|  |
| --- |
| dcganパッケージ |
| 内容  [1 パッケージの概要 2](#_Toc499650675)  [2 サンプルプログラム 3](#_Toc499650676)  [2.1 example1.py（簡単な例） 4](#_Toc499650677)  [2.2 example2.py（学習パラメータの設定例） 6](#_Toc499650680)  [2.3 example3.py（オブザーバーの使用例①－画像出力） 7](#_Toc499650683)  [2.4 example4.py（オブザーバーの使用例②－ロールバック） 8](#_Toc499650684)  [2.5 example5.py（FeatureModelの使用例－Generatorの評価） 10](#_Toc499650685)  [2.6 example6.py（Discriminatorのカスタマイズ例①－tensorflowでの定義） 12](#_Toc499650686)  [2.7 example7.py（Discriminatorのカスタマイズ例②－Kerasでの定義） 13](#_Toc499650687)  [2.8 example8.py（データセットクラスの使用例） 14](#_Toc499650688) |

|  |  |
| --- | --- |
| パッケージの概要 |  |
|  | |
| パッケージ「dcgan」はDCGANを再利用可能にするために作成したパッケージである。ユーザビリティを最大限に考慮して、**Generatorの定義と学習データを準備すればDCGANの学習を実行できる**ように設計した。 | |
| パッケージには５つのモジュールが用意されており、それぞれ以下のように使用される。 |  |
| |  |  |  | | --- | --- | --- | | train.py | DCGANの学習を行うモジュールで、関数**fit**()または**fit\_dataset**()で学習を実行する  使用例はexample1、example2（fit()）またはexample8（fit\_dataset()）を参照 | | | BaseObserver.py | DCGANの学習を監視するObserverの基本クラス  このクラスの派生クラスでイベント関数をオーバーライドし、学習時の各イベントで独自の処理を行うことができる イベントは以下の２つ 　・on\_completed\_batch\_train() 　　ミニバッチの単位の処理完了時にコールされる 　　（return Falseで学習を終了できる）  　・on\_completed\_epoch\_train()  　　エポックの単位の処理完了時にコールされる （return Falseで学習を終了できる）  使用例はexample3またはexample4を参照 | | | FeatureModel.py | Generatorを用いて**画像の特徴量算出**や**画像のレプリカ生成**を行う（右図の偶数列がレプリカ）  使用例はexample5を参照 | C:\Users\mtanaka\Desktop\workspace\results\20171025_094811_sea_temp_dlr{2.0e-04}_glr{2.0e-04}\g_h5(15000)_epoch(100)_[adam(0.1)]___.png | | BaseDataSet.py | 学習データを管理するクラスの基本クラスで、インターフェースとしての役割しか持たない | | | DirectoryImageDataSet.py | BaseDataSetクラスの派生クラスで、バッチ処理ごとに指定されたディレクトリからPNGファイルをロードする。  使用例はexample8を参照 | | | |
| また、train.pyの大部分は以下のURLを参考としているので、内部を理解したい場合は、つぎのURLも合わせて参照することが望ましい。 <http://bamos.github.io/2016/08/09/deep-completion/#ml-heavy-dcgans-in-tensorflow> | |

|  |  |
| --- | --- |
| サンプルプログラム | |
| ここでは、サンプルプログラム（example1~8）の解説を行いながら、パッケージの使用方法を学習する。  サンプルプログラムでは、すべての学習データにCIFAR10を使用しているので、Kerasの環境が整っていればすぐにプログラムを実行できる。  また、このデータの画像は32×32のRGBが画像であるので、学習データのデータセットのshapeは(画像の枚数, 32, 32, 3)である。 |  |

|  |
| --- |
| example1.py（簡単な例） 最も簡単な実装で、フローは以下の通り  ①Generatorの定義  ②データセットの準備  ③関数fit()で学習  ④学習したGeneratorをファイルに保存 注意して欲しいことは、以下の２つ  ①Generatorは**Keras**で定義すること  ②データセットは**-1.0~1.0に正規化**すること |
| *# Generatorの生成* generator = \_create\_keras\_generator() *# cifar10のデータのロード* (x\_train, y\_train), (\_, \_) = cifar10.load\_data() *# 犬画像の抽出* dataset = np.array([x **for** (x, y) **in** zip(x\_train, y\_train) **if** y == 5]) *# -1.0 ～ 1.0 に正規化* dataset = (dataset - 127.5) / 127.5  *# 学習* train.fit(generator, dataset)  *# Generator の保存* generator.save(**"generator.h5"**)  **def** \_create\_keras\_generator():  w = 2  h = 2  size = 64   model = Sequential()   model.add(Dense(units=w \* h \* size \* 8, input\_dim=100))model.add(Reshape((w, h, size \* 8)))model.add(Activation(**'relu'**))model.add(Conv2DTranspose(  size \* 8,  (5, 5),  strides=(2, 2),  padding=**'same'**,  kernel\_initializer=initializers.random\_normal(stddev=0.02)))model.add(Activation(**'relu'**))model.add(Conv2DTranspose(  size \* 4,  (5, 5),  strides=(2, 2),  padding=**'same'**,  kernel\_initializer=initializers.random\_normal(stddev=0.02)))model.add(Activation(**'relu'**))model.add(Conv2DTranspose(  size \* 2,  (5, 5),  strides=(2, 2),  padding=**'same'**,  kernel\_initializer=initializers.random\_normal(stddev=0.02)))model.add(Activation(**'relu'**))model.add(Conv2DTranspose(  3,  (5, 5),  strides=(2, 2),  padding=**'same'**,  kernel\_initializer=initializers.random\_normal(stddev=0.02)))model.add(Activation(**'tanh'**))**return** model |

|  |
| --- |
| example2.py（学習パラメータの設定例） 　フローはexample1と同じであるが、エポック数などの学習パラメータを指定している。 |
| train.fit(  generator,  dataset,  epochs=25, *# エポック数* data\_size\_per\_batch=64, *# ミニバッチのデータ数* d\_learning\_rate=2.0e-4, *# Discriminator の学習率係数* d\_beta1=0.5, *# Discriminator の beta* g\_learning\_rate=2.0e-4, *# Generator の学習率係数* g\_beta1=0.5 *# Generator の beta* ) |

|  |  |
| --- | --- |
| example3.py（オブザーバーの使用例①－画像出力） 　example2にObserverを追加したプログラム。ここでは、バッチ毎に処理結果を出力している。    　BaseObserverの派生クラス\_CustomObserverを作成し、on\_completed\_batch\_train()をオーバーライドして、**コンソールへの結果出力**と10バッチ毎に**サンプル画像**（その時点のGeneratorが生成できる画像）**のファイルへの出力**を行っている。  **class** \_CustumObserver(BaseObserver):  **def** on\_completed\_batch\_train(self, proxy, epoch\_id, batch\_id, counter, g\_loss, d\_loss,  elapsed\_time):  コンソールへの出力は単純であるので割愛する。**サンプル画像の生成**はon\_completed\_batch\_train()の第２引数**proxy**で行っている（Generatorで直接生成してもよい）。  images, d\_loss, g\_loss = proxy.create\_sample\_imgages(self.sample\_z, self.sample\_images)  proxyの関数create\_sample\_imgages()に乱数で生成したGeneratorへの入力（self.sample\_z）とサンプル画像（self.sample\_images）を渡してGeneratorの出力画像（値は-1.0~1.0）、DiscriminatorのlossとGeneratorのlossを得ている。  そして、このimagesを非正規化（値を0~255に）し、  images = (images + 1.0) \* 127.5  1枚の画像にマージ（詳細はexample3.pyのコードを参照）して、scipy.misc.imsave()を用いてファイルに保存している。 | |
| scipy.misc.imsave(image\_path, images.astype(np.uint8)) | C:\Users\mtanaka\Desktop\新しいフォルダー\train_62_0002_004760_d_loss{1.0459}_g_loss{0.6162}.png |
|  | |
|  | |
| example4.py（オブザーバーの使用例②－ロールバック） 　example2にObserverを追加したプログラム。ここでは、Generatorの破損を検出しロールバックを行っている。  　ロールバックはtrain.fit()内で自動的に生成されるチェックポイントの状態に戻ることである。  　チェックポイントが生成される周期は、train.fit()の引数rollback\_check\_point\_cycleで指定された値でのバッチごとに生成される。  train.fit(  generator,  dataset, **rollback\_check\_point\_cycle=200**, *# 200バッチ毎にロールバック用のチェックポイントを作成*  )  チェックポイントは通常２つ存在し、ロールバックが指示されると古い方にロールバックされる。すなわちrollback\_check\_point\_cycle=200の場合、最後にチェックポイントを生成してから100バッチ後のタイミングでロールバックを実行すると200+100＝300バッチ前の状態に戻ることになる。  ロールバックされる チェックポイント  現在  最後に生成された チェックポイント  このプログラムで実際にロールバックを行っているのは、\_CustomObserverクラスのメンバ関数on\_completed\_batch\_train()である。  ここでは、Generator Lossの10バッチ移動平均を算出し、  self.g\_loss\_moving\_avg\_counter += 1 n = min(self.g\_loss\_moving\_avg\_counter, 10) self.g\_loss\_moving\_avg = ((n - 1) \* self.g\_loss\_moving\_avg + g\_loss) / n  それが4.0を超えた場合にproxyのメンバ関数rollback()を介してロールバックを行っている。（1、２エポック目は対象外。またproxyはon\_completed\_batch\_train()の第２引数）  **if** (epoch\_id > 2) **and** (n == 10) **and** (self.g\_loss\_moving\_avg > 4.0): check\_point = **proxy.rollback()** *#ロールバック*  　proxy.rollback()の戻り値にはNoneとロールバックしたチェックポイントのファイル名の2種類があり、サンプルプログラムでは以下の３通りの処理を行っている。   * Noneの場合 チェックポイントが存在しない →学習終了 * チェックポイントが前回ロールバック時のチェックポイントと同じ →学習終了 * その他 →ロールバックに成功したのでGenerator Lossの移動平均をリセットして、 　学習を継続 | |

|  |  |
| --- | --- |
| example5.py（FeatureModelの使用例－Generatorの評価） 　ここでは、FeatureModelクラスと6枚の画像を用いてGeneratorの評価を行っている。  　はじめに、オブザーバーのコンストラクタに6枚の画像（正規化はしていない）を渡し、オブザーバー内でこれを正規化して保持する。  train.fit(  generator,  dataset, observer=\_CustumObserver(  generator,  ・・・  org\_dataset[-6:], *# Generatorの評価に使用する画像6枚*  ・・・  ) )  また、オブザーバーのコンストラクタでFeatureModelのインスタンスを生成する。  self.featureModel = FeatureModel(generator, seed=123)  FeatureModelの引数seedは初期値の乱数シードである。FeatureModelはGeneratorが実画像を出力するための入力値を推定するが、画像ごとに毎回内部パラメータを初期化する。その初期値はFeatureModelのコンストラクタで乱数を使用して生成されるが、その時のシードとしてseedが使用される。  そして、生成したFeatureModel を用いて**エポック終了時にGeneratorの評価が行われる**。  **def** on\_completed\_epoch\_train(self, proxy, epoch\_id, batch\_id, counter, elapsed\_time): self.\_save\_generator(counter)  **self.\_save\_evaluation\_image(counter) # この関数内で評価**  正規化した実画像を用いてFeatureModelでレプリカを生成する。  **def** \_save\_evaluation\_image(self, counter):  v1 = np.array([self.**featureModel.replicate**(y) **for** y **in** self.normalized\_evaluation\_images])  v1 = \_denormalize\_img\_dataset(v1) | |
| 実画像を奇数列、レプリカを偶数列に並べて一枚の画像にしてファイルに出力している。 |  |
| さらにここでは、実画像とレプリカの相関係数も学習の進捗評価に使用している。  coef = np.corrcoef(self.evaluation\_images.flatten(), v1.flatten())[0, 1] | |

|  |
| --- |
| example6.py（Discriminatorのカスタマイズ例①－tensorflowでの定義） ここではtensorflowで独自のDiscriminatorを定義してDCGANで学習を行っている。関数を作成し、その関数でDiscriminatorを定義し、train.fit()にその関数を渡して実行する。 |
| train.fit(generator, dataset, **custom\_discriminator=\_create\_discriminator**)  **def \_create\_discriminator**(input, is\_training, reuse=**False**):  name = **"custum\_discriminator"   with** tf.variable\_scope(name, reuse=**reuse**) **as** scope: *# 学習パラメータの共有設定*  output\_dim = 64   h0 = lrelu(conv2d(input, output\_dim, **'d\_h0\_conv'**))  h1 = lrelu(batch\_normalization(conv2d(h0, output\_dim\*2, **'d\_h1\_conv'**), is\_training, **'d\_bch\_h1'**))  h2 = lrelu(batch\_normalization(conv2d(h1, output\_dim\*4, **'d\_h2\_conv'**), is\_training, **'d\_bch\_h2'**))  h3 = lrelu(batch\_normalization(conv2d(h2, output\_dim\*8, **'d\_h3\_conv'**), is\_training, **'d\_bch\_h3'**))  h4 = linear(tf.reshape(h3, [-1, 2 \* 2 \* (output\_dim \* 8)]), 1, **'d\_h4\_lin'**)   t\_vars = tf.trainable\_variables()   **return** tf.nn.sigmoid(h4), h4, [var **for** var **in** t\_vars **if** var.name.startswith(name)] | |
| 関数の引数でinputとis\_trainingはtensorflowのプレースフォルダでreuseは通常のbool型の値である。  **この関数は実行時に2回コールされ**、それぞれのinputは実画像またはGeneratorの出力になる。**更新する重みは同じものでなければならない**ことから、reuseは初回コールのときFalseで2回目はTrueとなる。  関数の戻り値はtupleで「Discriminatorのモデル」、「Discriminatorのlogit」、「Discriminatorで学習する重みパラメータ」である。「Discriminatorのlogit」であるが、Lossの計算にtensorflowのtf.nn.sigmoid\_cross\_entropy\_with\_logits()を使用していることからh4が戻り値になっていると思われる  （<http://bamos.github.io/2016/08/09/deep-completion/#ml-heavy-dcgans-in-tensorflow>を参照） |

|  |
| --- |
| example7.py（Discriminatorのカスタマイズ例②－Kerasでの定義） ここではKerasで独自のDiscriminatorを定義してDCGANで学習を行っている。  クラスを作成し、そのクラスのコンストラクタでDiscriminatorを定義し、train.fit()に クラスのインスタンスを渡して実行する。  train.fit(generator, dataset, **custom\_discriminator=custom\_discriminator()**)  **class** custom\_discriminator:  **def** \_\_init\_\_(self):  inputs = Input((32, 32, 3), name=**"input"**)  x = Conv2D(64, (5, 5), strides=(2, 2), activation=**'relu'**, padding=**"same"**, name=**"conv2d\_0"**)(inputs)  x = Conv2D(128, (5, 5), strides=(2, 2), activation=**'relu'**, padding=**"same"**, name=**"conv2d\_1"**)(x)  x = BatchNormalization(name=**"bn\_1"**)(x)  x = Conv2D(256, (5, 5), strides=(2, 2), activation=**'relu'**, padding=**"same"**, name=**"conv2d\_2"**)(x)  x = BatchNormalization(name=**"bn\_2"**)(x)  x = Conv2D(512, (5, 5), strides=(2, 2), activation=**'relu'**, padding=**"same"**, name=**"conv2d\_3"**)(x)  x = BatchNormalization(name=**"bn\_3"**)(x)  x = Flatten()(x)  x = Dense(1, name=**"dense\_4"**)(x)  self.model = Model(inputs=inputs, outputs=x, name=**"custom\_discriminator"**)   **def** \_\_call\_\_(self, \*args, \*\*kwargs):  input = args[0] d\_logits = self.model(input) **return** Activation(**"sigmoid"**)(d\_logits), d\_logits, self.model.trainable\_weights  　tensorflowの場合と異なりクラスを使用している理由は、実画像とGeneratorそれぞれを用いた学習でDiscriminatorの重みを共有するために共有レイヤー（<https://keras.io/ja/getting-started/functional-api-guide/#_2>）を利用しているからである。 |

|  |
| --- |
| example8.py（データセットクラスの使用例） 　ここではBaseDataSetのサブクラスDirectoryImageDataSetを使用してDCGANで学習を行っている。  　DirectoryImageDataSetは**バッチごとに指定されたディレクトリからファイルをロード**している。これは学習データが巨大でメモリ上に展開できない場合に有用である。  コンストラクタにディレクトリへのパス、1バッチのデータ数、画像のshape（幅、高さ、深さ）を指定して、インスタンスを生成する。  dataset = DirectoryImageDataSet(temp\_dir, data\_size\_per\_batch, images.shape[1:])  それを**train.fit\_dataset**()に渡して実行する。  train.fit\_dataset(generator, dataset, …)  　同様にバッチの処理ごとにデータを生成する必要がある場合は、BaseDataSetを継承したクラスを新規に作成すればよい。 |