記事
Toshihiko Minamoto · 2021年11月25日 17m read

HealthShareにバインディングしたPython 3を使用したディープラーニングデモを実行する（パート2）

キーワード: Jupyterノートブック、TensorFlow GPU、Keras、ディープラーニング、MLP、HealthShare

1. 目的

前回の「パート1」では、ディープラーニングデモ環境をセットアップ
しました。今回「パート2」では、それを使ってできることをテストします。

私と同年代の人の中には、古典的なMLP（多層パーセプトロン）モデルから始めた人がたくさんいます。
直感的であるため、概念的に取り組みやすいからです。

それでは、AI/NNコミュニティの誰もが使用してきた標準的なデモデータを使って、Kerasの「ディープラーニングMLP」を試してみましょう。 いわゆる「教師あり学習」の一種です。
これを実行するのがどんなに簡単かをKerasレベルで見ることにします。

後で、その歴史と、なぜ「ディープラーニング」と呼ばれているのかについて触れることができます。流行語ともいえるこの分野は、実際に最近20年間で進化してきたものです。

HealthShareにも関連しているため、最終的には、少々実現的なユースケースを想像または予測できるようになることを願っています。

2. 範囲と免責事項

次のことを行います。

- tensorflow-gpu環境用に新しいJupyterカーネルをセットアップします。
- ANNコミュニティで一般的な標準のMNISTサンプルを使って、Keras MLPモデルを定義、トレーニング、および検証（テスト）します。
- 重要なパラメーターのほんの一部の非常に単純なものを簡単に説明します。
- デモデータを簡単に調べます。データを理解することは、あらゆる実験において常に重要なことです。
- データサンプルをCache/HealthShareに保存し、予測（分類）と推論を行うために読み取り直す作業がどれほど簡単であるかを実演します。

その後で、テストサンプルを少し回転させ、トレーニング済みのモデルをどれくらい混乱させられるかを確認し、それによって明確な制限を理解します。

学術的・数学的な部分は省略しますが、仕組みについて簡単に説明するところもあります。

免責事項: MNISTデータサンプルは、このデモの目的で公開されています。
ほとんどのデモコードは最小限に縮減されており、エラー処理の含まれないベアなコードでした。
Kerasコードのソースは「謝辞」に記載されています。 この内容は、必要に応じていつでも変更されます。

3. 前提条件

前の「パート1」の記事
に記載されているとおりにデモ環境をセットアップする以外で、以下の実験のための前提条件はありません。

4. Jupyterノートブックのセットアップ

前回インストールした「tensorflow-gpu」環境で次のコマンドを実行しました。

(tensorflow-gpu) C:\>conda install ipykernel
Solving environment: done

... ...

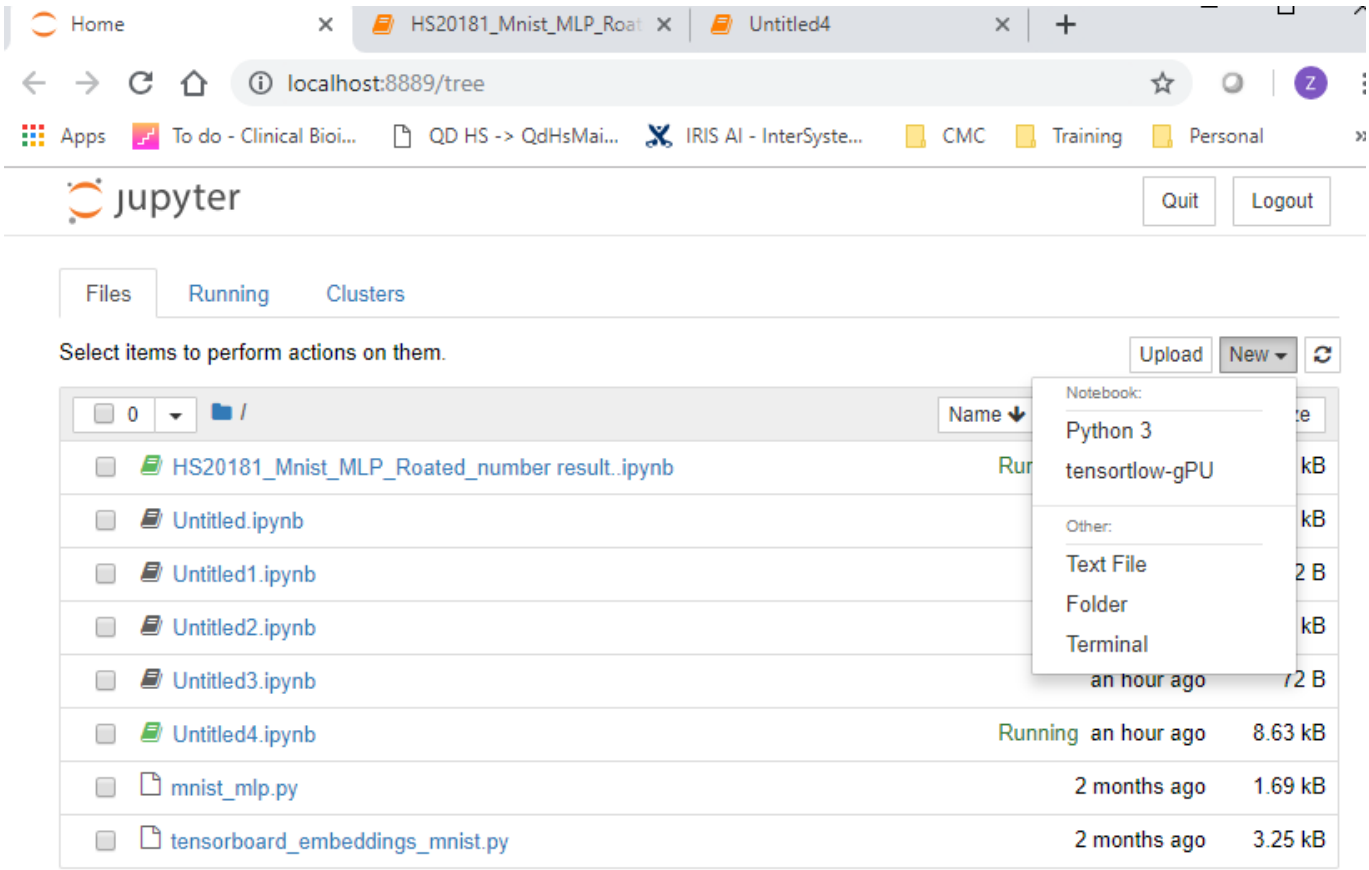
(tensorflow-gpu) C:\>python -m ipykernel install --user --name tensorflow-gpu --display-name "Tensorflow-GPU"
Installed kernelspec tensorflow-gpu in C:\Users\zhongli\AppData\Roaming\jupyter\kernels\tensorflow-gpu

こうすることで、「Tensorflow-GPU」などと呼ばれる新しいJupyterカーネルを作成しました。

これで、Anaconda Promptから次のようにしてJupyterノートブックを起動できるようになりました。

(tensorflow-gpu) C:\anaconda3\keras\Zhong>jupyter notebook
[... ...
[I 10:58:12.728 NotebookApp] The Jupyter Notebook is running at:
[I 10:58:12.729 NotebookApp] http://localhost:8889/?token=6b40f6e6749e88b80a338eec3330d06c181ead9b644...
[I 10:58:12.734 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 10:58:12.835 NotebookApp] ... ...

ブラウザのUIが以下のように起動していることを確認できます。
［New］をクリックすると、新しい「TensorflowGPU」タブが開きます。



5. ディープラーニングMLPモデルのトレーニング

標準のディープラーニングMLPモデルデモを試してみましょう。

5.1 環境のテスト

Jupyterタブを「MLPDemo_HS」のような名前に変更し、そのセル [1]
で以下の行を実行して「**Run**」をクリックします。

```
print('Hello World!')
```

```
Hello World!
```

5.2 PythonからHealthShareへの接続のテスト

セル[2]でPythonサンプルを実行し、まだ「パート1」の記事
に記載されたとおりにHealthShareデータベースインスタンスに接続できるかをテストします。

```
import codecs, sys
import intersys.pythonbind3

try:
  print ("Simple Python binding sample")

port = input("Cache server port (default 56778)? ")
  port = port.rstrip()
  if (port == ""):
  port = "56778"

url = "localhost["+port+"]:SAMPLES"
  print ("Connection string: " + url)

print ("Connecting to Cache server")
  conn = intersys.pythonbind3.connection( )
  conn.connectnow(url, "SYSTEM", "SYS", None)
  print ("Connected successfully")

print ("Creating database")
  database = intersys.pythonbind3.database( conn)

print ("Opening Sample.Person instance with ID 1 with default concurrency and timeout")
  person = database.openid( "Sample.Person", "1", -1, -1)

print ("Getting the value of the Name property")
  name = person.get("Name")
  print ("Value: " + name)

print ("Test completed successfully")
except intersys.pythonbind3.cacheexception( err):
  print ("InterSystems Cache' exception")
  print (sys.exctype)
  print (sys.excvalue)
  print (sys.exctraceback)
  print (str(err))
```

```
Simple Python binding sample
Cache server port (default 56778)?
Connection string: localhost[56778]:SAMPLES
Connecting to Cache server
Connected successfully
Creating database
Opening Sample.Person instance with ID 1 with default concurrency and timeout
Getting the value of the Name property
Value: Zevon,Mary M.
Test completed successfully
```

5.3 説明 - MLPモデルのトポロジーとMNISTデータセット

MLPネットワークのトポロジーは、以下に示すとおり単純です。
通常、1つの入力と1つの出力のレイヤーがあり、多数の非表示レイヤーがあります。

各レイヤーには多数のニューロン（ノード）があります。 各ニューロンには活性化関数があります。 以下のように、2つの異なるレイヤーのニューロン間に完全にメッシュ化された接続（「密度」モデル）が存在することになります。

これに対応し、以下でテストしているKeras MLPモデルには次のものが含まれます。

- **28 x**
  28ノードの計784ノードの入力レイヤー
  （よ
  うするに2
  8x28ピクセルの小さ
  な画像であり、それぞれは「0」から「9
  」の手書きの数字です。
  MNISTデータセットにはこのような画像がトレーニング用に60,000個、テスト用に10,000個含まれています）
- **10ノードの出力レイヤー**（0から9の間の入力画像の分類結果を表します）
- **2つの非表示レイヤー**（各レイヤーには512個のノードがあります）

これが、このデモモデルの主要トポロジーです。
他の詳細については今のところは省略して、実際に実行することにしましょう。

5.4. GoogleパブリッククラウドからMNISTサンプルデータを読み込む

では、上記のモデルをまったく最初から作成し始めることにしましょう。KerasパッケージとMNISTデータをJupyter Cellにロードして、 メニューの［Run］ボタンをクリックします。

```
### Import Keras modules
import keras
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import RMSprop

### Define key training parameter
batchsize = 128   # weights adjusted in 128 steps
numclasses = 10   # 10 classification results on the output layer
epochs = 20  # run the set of samples 20 times.
```

###load the data from Google public cloud
load the MNIST sample image data, split between train and test sets
(xtrain, ytrain), (xtest, ytest) = mnist.loaddata()

注意: 問題がある場合は、例外の内容に従うか（ほとんどの場合が、パッケージが見つからないといった例外です）、Googleで答えを探すか（99%の確率で回答を得られます）、以下に質問を投稿してください。

コードの最後の行で、60,000個と10,000個の全データセットが3次元整数のPython配列にロードされました。
トレーニングサンプルの1つをHealthShareデータベースに読み込んでみてみましょう。

5.5 データサンプルをHealthShareのグローバルに読み込む

HealthShare -> SAMPLESネームスペース - >
Sample.Person.clsで、この最も単純なクラスメソッドをスクラッチします。

```
ClassMethod SetTrainGlobals(d1 As %Integer = , d2 As %Integer = , value As %String = "", target As %String = "") As %BigInt [ SqlProc ]
{
Set ^XTrainInput(d1, d2) = value
Set ^YTrainTarget(d1) = target
return $$$OK
}
```

1つの入力トレーニングサンプルを文字列としてグローバル^XTrainInputに取り込み、入力トレーニングターゲットを^YTrainTargetに保存します。

リコンパイルし、セクション5.2に従って接続を更新してから、以下のようにPythonのセルから呼び出しを実行します。

```
result1 = person.runobjmethod("SetTrainGlobals", [0, 2, str(xtrain[0]), str(ytrain[0])])
```

HealthShare -> Samplesで、^XTrainGlobal(0, 2)
というグローバルが2次元整数の文字列で作成されたことがわかります。

後で、別の単純なメソッドを実行し、データをサンプルとしてPython変数に読み戻すことができます。

5.6 モデルを実行してトレーニング実行する

JupyterでのMLPモデルの定義とトレーニングを完了しましょう。

基本的に以下のコードの「**reshape**」は、それぞれの28 x 28のサンプルを0から255の値の1x
784個の値に変換し、その後で0から1.0の浮動小数点型に正規化します。

Model.Sequentialから**Model.Summary**までのコードは、784 x 512 x 512 x
10ノードのMLPを、「relu」活性化関数を使って定義するするものです。

最後に、 **model.fit**でトレーニングし、**model.evalute**でテスト結果を評価します。

```
xtrain = xtrain.reshape(60000, 784)
xtest = xtest.reshape(10000, 784)
xtrain = xtrain.astype('float32')
xtest = xtest.astype('float32')
xtrain /= 255
xtest /= 255

print(xtrain.shape[0], 'train samples')
```

```
print(xtest.shape[0], 'test samples')

# convert class vectors to binary class matrices
ytrain = keras.utils.tocategorical(ytrain, numclasses)
ytest = keras.utils.tocategorical(ytest, numclasses)

model = Sequential()
model.add(Dense(512, activation='relu', inputshape=(784,)))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(numclasses, activation='softmax'))

model.summary()

model.compile(loss='categoricalcrossentropy',
  optimizer=RMSprop(),
  metrics=['accuracy'])

history = model.fit(xtrain, ytrain,
  batchsize=batchsize,
  epochs=epochs,
  verbose=1,
  validationdata=(xtest, ytest))

score = model.evaluate(xtest, ytest, verbose=1)

print('Test loss:', score[0])
print('Test accuracy:', score[1])


Using TensorFlow backend.

60000 train samples
10000 test samples
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dropout_1 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
Train on 60000 samples, validate on 10000 samples
Epoch 1/20
60000/60000 [==============================] - 11s 178us/step - loss: 0.2476 - acc: 0
.9243 - val_loss: 0.1057 - val_acc: 0.9672
Epoch 2/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.1023 - acc: 0.
9685 - val_loss: 0.0900 - val_acc: 0.9730
```

```
Epoch 3/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0751 - acc: 0.
9780 - val_loss: 0.0756 - val_acc: 0.9783
Epoch 4/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0607 - acc: 0.
9816 - val_loss: 0.0771 - val_acc: 0.9801
Epoch 5/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0512 - acc: 0.
9844 - val_loss: 0.0761 - val_acc: 0.9810
Epoch 6/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0449 - acc: 0.
9866 - val_loss: 0.0747 - val_acc: 0.9809
Epoch 7/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0377 - acc: 0.
9885 - val_loss: 0.0765 - val_acc: 0.9811
Epoch 8/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0334 - acc: 0.
9898 - val_loss: 0.0774 - val_acc: 0.9840
Epoch 9/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0307 - acc: 0.
9911 - val_loss: 0.0771 - val_acc: 0.9842
Epoch 10/20
60000/60000 [==============================] - 6s 105us/step - loss: 0.0298 - acc: 0.
9911 - val_loss: 0.1015 - val_acc: 0.9813
Epoch 11/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0273 - acc: 0.
9922 - val_loss: 0.0869 - val_acc: 0.9833
Epoch 12/20
60000/60000 [==============================] - 6s 99us/step - loss: 0.0247 - acc: 0.9
926 - val_loss: 0.0945 - val_acc: 0.9824
Epoch 13/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0224 - acc: 0.
9935 - val_loss: 0.1040 - val_acc: 0.9823
Epoch 14/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0219 - acc: 0.
9939 - val_loss: 0.1038 - val_acc: 0.9835
Epoch 15/20
60000/60000 [==============================] - 6s 104us/step - loss: 0.0227 - acc: 0.
9936 - val_loss: 0.0909 - val_acc: 0.9849
Epoch 16/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0198 - acc: 0.
9944 - val_loss: 0.0998 - val_acc: 0.9826
Epoch 17/20
60000/60000 [==============================] - 6s 101us/step - loss: 0.0182 - acc: 0.
9951 - val_loss: 0.0984 - val_acc: 0.9832
Epoch 18/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0178 - acc: 0.
9955 - val_loss: 0.1150 - val_acc: 0.9839
Epoch 19/20
60000/60000 [==============================] - 6s 100us/step - loss: 0.0167 - acc: 0.
9954 - val_loss: 0.0975 - val_acc: 0.9847
Epoch 20/20
60000/60000 [==============================] - 6s 102us/step - loss: 0.0169 - acc: 0.
9956 - val_loss: 0.1132 - val_acc: 0.9832
10000/10000 [==============================] - 1s 71us/step
Test loss: 0.11318948425535869
Test accuracy: 0.9832
```

以上で、「トレーニング済み」となりました。 ほんの数行のコードで、このKerasディープラーニングMLPは、「

tensorflow-gpu」環境でかなり効率的に実行します。 これまでのすべてのキットインストールを検証します。

6 サンプルを使用してモデルをテストする

以下の指定されたサンプルを使用してトレーニング済みのモデルをテストしましょう。

10,000個のxtestセットから特定のサンプルをランダムに選択し、別のHealthShareグローバルに保存してから、
そのグローバルからPython配列にデモサンプルとして読み戻します。
それを使用してトレーニング済みのモデルをテストします。

次に、この入力サンプルを90度、180度、および270度に回転させてモデルを再テストし、混乱が生じないかを確認します。

6.1 サンプルをHealthShareに保存する - デモ

サンプルをランダムに選択しましょう。たとえば、10,000個のテストサンプルから12番目のサンプルを選択し、HSグローバルに保存します。

HealthShare -> SAMPLE -> Sample.Personクラスに新しいクラスメソッドを追加します。

```
ClassMethod SetT2Globals(d1 As %Integer = 0, d2 As %Integer = 0, d3 As %Integer = 0, value As
%String = "", target As %String = "") As %BigInt [ SqlProc ]
{
 Set ^XTestInput(d1, d2, d3) = value
 Set ^YTestTarget(d1, d2) = target
 return $$$OK
}
```

Sample.Person.cls をリコンパイルします。 Jupyterノートブックで、セクション5.2のコードを再実行し、データベースのバインディングを更新します。次に、この行を実行して、28 x 28
個の数字サンプルをグローバル^XTestInputに保存します。

```
import re
n = 12 # randomly choose a sample
for i in range(0, len(xtrain[n])):
  r1 = person.runobjmethod("SetT2Globals", [1, n, i, re.sub('0 \s0', ' 0 0', str(xtest[n][i])), str(ytest[n])])
```

これで、2次元配列のサンプルがHS SAMPLEグローバル^XTestInputに保存されていることがわかりました。
それぞれの数字は、0〜255のピクセルグレースケールです。
以下のHS管理ポータルから、それが「**9**」であることが簡単にわかります。

6.2 HealthShareグローバルからサンプルを読み取る - デモ

HSデータベースグローバルからサンプルを確実に読み取ることができます。

別のクラスメソッドをSample.Person.clsに追加して、それをリコンパイルします。

```
ClassMethod GetT2Globals(d1 As %Integer = 0, d2 As %Integer = 0, d3 As %Integer = 0) As %String [ SqlProc ]
{
 Set value = ^XTestInput(d1, d2, d3)
 return value
}
```

Jupyterで、前述のようにDBバインディングを更新してから、このPythonコードを実行してHealthShareから文字列としてグローバルを読み取り、1 x 2次元数値配列に変換します。

```
import re, ast
sample = ""
for i in range(0, len(xtrain[n])):
  sample += person.runobjmethod("GetT2Globals", [1, n, i])
#convert it to numpy ndarray
as1 = np.array(ast.literaleval(re.sub('\s+', ',', re.sub('0\]', '0', re.sub('\[ ', '', re.sub('\]\[', ' ', sample))))))
Sample12 = as1.reshape(1, 28, 28)
print(Sample12)
```

6.3 トレーニング済みモデルをテストする

これで、Jupyterで、この配列「Sample12」をトレーニング済みのモデルに送信できるようになりました。
model.predictやmodel.predictclassesがこの作業を行います。

```
Sample12 = Sample12.reshape(1, 784)
Sample12f = Sample12.astype('float32')/255 # normalise it to float between [0, 1]
Result12f = model.predict(Sample12f) #test the 1x784 sample, the result is a 1d matrix
print(Result12f)

Result12 = model.predictclasses(Sample12f) #test the sample, the result is a clasified lable.
print(Result12)
```

```
[[2.5672970e-27 1.1168821e-25 1.3736557e-20 6.2964843e-17 7.0107062e-09 6.2905544e-17
  1.5294099e-28 7.8019199e-17 3.5748028e-16 1.0000000e+00]]
[9]
```

結果には出力レイヤーのニューロン#9に最大値「1.0」があることが示されているため、分類結果は「9」となります。 これは正しい結果です。

確かに、人工的な28 x 28整数のサンプルをモデルに送信して試してみることができます。

6.4 サンプルを回転させてモデルを再テストする

このサンプルを反時計回りに90度回転させて、もう一度試してみることはできるでしょうか？

```
Sample12 = Sample12.reshape(28, 28) #reshape to 2D array values
Sample1290 = np.rot90(Sample12) #rotate in 90 degree
print(Sample1290)
```

次に、モデルを再テストします。

```
Sample12901 = Sample1290.reshape(1, 784)
Sample1290f = Sample12901.astype('float32')/255
Result1290f = model.predict(Sample1290f)
print(Result1290f)

Result1290 = model.predictclasses(Sample1290f)
```

print(Result1290)

```
[[2.9022769e-
05 1.2192334e-20 1.7143857
e-07 3.0004558e-11 2.4583075e-11 6.2443775e-01
 2.5749558e-05 3.7550735e-01 2.0722151e-08 5.5368415e-10]]
[5]
```

さて、**モデルは「5」と認識しました**。ニューロン#5が最大出力値でトリガーされています。
どうやら少し混乱しているようです！（確かに縦向きのものが「5」のように見えるので仕方ありませんね）。

では今度はサンプルを180度回転させましょう。モデルはどのように認識するでしょうか？

```
[[3.3131425e-11 3.0135434e-27 8.7524540e-23 7.1371946e-24 2.4029167e-13 4.2327470e-09
 1.0000000e+00 1.7086377e-18 1.3129146e-18 2.8595145e-22]]
[6]
```

もちろん、間違いなく「6」と認識しました！ 人間も「9」ではなく「6」と認識するでしょう。

では、最後に270度回転させてみましょう。

どうやら、また混乱しているようです。今度は「**4**」と認識しました。

```
[[1.6130849e-06 3.0311636e-
14 2.1490927e-03 2.7108688e-03 9.9499077e-01
 1.4130991e-04 6.2298268e-06 8.6649310e-09 2.9320630e-12 1.5710594e-07]]
[4]
```

6.5 パブリッククラウドツールとの比較

上記の配列をPythonコードの行を介してPNGにエクスポートし、回転して、反転して、画像上にまとめました。
次のようになっています。

次に、それぞれ「Google Vision API」、「Amazon Rekognition」、および「Microsoft Computer Vision API」にアップロードすると、結果はどうなるでしょうか。

この場合、AWSの「数字」のスコアが95%とわずかに最高となっています（これは絶対に代表的な結果ではありません）。

1. Google Vision APIの結果:

2. AWS Rekognitionの結果:

3. Microsoft Computer Visionの結果:

7. 次の内容

次は、以下のような他のいくつかの簡単なポイントについてフォローアップします。

- 一言で、MLPはどのように機能するのか？
- 制限と考えられるユースケースは？
- 現在のスタックで実行可能な最も一般的なML/DL/ANNモデルの簡単なウォークスルー
- 謝辞

#AI #Python #初心者 #機械学習 #HealthShare

ソースURL:
https://jp.community.intersystems.com/post/healthshare%E3%81%AB%E3%83%90%E3%82%A4%E3%83%B3%E3%83%87%E3%82%A3%E3%83%B3%E3%82%B0%E3%81%97%E3%81%9Fpython-3%E3%82%92%E4%BD%BF%E7%94%A8%E3%81%97%E3%81%9F%E3%83%87%E3%82%A3%E3%83%BC%E3%83%97%E3%83%A9%E3%83%BC%E3%83%8B%E3%83%B3%E3%82%B0%E3%83%87%E3%83%A2%E3%82%92%E5%AE%9F%E8%A1%8C%E3%81%99%E3%82%8B%EF%BC%88%E3%83%91%E3%83%BC%E3%83%882%EF%BC%89-%C2%A0