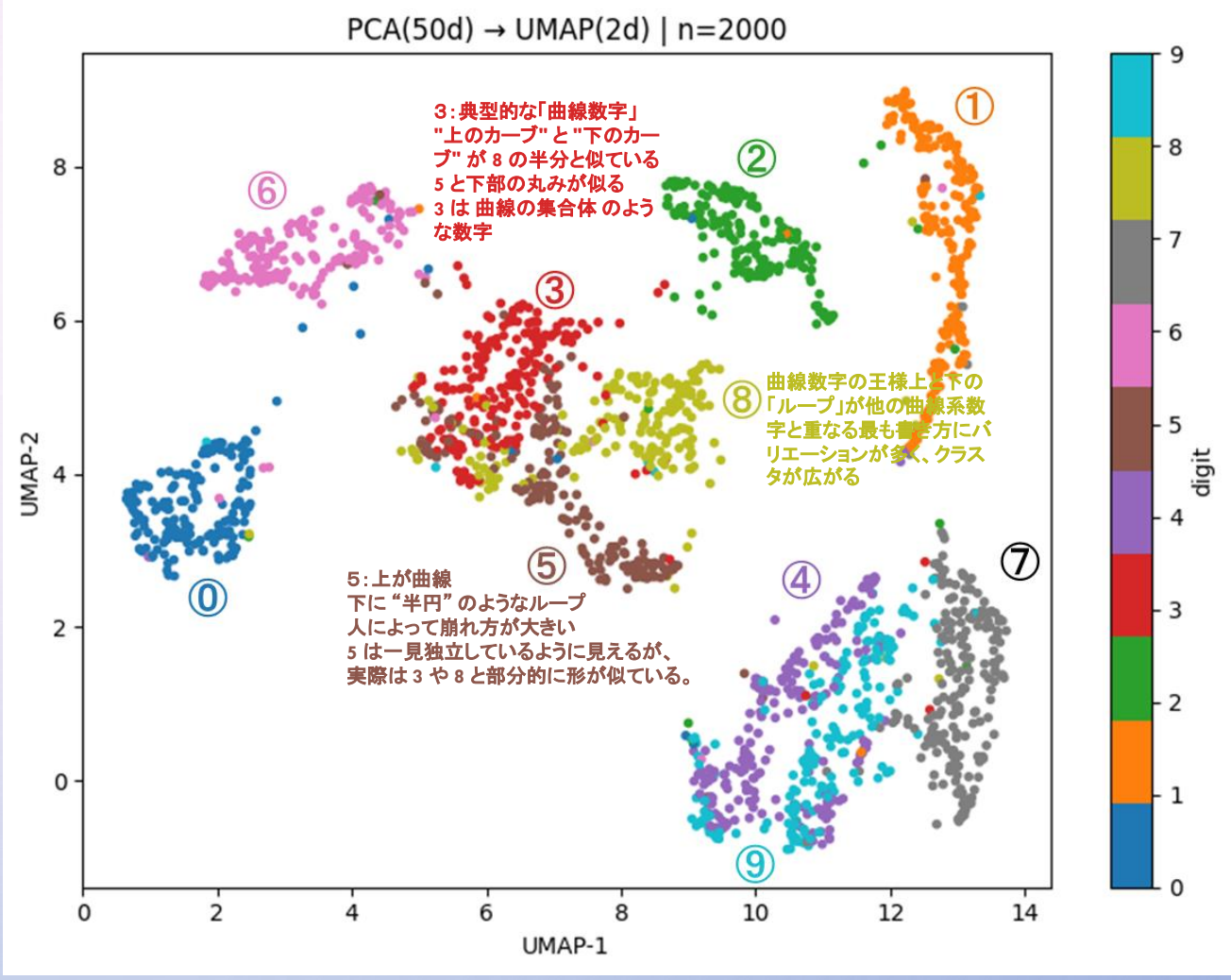
UMAP: ラベル別分析（1）

- クラスタが最も綺麗に分離している数字
- 1（オレンジ）
- 0（青）
- 6（ピンク色）
- これらは"形が特徴的で、書き手による揺らぎが少ない数字"です。
- 1：縦一本なのでバリエーションが少ない → 直線的な縦の島
- 0：ほぼ円形 → 中央密度の高い塊
- 6：上のループ + 下の円弧 → 分離しやすい
- これらの数字は、実際のモデルでも 誤分類率が低い 傾向があります。



UMAP: ラベル別分析(2)

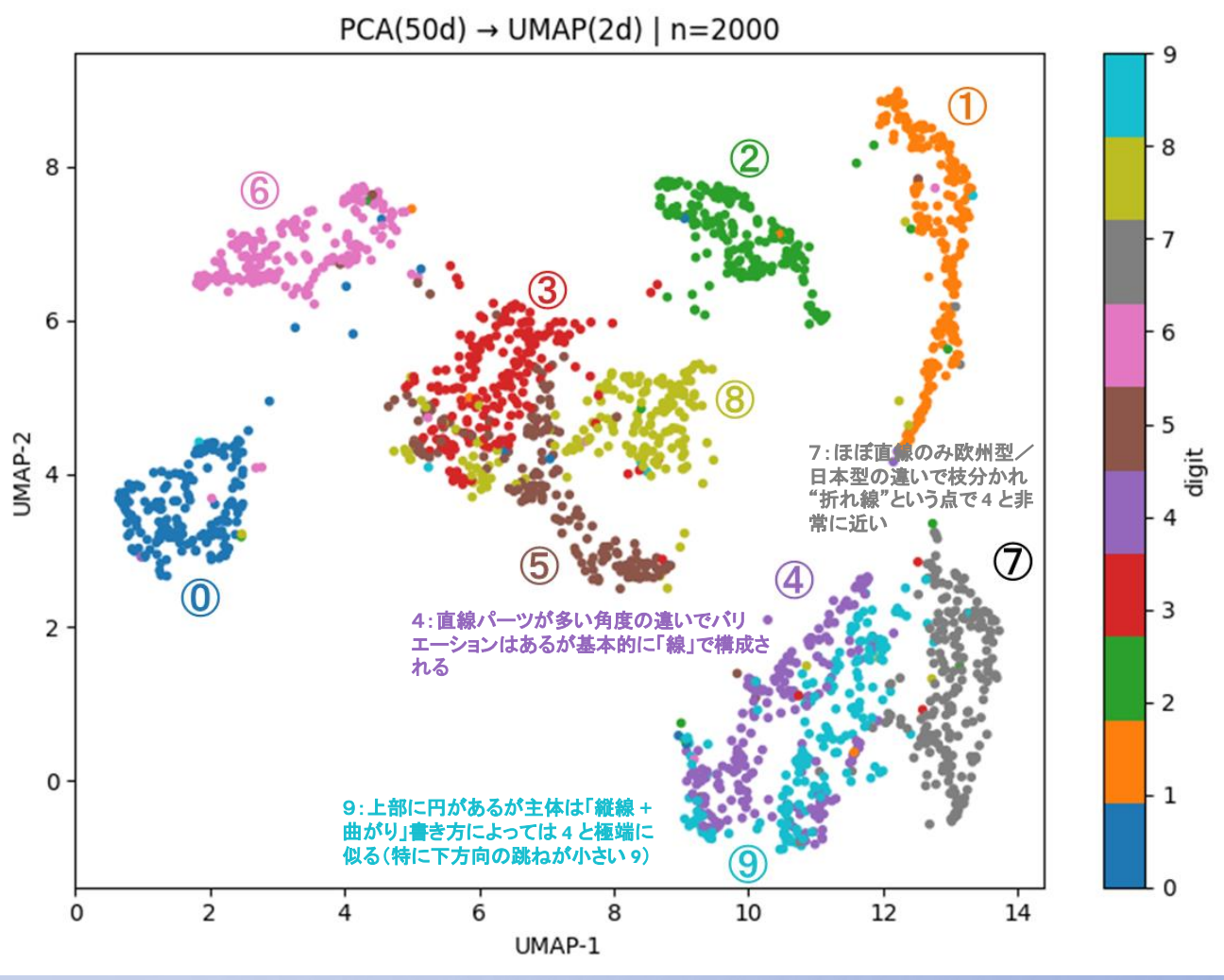
- なぜ 5・3・8 は一つの"巨大な島"になるのか？
- ✓ 共通点：曲線・丸み・ループ構造の有無
- UMAP は曲線の形状を低次元でまとめようとします。
- 5 → 下の曲線が強い
- 3 → 上下の曲線
- 8 → 二重の曲線
- つまり
  → 「曲線の量」が主成分
  → 曲線の多い数字同志は自然に近く配置される
- ✓ 書き方の揺らぎ(個人差)が強い
- → 曲線系は筆跡によって形が大きく変わる
  → そのぶんクラスタが散らばりやすい
  → 結果的に"曲線系の大クラスタ"として広がる



UMAP: ラベル別分析（3）

- なぜ 4・7・9 が“第二グループ”になるのか？
- ✓ 共通点：直線・折れ線構造の多さ
- 4 → 完全に直線主体
- 7 → 直線主体
- 9 → 直線 + 少し曲線
- → 曲線成分より 直線成分の寄与が大きい数字
- UMAP は直線的な筆跡を一箇所にまとめようとし、結果として 直線系の大ブロック が生まれます。

- ✓ 9 が曲線グループではなく「直線系」に入る理由
- 9 は一見“6 の逆”で曲線に見えるけど、手書きだと：
- 下が伸びて「4 のようになる」書き方が存在
- 上部のカーブより 縦線の比率が強い
- 直線成分と角度成分でクラスタ化されやすい
- → 4・7 との距離が近くなる



なぜこの2グループは“離れて”見えるのか？UMAP は高次元（784次元）空間の“局所的な距離関係”を保ちながら圧縮します。

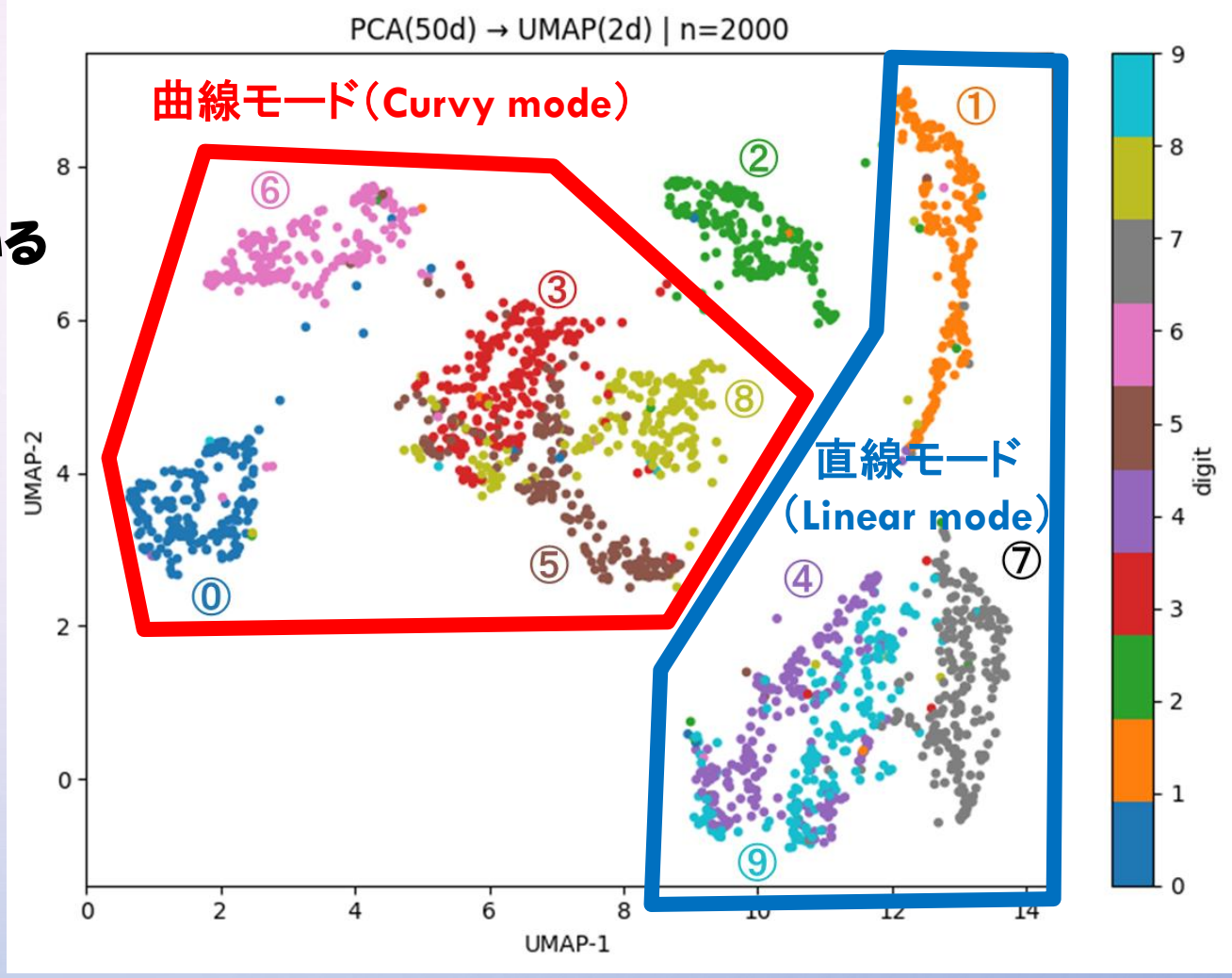
- 曲線系（5・3・8）→「丸み・膨らみ」が多い → 曲線特徴が多い
- 直線系（4・7・9）→「角・折れ線・直線」が多い → 直線特徴が多い

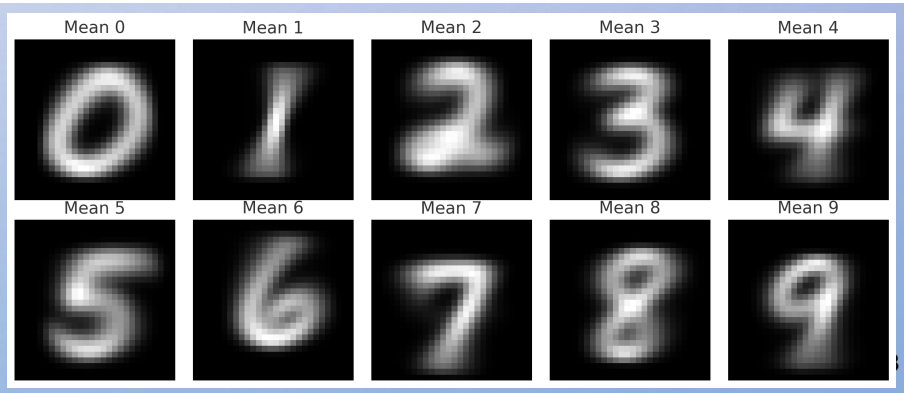
この2つは 筆記特徴が完全に異なる ため、UMAP の距離関数が自然と 二大グループの距離を広げる のです。

UMAP: ラベル別分析(4)

- 実は MNIST は 隠れた2つの書字モード を持っている

- 曲線モード(Curvy mode)
- 5・3・8
- 0(青)・6(ピンク)もやや近い
- 直線モード(Linear mode)
- 4・7・9
- 1(オレンジ)もここに近い仲間
- UMAP はこれを視覚的に“発掘”してくれたわけです。





これらのEDAを元に、ChatGPT環境で実行出来る範囲予測し、、submission.csvを作成して

- 何をやったか(ざっくり)
- アップロード済みの train.csv / test.csv を読み込み
- 画素値 pixel0～pixel783の各ピクセル値(0～255)を 0～1 にスケーリング
- 多クラス ロジスティック回帰(multinomial logistic regression) を学習
  - solver="lbfgs", max_iter=100
  - 収束警告は出たけど、実用レベルの係数は学習できている
- 学習モデルで test.csv に対して予測
- Kaggle Digit Recognizer 用形式の submission.csv を生成

mnist_logreg_0.91892.py

このモデル以外は60秒制限に引っかかり実行出来ず・・・

- スコア:0.91892

0.91892 は“ロジスティック回帰としては非常に妥当で、かなり良いスコア”です。

他モデルと比較（参考）

Digit Recognizer の代表的ベースラインを並べると:

モデル	典型スコア（Kaggle LB）
ロジスティック回帰	**0.91～0.93**
KNN (k=3～7)	0.96～0.97
ランダムフォレスト	0.94～0.96
SVM (RBF)	0.97～0.98
シンプルな1層MLP	0.97～0.98
CNN（LeNet相当）	0.99 付近
高精度CNN	0.995～0.999

Kaggleで実行出来る、最高スコアねらいの高精度CNNプログラムを作成して

- 方針ざっくり
- TensorFlow / Keras を使用（Kaggle GPUでそのまま動作）
- 28×28 グレースケール → (28, 28, 1) に reshape
- 0〜255 → 0〜1 にスケーリング
- StratifiedKFold 5分割で学習し、各foldごとに test を予測 → 平均アンサンブル
- Data Augmentation：回転・平行移動・ズームなど軽め
- CNN構造：
  - Conv-BN-ReLU ×2 → MaxPool → Dropout × 3ブロック
  - Dense(256) + BN + Dropout

この場合のKaggle GPU選択方法

おすすめ：GPU T4 × 2
代替：GPU P100
TPU v5e-8：使わない方がいい（このコードのままでは非対応）

- なぜ T4×2 / P100 で良いのか？
- ✅ 今のコードの特徴
- TensorFlow + Keras の 普通のGPU用コード
- tf.distribute.MirroredStrategy() などは使っていない
  → 1枚のGPUとして使われる（T4が2枚あっても1枚しか使わない）
- データセットは MNIST 相当で かなり軽い
  → 1枚の T4 でも十分高速（数分〜十数分レベル）
- 🔸 T4×2 vs P100
- P100 の方が1枚あたりの生性能はやや高いですが、
  MNIST＋この程度のCNNでは 体感差はほぼなしです。
- Kaggle だと「待ち時間／空き状況」の方が影響大きいので、
  空いている方（T4×2 か P100）を選べばOK と思って大丈夫です。

- なぜ TPU v5e-8 はやめた方がいいか？
- 現在のコードは TPU 対応の構造になっていない ため、
  そのままでは動きません。
- TPU を使うには：
  - tf.distribute.TPUStrategy の設定
  - tf.data.Dataset への書き換え
  - model.fit の扱い調整
    …など、コードの作りを結構変える必要があります。
- MNIST クラスの軽さなら、GPUで十分すぎるパフォーマンスが出るので
  TPUを使うメリットがほぼない です。

digit_cnn_5fold_0.99564.py

digit_cnn_5fold_0.99564.py

Digit Recognizer のスコア帯（経験則）

Digit Recognizer（MNIST）の LB スコアはざっくり次のようになります：

スコア：0.99564、実行時間15分程度 @ T2×2

モデル	典型スコア
ロジスティック回帰	0.91〜0.93
ランダムフォレスト / SVM	0.96〜0.98
シンプルCNN（1〜2層）	0.990〜0.995
中規模**CNN**（今回）	**0.995〜0.997**
かなり最適化したCNN	0.997〜0.998
上位0.1%（Ensemble, heavy aug）	0.9985〜0.9995

なぜ 0.995 初台が「妥当」なのか？

- 1. CNN でも一部の数字は非常に似ている
- 例として：
- 3 と 5
- 4 と 9
- 7 と 2
- 8 と 3/5/9
- このあたりは“画像そのものの曖昧さ”のため、モデルに非がなくても誤分類します。
- 特に ●4 vs 9 は MLP～CNN でも一番ミスが多いペア。
-

- 2. Kaggleの Digit Recognizer は「MNISTより難化」している
- MNIST より：
- ぼやけた数字
- 書き損じ風の数字
- 画素分布の偏り
- ラベルノイズ
- が追加されており、100%に近づくほどとんでもなく難しい です。

- 3. 5-fold アンサンブルは十分に強力な手法
- fold 平均でも 0.995～0.997 に落ち着きやすいです。
-

- 📊 0.99564 の順位感
- Kaggle 全体の順位でいえば：
- 上位5%～2% 前後
- スコア帯としては 優秀
- 上位 0.1%（0.9985～）は非常に強い時間をかけた大規模アンサンブルになります。
- あなたの 0.99564 は
  現実的な計算時間で到達できる '理想的なバランススコア' です。

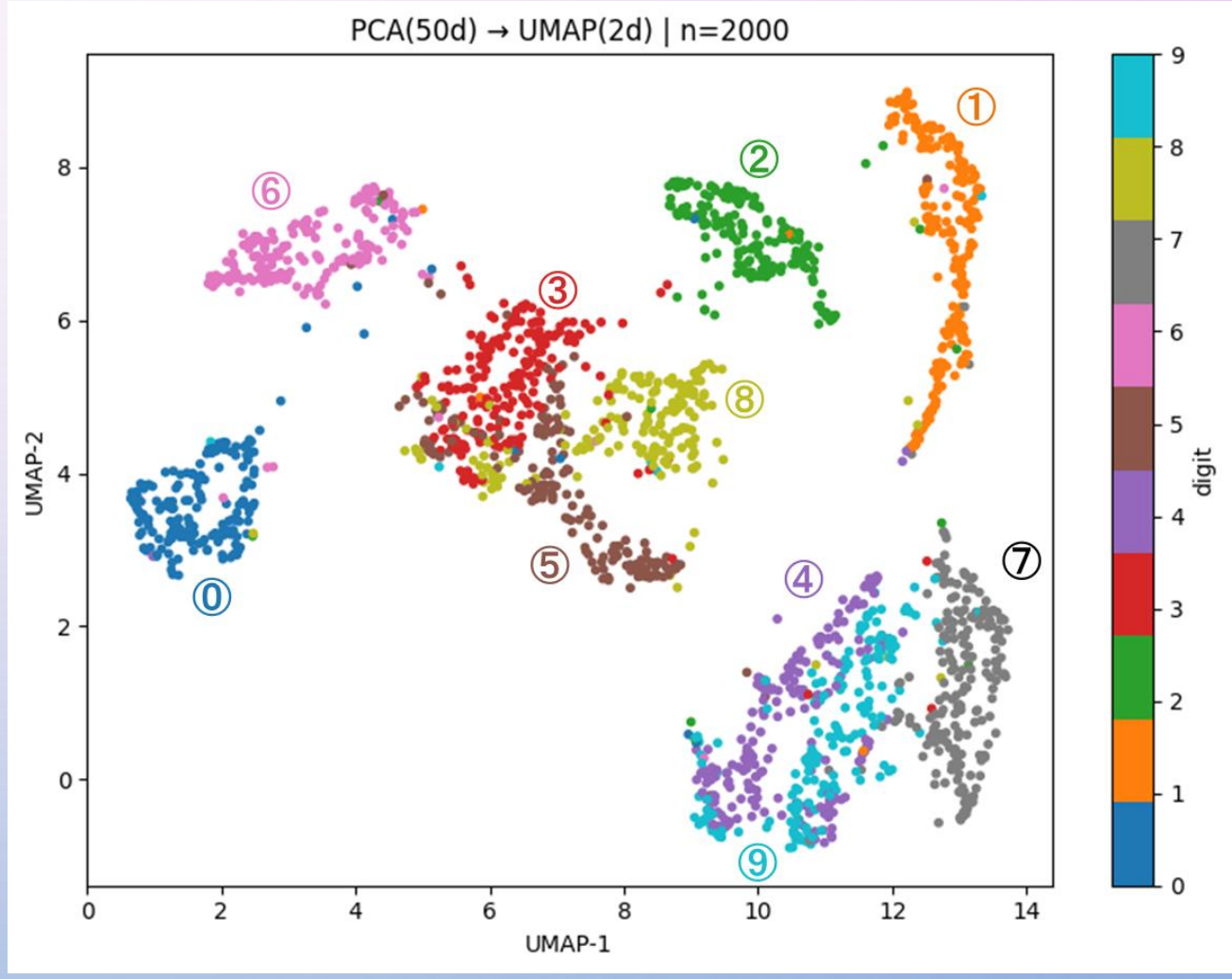
改善して 0.996～0.998台を狙うなら？

- さらに上を目指すなら：
- ✓ Mixup / CutMix
- ✓ Cutout
- ✓ DropBlock
- ✓ CosineAnnealing + WarmRestarts
- ✓ Conv層のチャンネル倍増
- ✓ 5-fold → 10-fold
- ✓ モデル2～3種類アンサンブル（CNN + EfficientNetB0など）
- などがありますが、計算コストが跳ね上がります。

先ほどのEDAを利用してみる！

umapで得られたEDA知見をもとのスコアアップ改良して

- ① 0 クラスは孤立しており非常に判別しやすい
  - → augmentation を強くかける必要なし
    → 誤分類率は元々低いので触らなくてよい
- ② 1・7・9・4 が互いに近く、境界が重なりやすい
  - 組み合わせ　理由
  - 1 と 7　　書き方が似ている。UMAP でも近接。
  - 4 と 9　　UMAP でクラスタが斜めに広がり接触。
  - 7 と 9　　右上方向に伸びてクラスタ同士が近い。
  - → ここが誤分類の主要ソースで、改善余地が最大。
- ③ 3・5・8 の三つ巴クラスター
  - 画像でも 3(赤)を中心に 5(茶)と 8(黄)が寄っている。
  - → 3 と 5 と 8 の境界は曖昧。
    → CNN がここを間違えやすいのは EDA と完全一致。
- ④ 6 はやや広がり気味の形状で境界が曖昧
  - → 6 のばらつきを吸収するためには deformation 系 augmentation が有効。



0.99564 → 0.99582

- 【改良施策 1】クラスごとに Data Augmentation を変更
  - 強めの augmentation を入れるクラス: 1, 4, 7, 9, 3, 5, 8, 6
  - 弱め(従来のまま or 少なめ)で十分なクラス: 0, 2
- 【改良施策 2】クラス別重み(Class Weight)
  - 6 → 1.2,　3,5,8 → 1.1,　　1,7,9,4 → 1.05,　0,2 → 1.0
- 【改良施策 3】Focal Loss で hard class を救う
  - 「境界付近の弱いサンプルだけ強調」→ 誤分類しやすい 1/7/9 や 3/5/8 に強い。

digit_cnn_5fold_umap_classaug_0.99582.py

スコア:0.99582、実行時間30分程度 @ T2×2

【改良施策 1】クラスごとに Data Augmentation を変更

- UMAP 的に「形状が近くて間違えやすいクラス」に強い augmentation を入れると精度が改善します。

0, 2 : UMAP 上でわりと孤立していて簡単 → 弱めの Aug

1,4,7,9 : 互いにクラスタが近く、混同しやすい → 強めの shift / rotation / zoom

3,5,8 : 中央部分の形が似ており、3 を中心に混ざりやすい → 強め

6 : UMAP でばらつきが大きいクラス → shear も入れて形の変化に強くする

```
# 0: ほぼ真ん中の丸い形 → ちょっとの回転/平行移動/拡大のみ
0: dict(
    rotation_range=5,         # 最大 ±5° だけ回転
    width_shift_range=0.05,   # 横方向に最大 5% シフト
    height_shift_range=0.05, # 縦方向に最大 5% シフト
    zoom_range=0.05           # 拡大縮小を ±5% だけ
),

# 2: 若干のバリエーションを許容したいので 0 より少し強め
2: dict(
    rotation_range=8,
    width_shift_range=0.08,
    height_shift_range=0.08,
    zoom_range=0.08
),
```

```
# 6: 形状が diffuse（ばらけている）クラス
# → shift/zoom をさらに強め、せん断(shear)も入れて形の変化に頑強にする
6: dict(
    rotation_range=20,
    width_shift_range=0.20,
    height_shift_range=0.20,
    shear_range=10,           # せん断変換（斜めにゆがめる）を最大 10°
    zoom_range=0.20
),
```

```
# 1,4,7,9: 書き方が似ていて UMAP でも近い
# → しっかり回転/シフト/ズームさせて、
#   境界の形を多めに学習させる
1: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
4: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
7: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
9: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
```

```
# 3,5,8: 丸みの部分が似ており、
#        UMAP 上でも重なりやすい
# → こちらも強めの Aug で
#   境界をはっきりさせる
3: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
5: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
8: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
```

学習時に、shiftやrotation, shearを各ラベル別に、ランダムにいれている

```
1: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
```

- ✓ digit=1 → "1専用"の Augmentation
- → rotate ±15°, shift ±15%, zoom ±15% をランダムで適用
- ✓ digit=6 → "6専用"の Augmentation
- → rotate ±20°, shift ±20%, zoom ±20%, shear ±10° をランダムで適用
- ✓ digit=0 → "0専用"の Augmentation
- → rotate ±5°, shift ±5%, zoom ±5% の弱めAugをランダム適用

- 例：digit=7 の画像1枚に対して
- ある epoch では 12° 回転 + 横シフト
- 次の epoch では 3° 回転 + zoom
- 次の epoch では -8° 回転 + 縦シフト
- のように 毎回ランダムで違う変形が行われます。

```
7: dict(
    rotation_range=15,
    width_shift_range=0.15,
    height_shift_range=0.15,
    zoom_range=0.15
),
```

なぜ「ラベル別 Augmentation」が効くのか？

- ✓ UMAP や PCA の分布で「混同しやすいクラス」が分かっていた
  - 1 vs 7
  - 3 vs 5 vs 8
  - 4 vs 9
  - 6 がバラバラに広い
  - これらのクラスは より強い Augmentation をかける必要がある。
- ✓ 逆に、0 や 2 のように“単純で安定した数字”に強すぎる Aug をかけると逆効果
  - そこで:
  - 境界が曖昧な digit → 強い Aug
  - 形が安定している digit → 弱い Aug

【改良施策 2】クラス別重み（Class Weight）

- UMAP（密集とばらつき）を反映すると、ばらつきが大きいクラスには わずかに重みを上げると精度が伸びる。
- 例：
- 6 → 1.2
- 3,5,8 → 1.1
- 1,7,9,4 → 1.05
- 0,2 → 1.0

```
# ------------------------------------------------
# 6. クラスごとの重み（class_weight）
#    → UMAP でばらつきや境界の曖昧さを見て、
#       難しいクラス(6, 3,5,8...) に少しだけ重みを大きくする。
#    → Keras の generator では class_weight 引数が使えないので、
#       あとで sample_weight に変換して使う。
# ------------------------------------------------
class_weight = {
    0: 1.00,  # 簡単
    1: 1.05,
    2: 1.00,  # 簡単
    3: 1.10,  # 3,5,8 は混同しやすい
    4: 1.05,
    5: 1.10,
    6: 1.20,  # 最もばらつきが大きい → 重みを高めに
    7: 1.05,
    8: 1.10,
    9: 1.05,
}
```

【改良施策 3】Focal Loss で hard class を救う

- 「境界付近の弱いサンプルだけ強調」→ 誤分類しやすい 1/7/9 や 3/5/8 に強い。

```python
# ----------------------------------------
# 3. Focal Loss
#    → 簡単なサンプルの影響を弱め、間違えやすいサンプルを強調する損失
# ----------------------------------------
def focal_loss(gamma=2.0, alpha=0.25):
    """
    y_true: (batch,) 形式のクラスラベル（0〜9）
    y_pred: softmax 出力 (batch, num_classes)

    ・普通の categorical_crossentropy に
      (1 - p_t)^gamma という重みをかけることで、
      すでに正しく分類できているサンプル（p_t が大きい）を軽く、
      間違えがちなサンプル（p_t が小さい）を重く扱う。
    """
    def loss(y_true, y_pred):
        # one-hot に変換
        y_true_onehot = tf.one_hot(tf.cast(y_true, tf.int32), depth=NUM_CLASSES)
        # 通常のクロスエントロピー
        ce = keras.losses.categorical_crossentropy(y_true_onehot, y_pred)
        # p_t = 正解クラスに対応する確率
        p_t = tf.reduce_sum(y_true_onehot * y_pred, axis=-1)
        # Focal の重み (1 - p_t)^gamma * alpha
        focal_factor = alpha * tf.pow(1.0 - p_t, gamma)
        return focal_factor * ce
    return loss
```

Focal Loss が効く理由

MNISTの約 85% は簡単な画像(easy class)。
普通の cross entropy(CCE)で学習すると
CCE は「簡単な画像の損失も大きく扱ってしまう

- CCE(Cross Entropy) はすべてのサンプルを等しく扱います。
- 例えば:
  - 簡単な 0 の画像(予測 0.999 正解) → 損失 ≈ 0.001
  - 難しい 7 の画像(予測 0.6 正解) → 損失 ≈ 0.5
- CCE は
  難しい画像の損失が Easy の100～500倍 なのですが…
- データ数の比率では Easy が大量にあるため、
  全体の重みは「簡単なサンプル」側に傾いて学習してしまう。
- → Hard class の改善が弱い
  → Hard class の誤分類が最後まで残る
  → 精度が 0.996 に届かない
- という MNIST 典型パターン。

クラス	特徴
0,1,2	ほぼ間違えない(Easy class)
1 vs 7	境界が近い(Hard)
3 vs 5 vs 8	似ている(Hard)
6	形がバラつき大(Hard)
4 vs 9	書き方が似る(Hard)

- CCE は「簡単な画像の損失も大きく扱ってしまう。CCE(Cross Entropy) はすべてのサンプルを等しく扱います。
- 例えば:
  - 簡単な 0 の画像(予測 0.999 正解) → 損失 ≈ 0.001
  - 難しい 7 の画像(予測 0.6 正解) → 損失 ≈ 0.5
- CCE は
  難しい画像の損失が Easy の100～500倍 なのですが…
- データ数の比率では Easy が大量にあるため、
  全体の重みは「簡単なサンプル」側に傾いて学習してしまう。
- → Hard class の改善が弱い
  → Hard class の誤分類が最後まで残る
  → 精度が 0.996 に届かない
- という MNIST 典型パターン。

MNIST の誤差は「Hard class が10～30枚残る」ことによる

- 0.996 → 約 20枚 の誤分類
  0.995 → 約 35枚 の誤分類
- 誤分類のほぼ全てが以下：
  - 7→1
  - 1→7
  - 3↔5
  - 8↔3
  - 4↔9
  - 6→5/0
- MNIST は“簡単な数字のミス”はほとんどないため、
  この 少数の Hard class だけ強化できる Focal Loss が相性バツグン
- Focal Loss とクラス別 Augmentation の相性が抜群
- 今回はさらに：
  - Hard class → Aug 強め
  - Easy class → Aug 弱め
  - Hard class → Focal で損失大
  - Easy class → Focal で損失小
- という組み合わせになっていたため、
  Hard class 専用ブースト回路 になっていた。
- だから：
- CCE + 通常Aug → 0.994～0.995
- FocalLoss + ClasswiseAug → 0.9958
- まで伸びた。

まとめ

- ✓ ロジスティック回帰: 0.91892、CNN: 0.99564 → GPU使ったCNN等が必須！
- ✓ Focal Loss は“間違えやすい数字の学習を強化する”
- ✓ MNIST には Hard class が少数だけ存在する
- ✓ CCE だと Easy class に学習が奪われる
- ✓ Focal Loss は Easy class を無視して Hard class に集中
- ✓ Classwise Augmentation と組み合わせると強烈に効く
- ✓ 結果：0.99582 近くまでスコア上昇した → EDA大事！

Algorithm	LB	Environment
mnist_logreg	0.91892	ChatGPT
digit_cnn_5fold	0.99564	Kaggle T2x2
digit_cnn_5fold_umap_classaug	0.99582	Kaggle T2x2