Generative Adversarial Minority Oversampling: Supplementary Material

Sankha Subhra Mullick
Indian Statistical Institute
Kolkata, India
sankha_r@isical.ac.in

Shounak Datta
Duke University
Durham, NC, USA
shounak.jaduniv@gmail.com

Swagatam Das
Indian Statistical Institute
Kolkata, India
swagatam.das@isical.ac.in

1 Proof of theorems

Theorem 1. *Optimizing the objective function $\mathcal{J}$ is equivalent to the problem of minimizing the following summation of Jensen-Shannon divergences:*

$$\sum_{i=1}^{c} JS\Big(\big(P_i p_i^d + (P_c - P_i)p_i^g\big)\Big|\Big|\sum_{j\neq i}\big(P_j p_j^d + (P_c - P_j)p_j^g\big)\Big)$$

Proof. For simplicity and without loss of generality we focus on a single minority class, say the i^{th} one. Then, we can start by finding that $M_i^*(\mathbf{x})$, which will maximize J_i (the component of $\mathcal{J}$ corresponding to the i^{th} class) for a given G. Therefore, we first find the partial differentiation of J_i, with respect to $M_i(\mathbf{x})$, as follows:

$$\frac{\partial J_i}{\partial M_i(\mathbf{x})} = \frac{P_i p_i^d}{M_i(\mathbf{x})} - \frac{\sum_{j\in\mathcal{C}\setminus\{i\}} P_j p_j^d}{1 - M_i(\mathbf{x})} + \frac{(P_c - P_i)p_i^g}{M_i(\mathbf{x})} - \frac{\sum_{j\in\mathcal{C}\setminus\{i\}}(P_c - P_j)p_j^g}{1 - M_i(\mathbf{x})} \tag{1}$$

Equating (1) to 0, and solving it for $M_i(\mathbf{x})$ gives,

$$M_i^*(\mathbf{x}) = \frac{P_i p_i^d + (P_c - P_i)p_i^g}{\sum_{k=1}^{c}\big(P_k p_k^d + (P_c - P_k)p_k^g\big)} \tag{2}$$

Plugging in the value of $M_i^*(\mathbf{x})$ from (2) back in J_i, we get,

$$
\begin{aligned}
J_i = &\int P_i p_i^d \log \frac{P_i p_i^d + (P_c - P_i)p_i^g}{\sum_{k=1}^c \left(P_k k_k^d + (P_c - P_k)p_k^g\right)} dx + \\
&\int \sum_{j \in \mathcal{C}\setminus\{i\}} P_j p_j^d \log \frac{\sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right)}{\sum_{k=1}^c \left(P_k p_k^d + (P_c - P_k)p_k^g\right)} dx + \\
&\int \left((P_c - P_i)p_i^g\right) \log \frac{P_i p_i^d + (P_c - P_i)p_i^g}{\sum_{k=1}^c \left(P_k k_k^d + (P_c - P_k)p_k^g\right)} dx + \\
&\int \sum_{j \in \mathcal{C}\setminus\{i\}} \left((P_c - P_j)p_j^g + P_j p_j^d\right) \log \frac{\sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right)}{\sum_{k=1}^c \left(P_k p_k^d + (P_c - P_k)p_k^g\right)} dx \\
J_i = &\int \left(P_i p_i^d + (P_c - P_i)p_i^g\right) \log \frac{P_i p_i^d + (P_c - P_i)p_i^g}{\frac{1}{2}\sum_{k=1}^c \left(P_k p_k^d + (P_c - P_k p_k^g\right)} dx - \log 2 \int \left(P_i p_i^d + (P_c - P_i)p_i^g\right) dx + \\
&\int \sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right) \log \frac{\sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right)}{\frac{1}{2}\sum_{k=1}^c \left(P_k p_k^d + (P_c - P_k)p_k^g\right)} dx - \\
&\log 2 \int \sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right) dx \\
J_i = &2JS\Big(\left(P_i p_i^d + (P_c - P_i)p_i^g\right)\Big|\Big| \sum_{j \in \mathcal{C}\setminus\{i\}} \left(P_j p_j^d + (P_c - P_j)p_j^g\right)\Big) - cP_c \log 2
\end{aligned} \tag{3}
$$

From (3), ignoring the constant scalar multiplicative factor and the additive factor $-cP_c \log 2$ (also a constant for a given problem) we can conclude that

$$
\min_G \max_M J \sim \min_{p^g} \sum_{i=1}^c JS\Big(\left(P_i p_i^d + (P_c - P_i)p_i^g\right)\Big|\Big| \sum_{j \neq i} \left(P_j p_j^d + (P_c - P_j)p_j^g\right)\Big), \tag{4}
$$

which completes the proof. $\square$

2 Network architecture and hyperparameter selection

2.1 SMOTE

The number of intra-class neighbours are varied between $\{3, 5, 7\}$, and finally set to 5 which is found to be the best performer.

2.2 Common settings

For all of the networks the batch size is set to 32. For all the generators involved in different algorithms the latent dimension is taken as 100 [6]. For GAMO using convolutional feature extraction the dimension of the feature space is taken as 512. The momentum parameter in batch normalization is set to 0.9, while the α in LeakyReLU is taken as 0.1 (Keras default settings). For convolutional layers the stride is 1, while that for deconvolution layers is set to 2 (for increasing the resolution of the image). We have used Adam [5] optimizer in all cases, for which the β_2 parameter is set to 0.5. The latent dimension for GAMO2pix is also set to 100. The maximum number of steps for different algorithms and datasets are listed in the following Table 1.

2.3 Augmentation

Data augmentation is performed using the “preprocessing” function of the “ImageDataGenerator” class available in “Keras” deep learning API. Table 2 lists the different parameters used for augmentation along with their associated values.

Table 1: List of maximum number of steps used by an algorithm on a dataset

Algorithm	Dataset	Maximum number of steps
Baseline CN	MNIST, Fashion-MNIST, CIFAR10, SVHN	50000
Baseline CN	CelebA, LSUN, SUN397	150000
SMOTE+CN	MNIST	Same as Baseline CN
Augment+CN	Fashion-MNIST, CIFAR10, SVHN, CelebA, LSUN, SUN397	Same as Baseline CN
cGAN+CN	MNIST	Same as Baseline CN
cG+CN	MNIST, Fashion-MNIST	Same as Baseline CN
cG+D+CN	MNIST, Fashion-MNIST	Same as Baseline CN
cDCGAN+CN	Fashion-MNIST, CIFAR10, SVHN, CelebA, LSUN, SUN397	Same as Baseline CN
DOS	Fashion-MNIST, CIFAR10, SVHN, CelebA, LSUN, SUN397	Same as Baseline CN
GAMO\D	MNIST, Fashion-MNIST, CIFAR10, SVHN, CelebA, LSUN, SUN397	Same as Baseline CN
GAMO	MNIST, Fashion-MNIST, CIFAR10, SVHN, CelebA, LSUN, SUN397	Same as Baseline CN
GAMO2pix	Fashion-MNIST, CIFAR10	25000
GAMO2pix	CelebA	50000

Table 2: List of parameters along with their corresponding values chosen for augmenting the datasets.

Parameters	Fashion-MNIST CIFAR10 SVHN	CelebA LSUN SUN50
rotation_range	20	20
width_shift_range	0.2	0.2
height_shift_range	0.2	0.2
shear_range	0.2	0.2
zoom_range	0.2	0.2
brightness_range	$(0.1, 1)$	$(0.1, 1)$
fill_mode	nearest	nearest
horizontal_flip	False	True

The name and value of the parameters follow the convention of the standard "Keras" implementation.

2.4 GAMO network

The GAMO network architecture and hyperparameters, along with the grid search space and the final network is listed in Table 3.

2.5 cGAN/cDCGAN network

The cGAN and cDCGAN network architecture and hyperparameters grid search space and the final network is listed in Table 4.

2.6 Classifier network

The classifier network architecture and hyperparameters grid search space and the final network is listed in Table 5.

Table 3: Grid search space along with the selected network architecture and hyperparameter settings of GAMO framework.

Dataset	Parameters	Grid search space	Final network
MNIST	No. of layers in cTG	$\{2,3,4\}$	Dense, 256, ReLU Dense, 64, ReLU
	BN in cTG	{True, False}	True
	No. of layers in IGU_i	-	Dense, n_i, softmax
	No. of layers in D	$\{2,3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	No. of layers in CN	$\{2,3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 10, softmax
Fashion-MNIST	No. of layers in C	$\{2,3,4\}$	5×5 Conv., 32, LeakyReLU 5×5 Conv., 32, LeakyReLU Dense, 512, tanh
	Average Pooling in C	{True, False}	True
	BN in cTG	{True, False}	True
	Other parameters	-	Identical to MNIST
CIFAR10	No. of layers in C	-	Similar to Fashion-MNIST
	Average Pooling in C	-	True
	No. of layers in cTG	$\{2,3\}$	Dense, 256, ReLU Dense, 64, ReLU
	BN in cTG	-	True
	No. of layers in IGU_i	-	Dense, n_i, softmax
	No. of layers in D	$\{3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	No. of layers in CN	$\{3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 10, softmax
CelebA	No. of layers in C	$\{4,5,6\}$	5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU Dense, 512, tanh
	Average Pooling in C	-	True
	No. of layers in cTG	$\{3,4\}$	Dense, 256, ReLU Dense, 64, ReLU
	BN in cTG	-	True
	No. of layers in IGU_i	-	Dense, n_i, softmax
	No. of layers in D	$\{3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	No. of layers in CN	$\{3,4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 5, softmax
Optimizer	Adam (β_1)	$\{0.0002, 0.002, 0.02\}$	0.0002

The β_2 parameter of Adam optimizer is set to 0.5 for all experiments.
If True then BN or Batch Normalization [4] is applied after every dense layer of cTG.
If True then 2×2 AveragePooling is applied after every convolution layer.
Due to the similar nature of the two datasets, the optimum architecture and parameter settings for CIFAR10 are also used for SVHN.
Due to the similar nature of the three datasets, the optimum architecture and parameter settings for CelebA are also used for LSUN, and SUN397.

2.7 DOS

The DOS network for a dataset is designed similarly to the baseline classifier network. The neighborhood size is set following the guideline of the original article [1].

Table 4: Grid search space along with the selected network architecture and hyperparameter settings of the cGAN/cDCGAN network.

Dataset	Parameters	Grid search space	Final network
MNIST	No. of layers in Generator	$\{3, 4\}$	Dense, 128, ReLU Dense, 256, ReLU Dense, 784, tanh
	No. of layers in Discriminator	$\{3, 4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
Fashion-MNIST	No. of layers in Generator	$\{4, 5, 6, 7, 8\}$	Dense, 6272, LeakyReLU 4×4 Conv., 128, LeakyReLU 4×4 Deconv., 128, LeakyReLU 4×4 Conv., 128, LeakyReLU 4×4 Deconv., 128, LeakyReLU 5×5 Conv., 128, LeakyReLU 5×5 Conv., 1, tanh
	BN in Generator	{True, False}	True
	Average Pooling in Generator	{True, False}	False
	No. of layers in Discriminator	$\{5, 6\}$	5×5 Conv., 32, LeakyReLU 5×5 Conv., 32, LeakyReLU Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	Average Pooling in Discriminator	{True, False}	True
CIFAR10	No. of layers in Generator	$\{4, 5, 6, 7, 8\}$	Dense, 512, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 5×5, Deconv., 3, tanh
	BN in Generator	-	True
	Average Pooling in Generator	-	False
	No. of layers in Discriminator	$\{5, 6\}$	5×5 Conv., 32, LeakyReLU 5×5 Conv., 32, LeakyReLU Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	Average Pooling in Discriminator	-	True
CelebA	No. of layers in Generator	$\{4, 5, 6, 7, 8\}$	Dense, 2048, LeakyReLU 4×4, Deconv., 64, LeakyReLU 4×4, Deconv., 64, LeakyReLU 4×4, Deconv., 64, LeakyReLU 4×4, Deconv., 3, tanh
	BN in Generator	-	True
	Average Pooling in Generator	-	False
	No. layers in Discriminator	$\{6, 7, 8\}$	5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 1, sigmoid
	Average Pooling in Discriminator	-	True
Optimizer	Adam (β_1)	$\{0.0002, 0.002, 0.02\}$	0.0002

The β_2 parameter of Adam optimizer is set to 0.5 for all experiments.
If True then BN or Batch Normalization [4] is applied after every dense layer of conditional generator.
In case of MNIST and Fashion-MNIST the generator used in cG+CN, and cG+D+CN has similar architecture to that of cGAN.
If True then 2×2 AveragePooling is applied after every convolution layer.
Due to the similar nature of the two datasets, the optimum architecture and parameter settings for CIFAR10 are also used for SVHN.
Due to the similar nature of the three datasets, the optimum architecture and parameter settings for CelebA are also used for LSUN, and SUN397.

Table 5: Grid search space along with the selected network architecture and hyperparameter settings of the classifier network.

Dataset	Parameters	Grid search space	Final network
MNIST	No. of Dense layers	$\{2, 3, 4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 10, softmax
Fashion-MNIST	No. of Conv. layers	$\{2, 3, 4\}$	5×5 Conv., 32, LeakyReLU 5×5 Conv., 32, LeakyReLU
	Average Pooling	{True, False}	True
	Other parameters	-	Identical to MNIST
CIFAR10	No. of Conv. layers	$\{2, 3, 4\}$	5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU
	Average Pooling	{True, False}	True
	No. of Dense layers	$\{2, 3, 4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 10, softmax
CelebA	No. of Conv. layers	$\{3, 4, 5\}$	5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU 5×5, Conv., 32, LeakyReLU
	Average Pooling	-	True
	No. of Dense layers	$\{2, 3, 4\}$	Dense, 256, LeakyReLU Dense, 128, LeakyReLU Dense, 5, softmax
Optimizer	Adam (β_1)	$\{0.0002, 0.002, 0.02\}$	0.0002

The β_2 parameter of Adam optimizer is set to 0.5 for all experiments.
If True then 2×2 AveragePooling is applied after every convolution layer.
Due to the similar nature of the two datasets, the optimum architecture and parameter settings for CIFAR10 are also used for SVHN.
Due to the similar nature of the three datasets, the optimum architecture and parameter settings for CelebA are also used for LSUN, and SUN397.

Note

The Dense parts are kept similar throughout analogous networks as that particular architecture is found to be performing better on average over all algorithms, after the grid search.

3 GAMO2pix

The GAMO2pix network architecture and hyperparameters of the final network is listed in Table 6.

Table 6: Network architecture and hyperparameter settings of the GAMO2pix network.

Dataset	Parameters	Final network
Fashion-MNIST	Mean layer	Dense, 100
	Log Variance layer	Dense, 100
	Decoder	Dense, 1568, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Conv., 1, tanh
CIFAR10	Decoder	Dense, 512, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Conv., 3, tanh
CelebA	Decoder	Dense, 512, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Deconv., 32, LeakyReLU 4×4, Conv., 3, tanh
Optimizer	Adam (β_1)	0.0002

For all the datasets, the corresponding feature extractor network trained by GAMO is connected to the Mean and Log Variance layer. Among all the components of GAMO2pix only the feature extractor is kept fixed throughout the training period.
The β_2 parameter of Adam optimizer is set to 0.5 for all experiments.
Due to the similar nature of the two datasets, the optimum architecture and parameter settings for CIFAR10 are also used for SVHN.
The Mean layer and the Log Variance layer is kept similar to Fashion-MNIST for CIFAR10, SVHN, and CelebA.

4 Results on non-image datasets

The five non-image datasets used for additionally validating the performance of GAMO are collected from University of California, Irvine, Machine Learning Repository [3], KEEL [7], and LibSVM [2]. The different properties of these datasets are detailed in Table 7. The results of the comparison between GAMO, Baseline CN, cGAN+CN, SMOTE+CN, and GAMO\D are summarized in the following Table 8. We have used a 10-fold stratified cross-validation and report the mean ACSA and GM. The best ACSA and GM among the contenders are boldfaced for each dataset. It is evident from Table 8 that GAMO on average can retain its commendable performance on non-image datasets as well. Interestingly, the average performance of SMOTE+CN in terms of ACSA is close to that of GAMO, justifying the popularity of SMOTE over the past couple of decades. However, compared to SMOTE+CN, the proposed GAMO shows a better consistency over all the classes as indicated by the higher average GM.

Table 7: Detailed description of the datasets.

Dataset name	Number of points	Number of dimensions	Number of classes	IR
Abalone19	4177	8	2	129.5
Chess	28056	6	18	168.6
Cover Type	581012	54	7	103.13
IJCNN1	141691	22	2	9.4
Magic	19020	10	2	1.8

Table 8: Results on non-image class imbalanced benchmark datasets.

Datasets	Baseline CN		SMOTE+CN		cGAN+CN		GAMO\D		GAMO	
	ACSA	GM	ACSA	GM	ACSA	GM	ACSA	GM	ACSA	GM
Abalone19	0.50	0.00	0.58	**0.48**	**0.59**	**0.48**	0.46	0.00	**0.59**	**0.48**
Chess	0.29	0.00	0.30	0.00	0.25	0.00	0.16	0.00	**0.32**	0.20
Cover Type	0.51	0.43	**0.70**	**0.64**	0.57	0.47	0.46	0.31	0.66	**0.64**
IJCNN1	0.91	0.90	0.93	0.93	0.93	0.93	0.89	0.88	**0.95**	**0.95**
Magic	0.82	0.82	0.81	0.81	0.82	0.82	0.73	0.72	**0.83**	**0.83**
Average Performance	0.60	0.43	0.66	0.57	0.63	0.54	0.54	0.38	**0.67**	**0.62**

References

[1] Shin Ando and Chun Yuan Huang. Deep over-sampling framework for classifying imbalanced data. In *Machine Learning and Knowledge Discovery in Databases*, pages 770–785. Springer International Publishing, 2017.

[2] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011.

[3] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on International Conference on Machine Learning*, pages 448–456, 2015.

[5] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980, presented at International Conference on Learning Representations (ICLR)*, 2015.

[6] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[7] Isaac Triguero, Sergio Gonzlez, Jose M. Moyano, Salvador Garca, Jess Alcal-Fdez, Julin Luengo, Alberto Fernndez, Maria Jos del Jess, Luciano Snchez, and Francisco Herrera. KEEL 3.0: an open source software for multi-stage analysis in data mining. *International Journal of Computational Intelligence Systems*, 10:1238–1249, 2011.

Codes

The codes are implemented in Python and compatible with any version which is 2.7 and above. The codes require support from additional libraries such as Keras deep learning API with any backend (for example TensorFlow) of choice, numpy, scikit-learn, scipy, os, sys, opencv, pickle, and matplotlib. Here we provide the codes of GAMO for MNIST and Fashion-MNIST, alongwith GAMO2pix for the later dataset. We also provide a pre-processing code which given the original dataset in csv format will output a class-imbalanced version similar to ours. Codes and datasets can also be downloaded from `https://github.com/SankhaSubhra/GAMO`.

Data pre-processing for MNIST

Download the MNIST dataset from source, convert it to csv format, and then execute the following to induce class-imbalance.

```
import numpy as np
import pickle as pk

trainDataOri=np.loadtxt('mnist_train.csv', delimiter= ',')
testDataOri=np.loadtxt('mnist_test.csv', delimiter= ',')
trainSetOri, trainLabOri=trainDataOri[:, 1:], trainDataOri[:, 0]
testSetOri, testLabOri=testDataOri[:, 1:], testDataOri[:, 0]

pointsInTrClass=((4000, 2000, 1000, 750, 500, 350, 200, 100, 60, 40))

numClass=10
pointsInTsClass=100
maxPTrClass, maxPTsClass=4000, 800

classLocTr=np.insert(np.cumsum(pointsInTrClass), 0, 0)
classMapTr, classMapTs, trainPoints, testPoints=list(), list(), list(), list()
for i in range(numClass):
    classMapTr.append(np.where(trainLabOri==i)[0])
    classMapTs.append(np.where(testLabOri==i)[0])
trainS=np.zeros((np.sum(pointsInTrClass), trainSetOri.shape[1]))
trainL=np.zeros((np.sum(pointsInTrClass),1))

for i in range(numClass):
    randIdxTr=np.random.randint(0, maxPTrClass, pointsInTrClass[i])
    trainPoints.append(classMapTr[i][randIdxTr])
    trainS[classLocTr[i]:classLocTr[i+1], :]=trainSetOri[trainPoints[i], :]
    trainL[classLocTr[i]:classLocTr[i+1], 0]=trainLabOri[trainPoints[i]]
trainDataFinal=np.hstack((trainS, trainL))

testS=np.zeros((int(numClass*pointsInTsClass), testSetOri.shape[1]))
testL=np.zeros((int(numClass*pointsInTsClass),1))
classLocTs=np.arange(0, (numClass+1)*pointsInTsClass, pointsInTsClass)
for i in range(numClass):
    randIdxTs=np.random.randint(0, maxPTsClass, pointsInTsClass)
    testPoints.append(classMapTs[i][randIdxTs])
    testS[classLocTs[i]:classLocTs[i+1], :]=testSetOri[testPoints[i], :]
    testL[classLocTs[i]:classLocTs[i+1], 0]=testLabOri[testPoints[i]]
testDataFinal=np.hstack((testS, testL))

sampledPoints={'Mnist_100_trainSamples':trainPoints, 'Mnist_100_testSamples':testPoints}
pk.dump(sampledPoints, open( 'Mnist_100_sampledPoints.pkl', 'wb' ))
np.savetxt('Mnist_100_trainData.csv', trainDataFinal, delimiter=",")
np.savetxt('Mnist_100_testData.csv', testDataFinal, delimiter=",")
```

GAMO main script for MNIST

The following is the “main” script which can be executed to classify MNIST dataset (with class-imbalance) using GAMO. Besides standard python libraries the script imports two additional ones (described in the following sections) namely “dense_net.py”, and “dense_suppli.py”, which respectively contains the functions necessary defining the network and performing supporting tasks.

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

import os
```

```python
import numpy as np
import dense_suppli as spp
import dense_net as nt
import matplotlib.pyplot as plt
from keras.preprocessing import image
from keras.layers import Input
from keras.models import Model
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical

# For selecting a GPU
# os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
# os.environ["CUDA_VISIBLE_DEVICES"]="1"

# Ground works
fileName=['Mnist_100_trainData.csv', 'Mnist_100_testData.csv']
fileStart='Mnist_100_Gamo'
fileEnd, savePath='_Model.h5', fileStart+'/'
adamOpt=Adam(0.0002, 0.5)
latDim, modelSamplePd, resSamplePd=100, 5000, 500
plt.ion()

batchSize, max_step=32, 50000

trainS, labelTr=spp.fileRead(fileName[0])
testS, labelTs=spp.fileRead(fileName[1])

n, m=trainS.shape[0], testS.shape[0]
trainS, testS=(trainS-127.5)/127.5, (testS-127.5)/127.5

labelTr, labelTs, c, pInClass, _=spp.relabel(labelTr, labelTs)
imbalancedCls, toBalance, imbClsNum, ir=spp.irFind(pInClass, c)

labelsCat=to_categorical(labelTr)

shuffleIndex=np.random.choice(np.arange(n), size=(n,), replace=False)
trainS=trainS[shuffleIndex]
labelTr=labelTr[shuffleIndex]
labelsCat=labelsCat[shuffleIndex]
classMap=list()
for i in range(c):
    classMap.append(np.where(labelTr==i)[0])

# model initialization
mlp=nt.denseMlpCreate()
mlp.compile(loss='mean_squared_error', optimizer=adamOpt)
mlp.trainable=False

dis=nt.denseDisCreate()
dis.compile(loss='mean_squared_error', optimizer=adamOpt)
dis.trainable=False

gen=nt.denseGamoGenCreate(latDim)

gen_processed, genP_mlp, genP_dis=list(), list(), list()
for i in range(imbClsNum):
    dataMinor=trainS[classMap[i], :]
    numMinor=dataMinor.shape[0]
    gen_processed.append(nt.denseGenProcessCreate(numMinor, dataMinor))

    ip1=Input(shape=(latDim,))
    ip2=Input(shape=(c,))
    op1=gen([ip1, ip2])
    op2=gen_processed[i](op1)
    op3=mlp(op2)
    genP_mlp.append(Model(inputs=[ip1, ip2], outputs=op3))
    genP_mlp[i].compile(loss='mean_squared_error', optimizer=adamOpt)

    ip1=Input(shape=(latDim,))
    ip2=Input(shape=(c,))
    ip3=Input(shape=(c,))
    op1=gen([ip1, ip2])
    op2=gen_processed[i](op1)
    op3=dis([op2, ip3])
    genP_dis.append(Model(inputs=[ip1, ip2, ip3], outputs=op3))
    genP_dis[i].compile(loss='mean_squared_error', optimizer=adamOpt)

# for record saving
batchDiv, numBatches, bSStore=spp.batchDivision(n, batchSize)
genClassPoints=int(np.ceil(batchSize/c))
fig, axs=plt.subplots(imbClsNum, 3)
```

```
if not os.path.exists(fileStart):
    os.makedirs(fileStart)
picPath=savePath+'Pictures'
if not os.path.exists(picPath):
    os.makedirs(picPath)

iter=np.int(np.ceil(max_step/resSamplePd)+1)
acsaSaveTr, gmSaveTr, accSaveTr=np.zeros((iter,)), np.zeros((iter,)), np.zeros((iter,))
acsaSaveTs, gmSaveTs, accSaveTs=np.zeros((iter,)), np.zeros((iter,)), np.zeros((iter,))
confMatSaveTr, confMatSaveTs=np.zeros((iter, c, c)), np.zeros((iter, c, c))
tprSaveTr, tprSaveTs=np.zeros((iter, c)), np.zeros((iter, c))

# training
step=0
while step<max_step:
    for j in range(numBatches):
        x1, x2=batchDiv[j, 0], batchDiv[j+1, 0]
        validR=np.ones((bSStore[j, 0],1))-np.random.uniform(0,0.1, size=(bSStore[j, 0], 1))
        mlp.train_on_batch(trainS[x1:x2], labelsCat[x1:x2])
        dis.train_on_batch([trainS[x1:x2], labelsCat[x1:x2]], validR)

        invalid=np.zeros((bSStore[j, 0], 1))+np.random.uniform(0, 0.1, size=(bSStore[j, 0], 1))
        randNoise=np.random.normal(0, 1, (bSStore[j, 0], latDim))
        fakeLabel=spp.randomLabelGen(toBalance, bSStore[j, 0], c)
        rLPerClass=spp.rearrange(fakeLabel, imbClsNum)
        fakePoints=np.zeros((bSStore[j, 0], 784))
        genFinal=gen.predict([randNoise, fakeLabel])
        for i1 in range(imbClsNum):
            if rLPerClass[i1].shape[0]!=0:
                temp=genFinal[rLPerClass[i1]]
                fakePoints[rLPerClass[i1]]=gen_processed[i1].predict(temp)

        mlp.train_on_batch(fakePoints, fakeLabel)
        dis.train_on_batch([fakePoints, fakeLabel], invalid)

        for i1 in range(imbClsNum):
            validA=np.ones((genClassPoints, 1))
            randomLabel=np.zeros((genClassPoints, c))
            randomLabel[:, i1]=1
            randNoise=np.random.normal(0, 1, (genClassPoints, latDim))
            oppositeLabel=np.ones((genClassPoints, c))-randomLabel
            genP_mlp[i1].train_on_batch([randNoise, randomLabel], oppositeLabel)
            genP_dis[i1].train_on_batch([randNoise, randomLabel, randomLabel], validA)

        if step%resSamplePd==0:
            saveStep=int(step//resSamplePd)

            pLabel=np.argmax(mlp.predict(trainS), axis=1)
            acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTr)
            print('Train: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
            print('TPR: ', np.round(tpr, 2))
            acsaSaveTr[saveStep], gmSaveTr[saveStep], accSaveTr[saveStep]=acsa, gm, acc
            confMatSaveTr[saveStep]=confMat
            tprSaveTr[saveStep]=tpr

            pLabel=np.argmax(mlp.predict(testS), axis=1)
            acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTs)
            print('Test: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
            print('TPR: ', np.round(tpr, 2))
            acsaSaveTs[saveStep], gmSaveTs[saveStep], accSaveTs[saveStep]=acsa, gm, acc
            confMatSaveTs[saveStep]=confMat
            tprSaveTs[saveStep]=tpr

            for i1 in range(imbClsNum):
                testNoise=np.random.normal(0, 1, (3, latDim))
                testLabel=np.zeros((3, c))
                testLabel[:, i1]=1
                genFinal=gen.predict([testNoise, testLabel])
                genImages=gen_processed[i1].predict(genFinal)
                genImages=np.reshape(genImages, (3, 28, 28))
                for i2 in range(3):
                    img=image.array_to_img(np.expand_dims(genImages[i2], axis=-1), scale=True)
                    axs[i1,i2].imshow(img, cmap='gray')
                    axs[i1,i2].axis('off')
            plt.show()
            plt.pause(5)

            figFileName=picPath+'/'+fileStart+'_'+str(step)+'.png'
            plt.savefig(figFileName, bbox_inches='tight')
```

```
        if step%modelSamplePd==0 and step!=0:
            direcPath=savePath+'gamo_models_'+str(step)
            if not os.path.exists(direcPath):
                os.makedirs(direcPath)
            gen.save(direcPath+'/GEN_'+str(step)+fileEnd)
            mlp.save(direcPath+'/MLP_'+str(step)+fileEnd)
            dis.save(direcPath+'/DIS_'+str(step)+fileEnd)
            for i in range(imbClsNum):
                gen_processed[i].save(direcPath+'/GenP_'+str(i)+'_'+str(step)+fileEnd)

        step=step+2
        if step>=max_step: break

figFileName=picPath+'/'+fileStart+'_'+str(step)+'.png'
plt.savefig(figFileName, bbox_inches='tight')

pLabel=np.argmax(mlp.predict(trainS), axis=1)
acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTr)
print('Performance on Train Set: Step:', step, 'ACSA:', np.round(acsa, 4), 'GM:', np.round(gm, 4))
print('TPR:', np.round(tpr, 2))
acsaSaveTr[-1], gmSaveTr[-1], accSaveTr[-1]=acsa, gm, acc
confMatSaveTr[-1]=confMat
tprSaveTr[-1]=tpr

pLabel=np.argmax(mlp.predict(testS), axis=1)
acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTs)
print('Performance on Test Set: Step:', step, 'ACSA:', np.round(acsa, 4), 'GM:', np.round(gm, 4))
print('TPR:', np.round(tpr, 2))
acsaSaveTs[-1], gmSaveTs[-1], accSaveTs[-1]=acsa, gm, acc
confMatSaveTs[-1]=confMat
tprSaveTs[-1]=tpr

direcPath=savePath+'gamo_models_'+str(step)
if not os.path.exists(direcPath):
    os.makedirs(direcPath)
gen.save(direcPath+'/GEN_'+str(step)+fileEnd)
mlp.save(direcPath+'/MLP_'+str(step)+fileEnd)
dis.save(direcPath+'/DIS_'+str(step)+fileEnd)
for i in range(imbClsNum):
    gen_processed[i].save(direcPath+'/GenP_'+str(i)+'_'+str(step)+fileEnd)

resSave=savePath+'Results'
np.savez(resSave, acsa=acsa, gm=gm, tpr=tpr, confMat=confMat, acc=acc)
recordSave=savePath+'Record'
np.savez(recordSave, acsaSaveTr=acsaSaveTr, gmSaveTr=gmSaveTr, accSaveTr=accSaveTr,
    acsaSaveTs=acsaSaveTs, gmSaveTs=gmSaveTs, accSaveTs=accSaveTs, confMatSaveTr=confMatSaveTr,
    confMatSaveTs=confMatSaveTs, tprSaveTr=tprSaveTr, tprSaveTs=tprSaveTs)
```

GAMO network for MNIST

Save the following functions in a single file named “dense_net.py” which will be imported in the “main”.

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

from keras import backend as K
from keras.layers import Input, Dense, RepeatVector, Lambda
from keras.layers import BatchNormalization, Concatenate, Multiply
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Model

def denseGamoGenCreate(latDim):
    noise=Input(shape=(latDim,))
    labels=Input(shape=(10,))
    gamoGenInput=Concatenate()([noise, labels])

    x=Dense(256, activation='relu')(gamoGenInput)
    x=BatchNormalization(momentum=0.9)(x)

    x=Dense(64, activation='relu')(x)
    gamoGenFinal=BatchNormalization(momentum=0.9)(x)

    gamoGen=Model([noise, labels], gamoGenFinal)
    gamoGen.summary()
    return gamoGen

def denseGenProcessCreate(numMinor, dataMinor):
    ip1=Input(shape=(64,))
```

```
    x=Dense(numMinor, activation='softmax')(ip1)
    x=RepeatVector(784)(x)
    genProcessFinal=Lambda(lambda x: K.sum(x*K.transpose(K.constant(dataMinor)), axis=2))(x)

    genProcess=Model(ip1, genProcessFinal)
    return genProcess

def denseDisCreate():
    imIn=Input(shape=(784,))
    labels=Input(shape=(10,))
    disInput=Concatenate()([imIn, labels])

    x=Dense(256)(disInput)
    x=LeakyReLU(alpha=0.1)(x)

    x=Dense(128)(x)
    x=LeakyReLU(alpha=0.1)(x)

    disFinal=Dense(1, activation='sigmoid')(x)

    dis=Model([imIn, labels], disFinal)
    dis.summary()
    return dis

def denseMlpCreate():

    imIn=Input(shape=(784,))

    x=Dense(256)(imIn)
    x=LeakyReLU(alpha=0.1)(x)

    x=Dense(128)(x)
    x=LeakyReLU(alpha=0.1)(x)

    mlpFinal=Dense(10, activation='softmax')(x)

    mlp=Model(imIn, mlpFinal)
    mlp.summary()
    return mlp
```

GAMO supplementary functions for MNIST: dense_suppli.py

Save the following functions in a single file named "dense_suppli.py", which will be imported in the "main".

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

import sys
import numpy as np
from sklearn.metrics import confusion_matrix
from scipy.spatial.distance import cdist
from keras.utils.np_utils import to_categorical

def relabel(labelTr, labelTs):
    unqLab, pInClass=np.unique(labelTr, return_counts=True)
    sortedUnqLab=np.argsort(pInClass, kind='mergesort')
    c=sortedUnqLab.shape[0]
    labelsNewTr=np.zeros((labelTr.shape[0],))-1
    labelsNewTs=np.zeros((labelTs.shape[0],))-1
    pInClass=np.sort(pInClass)
    classMap=list()
    for i in range(c):
        labelsNewTr[labelTr==unqLab[sortedUnqLab[i]]]=i
        labelsNewTs[labelTs==unqLab[sortedUnqLab[i]]]=i
        classMap.append(np.where(labelsNewTr==i)[0])
    return labelsNewTr, labelsNewTs, c, pInClass, classMap

def irFind(pInClass, c, irIgnore=1):
    ir=pInClass[-1]/pInClass
    imbalancedCls=np.arange(c)[ir>irIgnore]
    toBalance=np.subtract(pInClass[-1], pInClass[imbalancedCls])
    imbClsNum=toBalance.shape[0]
    if imbClsNum==0: sys.exit('No imbalanced classes found, exiting ...')
    return imbalancedCls, toBalance, imbClsNum, ir

def fileRead(fileName):
    dataTotal=np.loadtxt(fileName, delimiter=',')
```

```
    data=dataTotal[:, :-1]
    labels=dataTotal[:, -1]
    return data, labels

def indices(pLabel, tLabel):
    confMat=confusion_matrix(tLabel, pLabel)
    nc=np.sum(confMat, axis=1)
    tp=np.diagonal(confMat)
    tpr=tp/nc
    acsa=np.mean(tpr)
    gm=np.prod(tpr)**(1/confMat.shape[0])
    acc=np.sum(tp)/np.sum(nc)
    return acsa, gm, tpr, confMat, acc

def randomLabelGen(toBalance, batchSize, c):
    cumProb=np.cumsum(toBalance/np.sum(toBalance))
    bins=np.insert(cumProb, 0, 0)
    randomValue=np.random.rand(batchSize,)
    randLabel=np.digitize(randomValue, bins)-1
    randLabel_cat=to_categorical(randLabel)
    labelPadding=np.zeros((batchSize, c-randLabel_cat.shape[1]))
    randLabel_cat=np.hstack((randLabel_cat, labelPadding))
    return randLabel_cat

def batchDivision(n, batchSize):
    numBatches, residual=int(np.ceil(n/batchSize)), int(n%batchSize)
    if residual==0:
        residual=batchSize
    batchDiv=np.zeros((numBatches+1,1), dtype='int64')
    batchSizeStore=np.ones((numBatches, 1), dtype='int64')
    batchSizeStore[0:-1, 0]=batchSize
    batchSizeStore[-1, 0]=residual
    for i in range(numBatches):
        batchDiv[i]=i*batchSize
    batchDiv[numBatches]=batchDiv[numBatches-1]+residual
    return batchDiv, numBatches, batchSizeStore

def rearrange(labelsCat, numImbCls):
    labels=np.argmax(labelsCat, axis=1)
    arrangeMap=list()
    for i in range(numImbCls):
        arrangeMap.append(np.where(labels==i)[0])
    return arrangeMap
```

Data pre-processing for Fashion-MNIST

Download the Fashion-MNIST dataset from source, convert it to csv format, and then execute the following to induce class-imbalance.

```
import numpy as np
import pickle as pk

trainDataOri=np.loadtxt('fashionMnist_train.csv', delimiter= ',')
testDataOri=np.loadtxt('fashionMnist_test.csv', delimiter= ',')
trainSetOri, trainLabOri=trainDataOri[:, 1:], trainDataOri[:, 0]
testSetOri, testLabOri=testDataOri[:, 1:], testDataOri[:, 0]

n, m=trainSetOri.shape[0], testSetOri.shape[0]
for i in range(n):
    temp=np.reshape(trainSetOri[i], (28, 28))
    trainSetOri[i]=np.transpose(temp).flatten()
for i in range(m):
    temp=np.reshape(testSetOri[i], (28, 28))
    testSetOri[i]=np.transpose(temp).flatten()

pointsInTrClass=((4000, 2000, 1000, 750, 500, 350, 200, 100, 60, 40))

numClass=10
pointsInTsClass=100
maxPTrClass, maxPTsClass=4000, 800

classLocTr=np.insert(np.cumsum(pointsInTrClass), 0, 0)
classMapTr, classMapTs, trainPoints, testPoints=list(), list(), list(), list()
for i in range(numClass):
    classMapTr.append(np.where(trainLabOri==i)[0])
    classMapTs.append(np.where(testLabOri==i)[0])
trainS=np.zeros((np.sum(pointsInTrClass), trainSetOri.shape[1]))
trainL=np.zeros((np.sum(pointsInTrClass),1))
```

```
for i in range(numClass):
    randIdxTr=np.random.randint(0, maxPTrClass, pointsInTrClass[i])
    trainPoints.append(classMapTr[i][randIdxTr])
    trainS[classLocTr[i]:classLocTr[i+1], :]=trainSetOri[trainPoints[i], :]
    trainL[classLocTr[i]:classLocTr[i+1], 0]=trainLabOri[trainPoints[i]]
trainDataFinal=np.hstack((trainS, trainL))

testS=np.zeros((int(numClass*pointsInTsClass), testSetOri.shape[1]))
testL=np.zeros((int(numClass*pointsInTsClass),1))
classLocTs=np.arange(0, (numClass+1)*pointsInTsClass, pointsInTsClass)
for i in range(numClass):
    randIdxTs=np.random.randint(0, maxPTsClass, pointsInTsClass)
    testPoints.append(classMapTs[i][randIdxTs])
    testS[classLocTs[i]:classLocTs[i+1], :]=testSetOri[testPoints[i], :]
    testL[classLocTs[i]:classLocTs[i+1], 0]=testLabOri[testPoints[i]]
testDataFinal=np.hstack((testS, testL))

sampledPoints={'fMnist_100_trainSamples':trainPoints, 'fMnist_100_testSamples':testPoints}
pk.dump(sampledPoints, open( 'fMnist_100_sampledPoints.pkl', 'wb' ))
np.savetxt('fMnist_100_trainData.csv', trainDataFinal, delimiter=",")
np.savetxt('fMnist_100_testData.csv', testDataFinal, delimiter=",")
```

GAMO main script for Fashion-MNIST

Similar to GAMO for MNIST, the following works as a “main” script which classifies Fashion-MNIST dataset. The code in addition to the standard libraries requires “fashion_mnist_net.py” and “fashion_mnist_suppli.py” (described in the subsequent sections), which respectively provides the necessary functions to design the network and perform supporting tasks.

```
import os
import numpy as np
import fashion_mnist_suppli as spp
import fashion_mnist_net as nt
from keras.layers import Input
from keras.models import Model
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical

fileName=['fMnist_100_trainData.csv', 'fMnist_100_testData.csv']
fileStart='fMnist_100_Gamo'
fileEnd, savePath='_Model.h5', fileStart+'/'
adamOpt=Adam(0.0002, 0.5)
latDim, modelSamplePd, resSamplePd=100, 5000, 500

batchSize, max_step=32, 50000

trainS, labelTr=spp.fileRead(fileName[0])
testS, labelTs=spp.fileRead(fileName[1])

n, m=trainS.shape[0], testS.shape[0]
trainS, testS=(trainS-127.5)/127.5, (testS-127.5)/127.5
trainS, testS=np.reshape(trainS, (n, 28, 28, 1)), np.reshape(testS, (m, 28, 28, 1))

labelTr, labelTs, c, pInClass, _=spp.relabel(labelTr, labelTs)
imbalancedCls, toBalance, imbClsNum, ir=spp.irFind(pInClass, c)

labelsCat=to_categorical(labelTr)

shuffleIndex=np.random.choice(np.arange(n), size=(n,), replace=False)
trainS=trainS[shuffleIndex]
labelTr=labelTr[shuffleIndex]
labelsCat=labelsCat[shuffleIndex]
classMap=list()
for i in range(c):
    classMap.append(np.where(labelTr==i)[0])

mlp=nt.fMnistGamoMlpCreate()
mlp.compile(loss='mean_squared_error', optimizer=adamOpt)
mlp.trainable=False

gamoConv=nt.fMnistGamoConvCreate()
ip1=Input(shape=(28, 28, 1))
conv_mlp=Model(inputs=gamoConv.inputs, outputs=mlp(gamoConv.outputs))
conv_mlp.compile(loss='mean_squared_error', optimizer=adamOpt)

dis=nt.fMnistGamoDisCreate()
```

```
dis.compile(loss='mean_squared_error', optimizer=adamOpt)
dis.trainable=False

gen=nt.fMnistGamoGenCreate(latDim)

gen_processed, genP_mlp, genP_dis=list(), list(), list()
for i in range(imbClsNum):
    dataMinor=trainS[classMap[i], :]
    numMinor=dataMinor.shape[0]
    gen_processed.append(nt.fMnistGenProcessCreate(numMinor))

    ip1=Input(shape=(latDim,))
    ip2=Input(shape=(c,))
    ip3=Input(shape=(512, numMinor))
    op1=gen([ip1, ip2])
    op2=gen_processed[i]([op1, ip3])
    op3=mlp(op2)
    genP_mlp.append(Model(inputs=[ip1, ip2, ip3], outputs=op3))
    genP_mlp[i].compile(loss='mean_squared_error', optimizer=adamOpt)

    ip1=Input(shape=(latDim,))
    ip2=Input(shape=(c,))
    ip3=Input(shape=(512, numMinor))
    ip4=Input(shape=(c,))
    op1=gen([ip1, ip2])
    op2=gen_processed[i]([op1, ip3])
    op3=dis([op2, ip4])
    genP_dis.append(Model(inputs=[ip1, ip2, ip3, ip4], outputs=op3))
    genP_dis[i].compile(loss='mean_squared_error', optimizer=adamOpt)

batchDiv, numBatches, bSStore=spp.batchDivision(n, batchSize)
genClassPoints=int(np.ceil(batchSize/c))

inClassPoints=list()
inClassPoints=[None]*imbClsNum

validA=np.ones((genClassPoints, 1))

if not os.path.exists(fileStart):
    os.makedirs(fileStart)
iter=np.int(np.ceil(max_step/resSamplePd)+1)
acsaSaveTr, gmSaveTr, accSaveTr=np.zeros((iter,)), np.zeros((iter,)), np.zeros((iter,))
acsaSaveTs, gmSaveTs, accSaveTs=np.zeros((iter,)), np.zeros((iter,)), np.zeros((iter,))
confMatSaveTr, confMatSaveTs=np.zeros((iter, c, c)), np.zeros((iter, c, c))
tprSaveTr, tprSaveTs=np.zeros((iter, c)), np.zeros((iter, c))

step=0
while step<max_step:

    conv_mlp.fit(trainS, labelsCat, batch_size=batchSize, verbose=0)
    trainSFeatures=gamoConv.predict(trainS, batch_size=500)
    for i1 in range(imbClsNum):
        inClassPoints[i1]=np.expand_dims(np.transpose(trainSFeatures[classMap[i1]]), axis=0)

    for j in range(numBatches):
        x1, x2=batchDiv[j, 0], batchDiv[j+1, 0]
        validR=np.ones((bSStore[j, 0],1))-np.random.uniform(0,0.1, size=(bSStore[j, 0], 1))
        mlp.train_on_batch(trainSFeatures[x1:x2], labelsCat[x1:x2])
        dis.train_on_batch([trainSFeatures[x1:x2], labelsCat[x1:x2]], validR)

        invalid=np.zeros((bSStore[j, 0], 1))+np.random.uniform(0, 0.1, size=(bSStore[j, 0], 1))
        randNoise=np.random.normal(0, 1, (bSStore[j, 0], latDim))
        fakeLabel=spp.randomLabelGen(toBalance, bSStore[j, 0], c)
        rLPerClass=spp.rearrange(fakeLabel, imbClsNum)
        fakePoints=np.zeros((bSStore[j, 0], 512))
        genFinal=gen.predict([randNoise, fakeLabel])
        for i1 in range(imbClsNum):
            if rLPerClass[i1].shape[0]!=0:
                temp=genFinal[rLPerClass[i1]]
                minorPoints=np.repeat(inClassPoints[i1], rLPerClass[i1].shape[0], axis=0)
                fakePoints[rLPerClass[i1]]=gen_processed[i1].predict([temp, minorPoints])

        mlp.train_on_batch(fakePoints, fakeLabel)
        dis.train_on_batch([fakePoints, fakeLabel], invalid)

        for i1 in range(imbClsNum):
            rLabel=np.zeros((genClassPoints, c))
            rLabel[:, i1]=1
            randNoise=np.random.normal(0, 1, (genClassPoints, latDim))
            oppositeLabel=np.ones((genClassPoints, c))-rLabel
            minorPoints=np.repeat(inClassPoints[i1], genClassPoints, axis=0)
```

```
            genP_mlp[i1].train_on_batch([randNoise, rLabel, minorPoints], oppositeLabel)
            genP_dis[i1].train_on_batch([randNoise, rLabel, minorPoints, rLabel], validA)

        if step%resSamplePd==0:
            saveStep=int(step//resSamplePd)

            pLabel=np.argmax(conv_mlp.predict(trainS), axis=1)
            acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTr)
            print('Train: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
            print('TPR: ', np.round(tpr, 2))
            acsaSaveTr[saveStep], gmSaveTr[saveStep], accSaveTr[saveStep]=acsa, gm, acc
            confMatSaveTr[saveStep]=confMat
            tprSaveTr[saveStep]=tpr

            pLabel=np.argmax(conv_mlp.predict(testS), axis=1)
            acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTs)
            print('Test: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
            print('TPR: ', np.round(tpr, 2))
            acsaSaveTs[saveStep], gmSaveTs[saveStep], accSaveTs[saveStep]=acsa, gm, acc
            confMatSaveTs[saveStep]=confMat
            tprSaveTs[saveStep]=tpr

        if step%modelSamplePd==0 and step!=0:
            direcPath=savePath+'gamo_models_'+str(step)
            if not os.path.exists(direcPath):
                os.makedirs(direcPath)
            dis.save(direcPath+'/DIS_'+str(step)+fileEnd)
            gen.save(direcPath+'/GEN_'+str(step)+fileEnd)
            mlp.save(direcPath+'/MLP_'+str(step)+fileEnd)
            gamoConv.save(direcPath+'/Conv_'+str(step)+fileEnd)
            for i in range(imbClsNum):
                gen_processed[i].save(direcPath+'/GenP_'+str(i)+'_'+str(step)+fileEnd)

        step=step+2
        if step>=max_step: break

pLabel=np.argmax(conv_mlp.predict(trainS), axis=1)
acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTr)
print('Performance on Train Set: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
print('TPR: ', np.round(tpr, 2))
acsaSaveTr[-1], gmSaveTr[-1], accSaveTr[-1]=acsa, gm, acc
confMatSaveTr[-1]=confMat
tprSaveTr[-1]=tpr

pLabel=np.argmax(conv_mlp.predict(testS), axis=1)
acsa, gm, tpr, confMat, acc=spp.indices(pLabel, labelTs)
print('Performance on Test Set: Step: ', step, 'ACSA: ', np.round(acsa, 4), 'GM: ', np.round(gm, 4))
print('TPR: ', np.round(tpr, 2))
acsaSaveTs[-1], gmSaveTs[-1], accSaveTs[-1]=acsa, gm, acc
confMatSaveTs[-1]=confMat
tprSaveTs[-1]=tpr

direcPath=savePath+'gamo_models_'+str(step)
if not os.path.exists(direcPath):
    os.makedirs(direcPath)
dis.save(direcPath+'/DIS_'+str(step)+fileEnd)
gen.save(direcPath+'/GEN_'+str(step)+fileEnd)
mlp.save(direcPath+'/MLP_'+str(step)+fileEnd)
gamoConv.save(direcPath+'/Conv_'+str(step)+fileEnd)
for i in range(imbClsNum):
    gen_processed[i].save(direcPath+'/GenP_'+str(i)+'_'+str(step)+fileEnd)

resSave=savePath+'Results'
np.savez(resSave, acsa=acsa, gm=gm, tpr=tpr, confMat=confMat, acc=acc)
recordSave=savePath+'Record'
np.savez(recordSave, acsaSaveTr=acsaSaveTr, gmSaveTr=gmSaveTr, accSaveTr=accSaveTr,
    acsaSaveTs=acsaSaveTs, gmSaveTs=gmSaveTs, accSaveTs=accSaveTs, confMatSaveTr=confMatSaveTr,
    confMatSaveTs=confMatSaveTs, tprSaveTr=tprSaveTr, tprSaveTs=tprSaveTs)
```

GAMO network for Fashion-MNIST: fashion_mnist_net.py

Save the following functions in a single file named "fashion_mnist_net.py", which will be imported in the "main".

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division
```

```python
from keras import backend as K
from keras.layers import Input, Dense, RepeatVector, Lambda, Multiply, Conv2D, Conv2DTranspose
from keras.layers import BatchNormalization, Concatenate, AveragePooling2D, Flatten, Reshape
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Model

def fMnistGamoGenCreate(latDim):
    noise=Input(shape=(latDim,))
    labels=Input(shape=(10,))
    gamoGenInput=Concatenate()([noise, labels])

    x=Dense(256, activation='relu')(gamoGenInput)
    x=BatchNormalization(momentum=0.9)(x)

    x=Dense(64, activation='relu')(x)
    gamoGenFinal=BatchNormalization(momentum=0.9)(x)

    gamoGen=Model([noise, labels], gamoGenFinal)
    gamoGen.summary()
    return gamoGen

def fMnistGenProcessCreate(numMinor):
    ip1=Input(shape=(64, ))
    ip2=Input(shape=(512, numMinor))

    x=Dense(numMinor, activation='softmax')(ip1)
    x=RepeatVector(512)(x)

    x=Multiply()([x, ip2])
    genProcFinal=Lambda(lambda x: K.sum(x, axis=-1))(x)

    genProcess=Model([ip1, ip2], genProcFinal)
    return genProcess

def fMnistGamoConvCreate():
    ip1=Input(shape=(28, 28, 1))

    x=Conv2D(32, kernel_size=5, padding='same')(ip1)
    x=LeakyReLU(0.1)(x)
    x=AveragePooling2D(pool_size=(2, 2), strides=2, padding='same')(x)

    x=Conv2D(32, kernel_size=5, padding='same')(x)
    x=LeakyReLU(0.1)(x)
    x=AveragePooling2D(pool_size=(2, 2), strides=2, padding='same')(x)

    x=Flatten()(x)
    gamoConvFinal=Dense(512, activation='tanh')(x)

    gamoConv=Model(ip1, gamoConvFinal)
    gamoConv.summary()
    return gamoConv

def fMnistGamoDisCreate():
    imIn=Input(shape=(512,))
    labels=Input(shape=(10,))
    disInput=Concatenate()([imIn, labels])

    x=Dense(256)(disInput)
    x=LeakyReLU(alpha=0.1)(x)

    x=Dense(128)(x)
    x=LeakyReLU(alpha=0.1)(x)

    gamoDisFinal=Dense(1, activation='sigmoid')(x)

    gamoDis=Model([imIn, labels], gamoDisFinal)
    gamoDis.summary()
    return gamoDis

def fMnistGamoMlpCreate():
    ip1=Input(shape=(512,))

    x=Dense(256)(ip1)
    x=LeakyReLU(alpha=0.1)(x)

    x=Dense(128)(x)
    x=LeakyReLU(alpha=0.1)(x)

    gamoMlpFinal=Dense(10, activation='softmax')(x)

    gamoMlp=Model(ip1, gamoMlpFinal)
```

```
    gamoMlp.summary()
    return gamoMlp
```

GAMO supplementary functions for Fashion-MNIST: fashion_mnist_suppli.py

Save the following functions in a single file named "fashion_mnist_suppli.py", which will be imported in the "main".

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

import sys
import numpy as np
from sklearn.metrics import confusion_matrix
from keras.preprocessing import image
from keras.utils.np_utils import to_categorical

def relabel(labelTr, labelTs):
    unqLab, pInClass=np.unique(labelTr, return_counts=True)
    sortedUnqLab=np.argsort(pInClass, kind='mergesort')
    c=sortedUnqLab.shape[0]
    labelsNewTr=np.zeros((labelTr.shape[0],))-1
    labelsNewTs=np.zeros((labelTs.shape[0],))-1
    pInClass=np.sort(pInClass)
    classMap=list()
    for i in range(c):
        labelsNewTr[labelTr==unqLab[sortedUnqLab[i]]]=i
        labelsNewTs[labelTs==unqLab[sortedUnqLab[i]]]=i
        classMap.append(np.where(labelsNewTr==i)[0])
    return labelsNewTr, labelsNewTs, c, pInClass, classMap

def irFind(pInClass, c, irIgnore=1):
    ir=pInClass[-1]/pInClass
    imbalancedCls=np.arange(c)[ir>irIgnore]
    toBalance=np.subtract(pInClass[-1], pInClass[imbalancedCls])
    imbClsNum=toBalance.shape[0]
    if imbClsNum==0: sys.exit('No imbalanced classes found, exiting ...')
    return imbalancedCls, toBalance, imbClsNum, ir

def fileRead(fileName):
    dataTotal=np.loadtxt(fileName, delimiter=',')
    data=dataTotal[:, :-1]
    labels=dataTotal[:, -1]
    return data, labels

def indices(pLabel, tLabel):
    confMat=confusion_matrix(tLabel, pLabel)
    nc=np.sum(confMat, axis=1)
    tp=np.diagonal(confMat)
    tpr=tp/nc
    acsa=np.mean(tpr)
    gm=np.prod(tpr)**(1/confMat.shape[0])
    acc=np.sum(tp)/np.sum(nc)
    return acsa, gm, tpr, confMat, acc

def randomLabelGen(toBalance, batchSize, c):
    cumProb=np.cumsum(toBalance/np.sum(toBalance))
    bins=np.insert(cumProb, 0, 0)
    randomValue=np.random.rand(batchSize,)
    randLabel=np.digitize(randomValue, bins)-1
    randLabel_cat=to_categorical(randLabel)
    labelPadding=np.zeros((batchSize, c-randLabel_cat.shape[1]))
    randLabel_cat=np.hstack((randLabel_cat, labelPadding))
    return randLabel_cat

def batchDivision(n, batchSize):
    numBatches, residual=int(np.ceil(n/batchSize)), int(n%batchSize)
    if residual==0:
        residual=batchSize
    batchDiv=np.zeros((numBatches+1,1), dtype='int64')
    batchSizeStore=np.ones((numBatches, 1), dtype='int64')
    batchSizeStore[0:-1, 0]=batchSize
    batchSizeStore[-1, 0]=residual
    for i in range(numBatches):
        batchDiv[i]=i*batchSize
    batchDiv[numBatches]=batchDiv[numBatches-1]+residual
    return batchDiv, numBatches, batchSizeStore
```

```
def rearrange(labelsCat, numImbCls):
    labels=np.argmax(labelsCat, axis=1)
    arrangeMap=list()
    for i in range(numImbCls):
        arrangeMap.append(np.where(labels==i)[0])
    return arrangeMap
```

GAMO2pix main script for Fashion-MNIST

The GAMO2pix main script requires to import the custom network library called "fashion_mnist_gamo2pix_net.py", which we describe in the following section. Additionally, GAMO2pix requires the custom libraries used by GAMO for Fashion-MNIST as well as the saved networks.

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

import os
import numpy as np
import matplotlib.pyplot as plt
import fashion_mnist_gamo2pix_net as nt

import fashion_mnist_net as ntv
import fashion_mnist_suppli as spp

import keras.backend as K
from keras.layers import Input
from keras.preprocessing import image
from keras.models import Model, load_model
from keras.optimizers import Adam
from keras.utils.np_utils import to_categorical
from keras.losses import mean_squared_error

def vae_loss(y_true, y_pred):
    mse_loss=28*28*mean_squared_error(K.flatten(y_true), K.flatten(y_pred))
    kl_loss=-0.5*K.sum(1+z_sigma-K.square(z_mean)-K.exp(z_sigma), axis=-1)
    return K.mean(mse_loss+kl_loss)

# For selecting a GPU
# os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
# os.environ["CUDA_VISIBLE_DEVICES"]="1"

fileName=['fMnist_100_trainData.csv', 'fMnist_100_testData.csv']
folderStart='fMnist_100_Gamo/gamo_models_50000/'
imgFolderStart='fMnist_100_ImgGen'

genPath=folderStart+'GEN_50000_Model.h5'
genPPath=['GenP_0_50000_Model.h5', 'GenP_1_50000_Model.h5', 'GenP_2_50000_Model.h5',
    'GenP_3_50000_Model.h5', 'GenP_4_50000_Model.h5', 'GenP_5_50000_Model.h5',
    'GenP_6_50000_Model.h5', 'GenP_7_50000_Model.h5', 'GenP_8_50000_Model.h5',
    'GenP_9_50000_Model.h5']
convPath=folderStart+'Conv_50000_Model.h5'

fileEnd, savePath='_Model.h5', imgFolderStart+'/'

plt.ion()
adamOpt=Adam(0.0002, 0.5)
latDim, modelSamplePd, resSamplePd=100, 2000, 500

batchSize, max_step=32, 25000

trainS, labelTr=spp.fileRead(fileName[0])
testS, labelTs=spp.fileRead(fileName[1])

n, m=trainS.shape[0], testS.shape[0]
trainS, testS=(trainS-127.5)/127.5, (testS-127.5)/127.5
trainS, testS=np.reshape(trainS, (n, 28, 28, 1)), np.reshape(testS, (m, 28, 28, 1))

labelTr, labelTs, c, pInClass, _=spp.relabel(labelTr, labelTs)
imbalancedCls, toBalance, imbClsNum, ir=spp.irFind(pInClass, c)

labelsCat=to_categorical(labelTr)

shuffleIndex=np.random.choice(np.arange(n), size=(n,), replace=False)
trainS=trainS[shuffleIndex]
labelTr=labelTr[shuffleIndex]
labelsCat=labelsCat[shuffleIndex]
```

```python
classMap=list()
for i in range(c):
    classMap.append(np.where(labelTr==i)[0])

if not os.path.exists(imgFolderStart):
    os.makedirs(imgFolderStart)

for i in range(imbClsNum):

    encoder=nt.encoderCreate(convPath)
    encoder.trainable=False
    vaeEncoder=nt.vaeEncoderCreate(latDim)
    decoder=nt.decoderCreate(latDim)

    ip1=Input(shape=(28, 28, 1))
    op1=encoder(ip1)
    [op2, z_mean, z_sigma]=vaeEncoder(op1)
    op3=decoder(op2)
    autoencoder=Model(inputs=ip1, outputs=op3)
    autoencoder.compile(loss=vae_loss, optimizer=adamOpt)

    dataMinor=trainS[classMap[i], :]
    dataMinorFt=encoder.predict(dataMinor)
    numMinor=dataMinorFt.shape[0]
    tempData=np.copy(np.expand_dims(np.transpose(dataMinorFt), axis=0))

    gamoGen=load_model(genPath)
    gamoGenP=ntv.fMnistGenProcessCreate(numMinor)
    gamoGenP.load_weights(folderStart+genPPath[i])
    ea=nt.gamoExtractAlphas(numMinor)
    ea.set_weights(gamoGenP.get_weights())

    batchDiv, numBatches, bSStore=spp.batchDivision(numMinor, batchSize)
    fig1, axs1=plt.subplots(3, 2)
    fig2, axs2=plt.subplots(3, 3)

    picPath=savePath+'Pictures_'+str(i)
    if not os.path.exists(picPath):
        os.makedirs(picPath)

    direcPath=savePath+'models_'+str(i)
    if not os.path.exists(direcPath):
        os.makedirs(direcPath)

    step=0
    while step<max_step:
        for j in range(numBatches):

            repData=np.repeat(tempData, bSStore[j, 0], axis=0)

            x1, x2=batchDiv[j, 0], batchDiv[j+1, 0]
            autoencoder.train_on_batch(dataMinor[x1:x2], dataMinor[x1:x2])

            if step%resSamplePd==0:
                randInput=np.random.choice(numMinor, 3, replace=False)
                genImage=autoencoder.predict(dataMinor[randInput])
                for i1 in range(3):
                    realImageShow=image.array_to_img(dataMinor[randInput[i1]], scale=True)
                    genImageShow=image.array_to_img(genImage[i1], scale=True)
                    axs1[i1, 0].imshow(realImageShow)
                    axs1[i1, 1].imshow(genImageShow)
                    axs1[i1, 0].axis('off')
                    axs1[i1, 1].axis('off')
                plt.show()
                plt.pause(5)

                print('Train_Class: ', i, 'Step: ', step, ' completed')
                figFileName=picPath+'/Train_'+str(step)+'.png'
                fig1.savefig(figFileName, bbox_inches='tight')

                testNoise=np.random.normal(0, 1, (9, latDim))
                testLabel=np.zeros((9, c))
                testLabel[:, i]=1
                alphas=ea.predict(gamoGen.predict([testNoise, testLabel]))
                repData=np.repeat(tempData, 9, axis=0)
                gamoGenPData=np.sum(alphas*repData, axis=-1)
                [encoded, t1, t2]=vaeEncoder.predict(gamoGenPData)
                genImages=decoder.predict(encoded)
                for i1 in range(3):
                    for i2 in range(3):
                        img=image.array_to_img(genImages[(i1*3)+i2], scale=True)
```

```
                        axs2[i1,i2].imshow(img)
                        axs2[i1,i2].axis('off')
                plt.show()
                plt.pause(5)

                print('Test_Class: ', i, 'Step: ', step, ' completed')
                figFileName=picPath+'/Test_'+str(step)+'.png'
                fig2.savefig(figFileName, bbox_inches='tight')

            if step%modelSamplePd==0 and step!=0:
                vaeEncoder.save(direcPath+'/vaeEncoder_'+str(step)+fileEnd)
                decoder.save(direcPath+'/Decoder_'+str(step)+fileEnd)

            step=step+1
            if step>=max_step: break

    randInput=np.random.choice(numMinor, 3, replace=False)
    genImage=autoencoder.predict(dataMinor[randInput])
    for i1 in range(3):
        realImageShow=image.array_to_img(dataMinor[randInput[i1]], scale=True)
        genImageShow=image.array_to_img(genImage[i1], scale=True)
        axs1[i1, 0].imshow(realImageShow)
        axs1[i1, 1].imshow(genImageShow)
        axs1[i1, 0].axis('off')
        axs1[i1, 1].axis('off')
    plt.show()
    plt.pause(5)

    print('Train_Class: ', i, 'Step: ', step, ' completed')
    figFileName=picPath+'/Train_'+str(step)+'.png'
    fig1.savefig(figFileName, bbox_inches='tight')

    testNoise=np.random.normal(0, 1, (9, latDim))
    testLabel=np.zeros((9, c))
    testLabel[:, i]=1
    alphas=ea.predict(gamoGen.predict([testNoise, testLabel]))
    repData=np.repeat(tempData, 9, axis=0)
    gamoGenPData=np.sum(alphas*repData, axis=-1)
    [encoded, t1, t2]=vaeEncoder.predict(gamoGenPData)
    genImages=decoder.predict(encoded)
    for i1 in range(3):
        for i2 in range(3):
            img=image.array_to_img(genImages[(i1*3)+i2], scale=True)
            axs2[i1,i2].imshow(img)
            axs2[i1,i2].axis('off')
    plt.show()
    plt.pause(5)

    print('Test_Class: ', i, 'Step: ', step, ' completed')
    figFileName=picPath+'/Test_'+str(step)+'.png'
    fig2.savefig(figFileName, bbox_inches='tight')

    vaeEncoder.save(direcPath+'/vaeEncoder_'+str(step)+fileEnd)
    decoder.save(direcPath+'/Decoder_'+str(step)+fileEnd)
```

GAMO2pix network for Fashion-MNIST: fashion_mnist_gamo2pix_net.py

Save the following functions in a single file named "fashion_mnist_gamo2pix_net.py", which will be imported in the GAMO2pix "main" for Fashion-MNIST.

```
# For running in python 2.7+
from __future__ import print_function, unicode_literals
from __future__ import absolute_import, division

from keras import backend as K
from keras.layers import Input, Dense, RepeatVector, Conv2D, Reshape, Lambda
from keras.layers import BatchNormalization, AveragePooling2D, Flatten, Conv2DTranspose
from keras.layers.advanced_activations import LeakyReLU
from keras.models import Model, load_model
from keras import regularizers

import numpy as np
import fashion_mnist_net as nt

def encoderCreate(convPath):

    convCopy=nt.fMnistGamoConvCreate()
    convCopy.load_weights(convPath)
```

```
    convCopy.trainable=False

    return convCopy

def sampling(args):
    z_mean, z_sigma=args
    batch_size=K.shape(z_mean)[0]
    latDim=K.int_shape(z_mean)[1]
    epsilon=K.random_normal(shape=(batch_size, latDim))
    return z_mean+K.exp(z_sigma)*epsilon

def vaeEncoderCreate(latDim):

    ip1=Input(shape=(512,))

    z_mean=Dense(latDim, name='z_mean')(ip1)
    z_sigma=Dense(latDim, name='z_sigma')(ip1)
    z=Lambda(sampling)([z_mean, z_sigma])

    vaeEncoder=Model(ip1, [z, z_mean, z_sigma])

    return vaeEncoder

def decoderCreate(latDim):

    ip1=Input(shape=(latDim,))
    x=Dense(7*7*32)(ip1)
    x=LeakyReLU(0.1)(x)
    x=Reshape((7, 7, 32))(x)

    x=Conv2DTranspose(32, kernel_size=4, strides=2, padding='same')(x)
    x=LeakyReLU(0.1)(x)

    x=Conv2DTranspose(32, kernel_size=4, strides=2, padding='same')(x)
    x=LeakyReLU(0.1)(x)

    decodeOut=Conv2D(1, kernel_size=4, strides=1, padding='same', activation='tanh')(x)

    decoder=Model(ip1, decodeOut)

    decoder.summary()

    return decoder

def gamoExtractAlphas(numMinor):
    ip1=Input(shape=(64, ))

    x=Dense(numMinor, activation='softmax')(ip1)
    op1=RepeatVector(512)(x)

    extAlphas=Model(ip1, op1)

    return extAlphas
```