torchgan Documentation

Release v0.0.2

Avik Pal and Aniket Das

Jul 03, 2019

GETTING STARTED

The `torchgan` package consists of various generative adversarial networks and utilities that have been found useful in training them. This package provides an easy to use API which can be used to train popular GANs as well as develop newer variants. The core idea behind this project is to facilitate easy and rapid generative adversarial model research.

CHAPTER 1

Installation

Follow the following instructions to set up `torchgan`. `Torchgan` is tested and known to work on major `linux distrubutions`. If you face any problem with other `operating systems` feel free to file an `issue`.

1.1 Pip Installation

To install the last released version make a pip install.

```
$ pip3 install torchgan
```

For the latest version.

```
$ pip3 install git+https://github.com/torchgan/torchgan.git
```

1.2 Conda Installation

Installing via `conda` is currently unavailable. It will be available once we are at `v0.1`

1.3 Install from Source

```
$ git clone https://github.com/torchgan/torchgan
$ cd torchgan
$ python setup.py install
```

CHAPTER 2

Dependencies

2.1 Mandatory Dependencies

- Numpy
- Pytorch 0.4.1
- Torchvision

2.2 Optional Dependencies

- TensorboardX : For Tensorboard Logging. Install using `pip install tensorboardX`.
- Visdom : For logging using Xisdom. Install using `pip install visdom`.

CHAPTER 3

Philosophy

Nowadays there are a lot of repositories for training Generative Adversarial Networks in Pytorch, however, there are some challenges which still remain:

- Most of these are not documented
- Majority of them are not maintained
- They are built without considering the ease of usage in mind
- These are not properly tested and often are not supported by the newer releases of Pytorch
- There is no proper unified API among these repositories

The idea of this framework is to provide an elegant design to solve issues regarding training and visualizing GANs. The design principles of this framework are the following:

- A common unified API for designing GANs
- Well documented code and API
- Proper examples to facilitate ease of use
- Easy to integrate with your applications
- Provide a easy API for fast prototyping and research
- Provide advanced features without taking away the ability to customize from users
- Presence of popular loss functions, metrics and modules from cutting edge research

CHAPTER 4

Contributing

4.1 Contribution Guidelines

Contributions in all forms are always welcome. Follow the following guidelines while contributing :-

1. If contributing a `new feature`, first open an issue on github. Describe the feature and provide some references. Also clarify why it shall be a good feature to have in the `core library` and not simply as a `representative example`.
2. If submitting a `bug fix`, file the issue on github. Make sure the bug exists on the `master`.
3. If submitting a `new model`, open a PR in the `model zoo` repository. Follow the contribution guidelines present there.
4. Also fell free to submit `documentation changes`.

For you PR to be merged it must strictly adhere to the style guidelines, we use `flake8` for that purpose. Also all existing tests must pass. No breaking changes will be accepted unless when we are making a change in the `major version`. Also be sure to add `tests` and `documentation` for any code that you submit.

4.2 Contributors

We are thankful to all our contributors! For a complete list of contributors, please see the official Contributors List on github.

CHAPTER 5

Starter Example

As a starter example we will try to train a `DCGAN` on `CIFAR-10`. `DCGAN` is in-built into to the library, but let it not fool you into believing that we can only use this package for some fixed limited tasks. This library is fully customizable. For that have a look at the `Examples`.

But for now let us just use this as a small demo example

First we import the necessary files

```
import torch
import torchvision
from torch.optim import Adam
import torch.utils.data as data
import torchvision.datasets as dsets
import torchvision.transforms as transforms
import torchgan
from torchgan.models import DCGANGenerator, DCGANDiscriminator
from torchgan.losses import MinimaxGeneratorLoss, MinimaxDiscriminatorLoss,
from torchgan.trainer import Trainer
```

Now write a function which returns the `data loader` for `CIFAR10`.

```
def cifar10_dataloader():
    train_dataset = dsets.CIFAR10(root='./cifar10', train=True,
                                  transform=transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize(mean = (0.5, 0.5, 0.5), std␣
↪= (0.5, 0.5, 0.5))]),
                                  download=True)
    train_loader = data.DataLoader(train_dataset, batch_size=128, shuffle=True)
    return train_loader
```

Now lets us create the `Trainer` object and pass the `data loader` to it.

```
trainer = Trainer({"generator": {"name": DCGANGenerator, "args": {"out_channels": 3,
↪"step_channels": 16}, "optimizer": {"name": Adam, "args": {"lr": 0.0002, "betas":␣
↪(0.5, 0.999)}}},
```

(continues on next page)

(continued from previous page)

```
                    "discriminator": {"name": DCGANDiscriminator, "args": {"in_channels
↪": 3, "step_channels": 16}, "optimizer": {"name": Adam, "args": {"lr": 0.0002,
↪"betas": (0.5, 0.999)}}}},
                   [MinimaxGeneratorLoss(), MinimaxDiscriminatorLoss()],
                   sample_size=64, epochs=20)

trainer(cifar10_dataloader())
```

Now log into `tensorboard` and visualize the training process.

CHAPTER 6

License

MIT License:

```
Copyright (c) 2018 - Present Avik Pal and Aniket Das

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

  1. The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
  2. Every publication and presentation for which work based on the Program or
its output has been used must contain an appropriate citation and acknowledgement
of the authors of this Program.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

CHAPTER 7

torchgan.layers

This layers subpackage is a collection of popular building blocks for GAN architectures. Currently the following blocks are supported:

7.1 Residual Blocks

7.1.1 ResidualBlock2d

class torchgan.layers.**ResidualBlock2d**(*filters*, *kernels*, *strides=None*, *paddings=None*, *nonlinearity=None*, *batchnorm=True*, *shortcut=None*, *last_nonlinearity=None*)

Residual Block Module as described in "Deep Residual Learning for Image Recognition by He et. al."

The output of the residual block is computed in the following manner:

$$output = activation(layers(x) + shortcut(x))$$

where

- x : Input to the Module
- $layers$: The feed forward network
- $shortcut$: The function to be applied along the skip connection
- $activation$: The activation function applied at the end of the residual block

Parameters

- **filters** (*list*) – A list of the filter sizes. For ex, if the input has a channel dimension of 16, and you want 3 convolution layers and the final output to have a channel dimension of 16, then the list would be [16, 32, 64, 16].
- **kernels** (*list*) – A list of the kernel sizes. Each kernel size can be an integer or a tuple, similar to Pytorch convention. The length of the `kernels` list must be 1 less than the `filters` list.
- **strides** (*list, optional*) – A list of the strides for each convolution layer.
- **paddings** (*list, optional*) – A list of the padding in each convolution layer.
- **nonlinearity** (*torch.nn.Module, optional*) – The activation to be used after every convolution layer.
- **batchnorm** (*bool, optional*) – If set to `False`, batch normalization is not used after every convolution layer.
- **shortcut** (*torch.nn.Module, optional*) – The function to be applied on the input along the skip connection.
- **last_nonlinearity** (*torch.nn.Module, optional*) – The activation to be applied at the end of the residual block.

forward(*x*)

Computes the output of the residual block

Parameters **x** (*torch.Tensor*) – A 4D Torch Tensor which is the input to the Residual Block.

Returns 4D Torch Tensor after applying the desired functions as specified while creating the object.

7.1.2 ResidualBlockTranspose2d

class torchgan.layers.**ResidualBlockTranspose2d**(*filters*, *kernels*, *strides=None*, *paddings=None*, *nonlinearity=None*, *batchnorm=True*, *shortcut=None*, *last_nonlinearity=None*)

A customized version of Residual Block having Conv Transpose layers instead of Conv layers.

The output of this block is computed in the following manner:

$$output = activation(layers(x) + shortcut(x))$$

where

- x : Input to the Module
- $layers$: The feed forward network
- $shortcut$: The function to be applied along the skip connection
- $activation$: The activation function applied at the end of the residual block

Parameters

- **filters** (*list*) – A list of the filter sizes. For ex, if the input has a channel dimension of 16, and you want 3 transposed convolution layers and the final output to have a channel dimension of 16, then the list would be [16, 32, 64, 16].
- **kernels** (*list*) – A list of the kernel sizes. Each kernel size can be an integer or a tuple, similar to Pytorch convention. The length of the `kernels` list must be 1 less than the `filters` list.
- **strides** (*list, optional*) – A list of the strides for each convolution layer.
- **paddings** (*list, optional*) – A list of the padding in each convolution layer.
- **nonlinearity** (*torch.nn.Module, optional*) – The activation to be used after every convolution layer.
- **batchnorm** (*bool, optional*) – If set to `False`, batch normalization is not used after every convolution layer.
- **shortcut** (*torch.nn.Module, optional*) – The function to be applied on the input along the skip connection.
- **last_nonlinearity** (*torch.nn.Module, optional*) – The activation to be applied at the end of the residual block.

forward(*x*)

Computes the output of the residual block

Parameters **x** (*torch.Tensor*) – A 4D Torch Tensor which is the input to the Transposed Residual Block.

Returns 4D Torch Tensor after applying the desired functions as specified while creating the object.

7.2 Densenet Blocks

7.2.1 BasicBlock2d

class torchgan.layers.**BasicBlock2d**(*in_channels*, *out_channels*, *kernel*, *stride=1*, *padding=0*, *batchnorm=True*, *nonlinearity=None*)

Basic Block Module as described in "Densely Connected Convolutional Networks by Huang et. al."

The output is computed by `concatenating` the `input` tensor to the `output` tensor (of the internal model) along the `channel` dimension.

The internal model is simply a sequence of a `Conv2d` layer and a `BatchNorm2d` layer, if activated.

Parameters

- **in_channels** (*int*) – The channel dimension of the input tensor.
- **out_channels** (*int*) – The channel dimension of the output tensor.
- **kernel** (*int, tuple*) – Size of the Convolutional Kernel.
- **stride** (*int, tuple, optional*) – Stride of the Convolutional Kernel.
- **padding** (*int, tuple, optional*) – Padding to be applied on the input tensor.
- **batchnorm** (*bool, optional*) – If `True`, batch normalization shall be performed.
- **nonlinearity** (*torch.nn.Module, optional*) – Activation to be applied. Defaults to `torch.nn.LeakyReLU`.

forward(*x*)

Computes the output of the basic dense block

Parameters **x** (*torch.Tensor*) – The input tensor having channel dimension same as `in_channels`.

Returns 4D Tensor by concatenating the input to the output of the internal model.

7.2.2 BottleneckBlock2d

class torchgan.layers.**BottleneckBlock2d**(*in_channels*, *out_channels*, *kernel*, *stride=1*, *padding=0*, *bottleneck_channels=None*, *batchnorm=True*, *nonlinearity=None*)

Bottleneck Block Module as described in "Densely Connected Convolutional Networks by Huang et. al."

The output is computed by `concatenating` the `input` tensor to the `output` tensor (of the internal model) along the `channel` dimension.

The internal model is simply a sequence of 2 `Conv2d` layers and 2 `BatchNorm2d` layers, if activated. This Module is much more computationally efficient than the `BasicBlock2d`, and hence is more recommended.

Parameters

- **in_channels** (*int*) – The channel dimension of the input tensor.
- **out_channels** (*int*) – The channel dimension of the output tensor.
- **kernel** (*int, tuple*) – Size of the Convolutional Kernel.
- **stride** (*int, tuple, optional*) – Stride of the Convolutional Kernel.
- **padding** (*int, tuple, optional*) – Padding to be applied on the input tensor.

- **bottleneck_channels** (*int, optional*) – The channels in the intermediate convolutional layer. A higher value will make learning of more complex functions possible. Defaults to `4 * in_channels`.
- **batchnorm** (*bool, optional*) – If `True`, batch normalization shall be performed.
- **nonlinearity** (*torch.nn.Module, optional*) – Activation to be applied. Defaults to `torch.nn.LeakyReLU`.

forward(*x*)

Computes the output of the bottleneck dense block

Parameters **x** (*torch.Tensor*) – The input tensor having channel dimension same as `in_channels`.

Returns 4D Tensor by concatenating the input to the output of the internal model.

7.2.3 TransitionBlock2d

class torchgan.layers.**TransitionBlock2d**(*in_channels*, *out_channels*, *kernel*, *stride=1*, *padding=0*, *batchnorm=True*, *nonlinearity=None*)

Transition Block Module as described in "Densely Connected Convolutional Networks by Huang et. al."

This is a simple `Sequential` model of a `Conv2d` layer and a `BatchNorm2d` layer, if activated.

Parameters

- **in_channels** (*int*) – The channel dimension of the input tensor.
- **out_channels** (*int*) – The channel dimension of the output tensor.
- **kernel** (*int, tuple*) – Size of the Convolutional Kernel.
- **stride** (*int, tuple, optional*) – Stride of the Convolutional Kernel.
- **padding** (*int, tuple, optional*) – Padding to be applied on the input tensor.
- **batchnorm** (*bool, optional*) – If `True`, batch normalization shall be performed.
- **nonlinearity** (*torch.nn.Module, optional*) – Activation to be applied. Defaults to `torch.nn.LeakyReLU`.

forward(*x*)

Computes the output of the transition block

Parameters **x** (*torch.Tensor*) – The input tensor having channel dimension same as `in_channels`.

Returns 4D Tensor by applying the `model` on `x`.

7.2.4 TransitionBlockTranspose2d

class torchgan.layers.**TransitionBlockTranspose2d**(*in_channels*, *out_channels*, *kernel*, *stride=1*, *padding=0*, *batchnorm=True*, *nonlinearity=None*)

Transition Block Transpose Module is constructed by simply reversing the effect of Transition Block Module. We replace the `Conv2d` layers by `ConvTranspose2d` layers.

Parameters

- **in_channels** (*int*) – The channel dimension of the input tensor.
- **out_channels** (*int*) – The channel dimension of the output tensor.

- **kernel** (*int, tuple*) – Size of the Convolutional Kernel.
- **stride** (*int, tuple, optional*) – Stride of the Convolutional Kernel.
- **padding** (*int, tuple, optional*) – Padding to be applied on the input tensor.
- **batchnorm** (*bool, optional*) – If `True`, batch normalization shall be performed.
- **nonlinearity** (*torch.nn.Module, optional*) – Activation to be applied. Defaults to `torch.nn.LeakyReLU`.

forward(*x*)

Computes the output of the transition block transpose

Parameters **x** (*torch.Tensor*) – The input tensor having channel dimension same as `in_channels`.

Returns 4D Tensor by applying the `model` on `x`.

7.2.5 DenseBlock2d

class torchgan.layers.**DenseBlock2d**(*depth*, *in_channels*, *growth_rate*, *block*, *kernel*, *stride=1*, *padding=0*, *batchnorm=True*, *nonlinearity=None*)

Dense Block Module as described in "Densely Connected Convolutional Networks by Huang et. al."

Parameters

- **depth** (*int*) – The total number of `blocks` that will be present.
- **in_channels** (*int*) – The channel dimension of the input tensor.
- **growth_rate** (*int*) – The rate at which the channel dimension increases. The output of the module has a channel dimension of size `in_channels + depth * growth_rate`.
- **block** (*torch.nn.Module*) – Should be once of the Densenet Blocks. Forms the building block for the Dense Block.
- **kernel** (*int, tuple*) – Size of the Convolutional Kernel.
- **stride** (*int, tuple, optional*) – Stride of the Convolutional Kernel.
- **padding** (*int, tuple, optional*) – Padding to be applied on the input tensor.
- **batchnorm** (*bool, optional*) – If `True`, batch normalization shall be performed.
- **nonlinearity** (*torch.nn.Module, optional*) – Activation to be applied. Defaults to `torch.nn.LeakyReLU`.

forward(*x*)

Computes the output of the transition block transpose

Parameters **x** (*torch.Tensor*) – The input tensor having channel dimension same as `in_channels`.

Returns 4D Tensor by applying the `model` on `x`.

7.3 Self Attention

7.3.1 SelfAttention2d

class torchgan.layers.**SelfAttention2d**(*input_dims*, *output_dims=None*, *return_attn=False*)

Self Attention Module as proposed in the paper "Self-Attention Generative Adversarial Networks by Han Zhang et. al."

$$attention = softmax((query(x))^T * key(x))$$

$$output = \gamma * value(x) * attention + x$$

where

- $query$: 2D Convolution Operation
- key : 2D Convolution Operation
- $value$: 2D Convolution Operation
- x : Input

Parameters

- **input_dims** (*int*) – The input channel dimension in the input `x`.
- **output_dims** (*int, optional*) – The output channel dimension. If `None` the output channel value is computed as `input_dims // 8`. So if the `input_dims` is **less than 8** then the layer will give an error.
- **return_attn** (*bool, optional*) – Set it to `True` if you want the attention values to be returned.

forward(*x*)

Computes the output of the Self Attention Layer

Parameters **x** (*torch.Tensor*) – A 4D Tensor with the channel dimension same as `input_dims`.

Returns A tuple of the `output` and the `attention` if `return_attn` is set to `True` else just the `output` tensor.

7.4 Spectral Normalization

7.4.1 SpectralNorm2d

class torchgan.layers.**SpectralNorm2d**(*module*, *name='weight'*, *power_iterations=1*)

2D Spectral Norm Module as described in "Spectral Normalization for Generative Adversarial Networks by Miyato et. al." The spectral norm is computed using `power iterations`.

Computation Steps:

$$v_{t+1} = \frac{W^T W v_t}{||W^T W v_t||} = \frac{(W^T W)^t v}{||(W^T W)^t v||}$$

$$u_{t+1} = W v_t$$

$$v_{t+1} = W^T u_{t+1}$$

$$Norm(W) = ||Wv|| = u^T W v$$

$$Output = \frac{W}{Norm(W)} = \frac{W}{u^T W v}$$

Parameters

- **module** (*torch.nn.Module*) – The Module on which the Spectral Normalization needs to be applied.
- **name** (*str, optional*) – The attribute of the `module` on which normalization needs to be performed.
- **power_iterations** (*int, optional*) – Total number of iterations for the norm to converge. `1` is usually enough given the weights vary quite gradually.

Example

```
>>> layer = SpectralNorm2d(Conv2d(3, 16, 1))
>>> x = torch.rand(1, 3, 10, 10)
>>> layer(x)
```

forward(**args*)

Computes the output of the `module` and appies spectral normalization to the `name` attribute of the `module`.

Returns The output of the `module`.

7.5 Minibatch Discrimination

7.5.1 MinibatchDiscrimination1d

class torchgan.layers.**MinibatchDiscrimination1d**(*in_features*, *out_features*, *intermediate_features=16*)

1D Minibatch Discrimination Module as proposed in the paper "Improved Techniques for Training GANs by Salimans et. al."

Allows the Discriminator to easily detect mode collapse by augmenting the activations to the succeeding layer with side information that allows it to determine the 'closeness' of the minibatch examples with each other

$$M_i = T * f(x_i)$$

$$c_b(x_i, x_j) = \exp(-||M_{i,b} - M_{j,b}||_1) \in \mathbb{R}.$$

$$o(x_i)_b = \sum_{j=1}^{n} c_b(x_i, x_j) \in \mathbb{R}$$

$$o(x_i) = \left[o(x_i)_1, o(x_i)_2, \ldots, o(x_i)_B\right] \in \mathbb{R}^B$$

$$o(X) \in \mathbb{R}^{n \times B}$$

This is followed by concatenating $o(x_i)$ and $f(x_i)$

where

- $f(x_i) \in \mathbb{R}^A$: Activations from an intermediate layer
- $f(x_i) \in \mathbb{R}^A$: Parameter Tensor for generating minibatch discrimination matrix

Parameters

- **in_features** (*int*) – Features input corresponding to dimension A
- **out_features** (*int*) – Number of output features that are to be concatenated corresponding to dimension B
- **intermediate_features** (*int*) – Intermediate number of features corresponding to dimension C

Returns A Tensor of size $(N, in_features + out_features)$ where N is the batch size

forward(*x*)

Computes the output of the Minibatch Discrimination Layer

Parameters **x** (*torch.Tensor*) – A Torch Tensor of dimensions :math: *(N, infeatures)*

Returns math: *(N,infeatures + outfeatures)* after applying Minibatch Discrimination

Return type 3D Torch Tensor of size

7.6 Virtual Batch Normalization

7.6.1 VirtualBatchNorm

class torchgan.layers.**VirtualBatchNorm**(*in_features*, *eps=1e-05*)

Virtual Batch Normalization Module as proposed in the paper "Improved Techniques for Training GANs by Salimans et. al."

Performs Normalizes the features of a batch based on the statistics collected on a reference batch of samples that are chosen once and fixed from the start, as opposed to regular batch normalization that uses the statistics of the batch being normalized

Virtual Batch Normalization requires that the size of the batch being normalized is at least a multiple of (and ideally equal to) the size of the reference batch. Keep this in mind while choosing the batch size in `torch.utils.data.DataLoader` or use `drop_last=True`

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

where

- x : Batch Being Normalized
- x_{ref} : Reference Batch

Parameters

- **in_features** (*int*) – Size of the input dimension to be normalized
- **eps** (*float, optional*) – Value to be added to variance for numerical stability while normalizing

forward(*x*)

Computes the output of the Virtual Batch Normalization

Parameters **x** (*torch.Tensor*) – A Torch Tensor of dimension at least 2 which is to be Normalized

Returns Torch Tensor of the same dimension after normalizing with respect to the statistics of the reference batch

CHAPTER 8

torchgan.logging

This subpackage provides strong visualization capabilities using a variety of Backends. It is strongly integrated with the Trainer. The `Logger` supports a variety of configurations and customizations.

Note: The Logger API is currently deeply integrated with the `Trainer` and hence might not be a very pleasant thing to use externally. However, work is being done to make them as much independent as possible and support extendibility of the Logger. Hence, this is expected to improve in the future.

8.1 Backends

Currently available backends are:

1. **TensorboardX:** To enable this set the `TENSORBOARD_LOGGING` to 1. If the package is pre-installed on your system, this variable is enabled by default.

   If you want to disable this then `os.environ["TENSORBOARD_LOGGING"] = "0"`. Make sure to do it before loading torchgan.

Once the logging begins, you need to start a tensorboard server using this code `tensorboard --logdir runs`.

2. **Visdom:** To enable this set the `VISDOM_LOGGING` to 1. If the package is pre-installed on your system, this variable is enabled by default.

   If you want to disable this then `os.environ["VISDOM_LOGGING"] = "0"`. We recommend using visdom if you need to save your plots. In general tensorboard support is better in terms of the image display.

   Warning: If this package is present and **VISDOM_LOGGING** is set to 1, then a server must be started using the command *python -m visdom.server* before the Training is started. Otherwise the code will simply crash.

3. **Console:** The details of training are printed on the console. This is enabled by default but can be turned off by `os.environ["CONSOLE_LOGGING"] = "0"`.

Add more backends for visualization is a work-in-progress.

Note: It is the responsibility of the user to install the necessary packages needed for visualization. If the necessary packages are missing the logging will not occur or if the user trys to force it the program will terminate with an error message.

Note: It is recommended to use only **1 logging service** (apart from the **Console**). Using multiple Logging services might affect the training time. It is recommended to use **Visdom** only if the plots are to be downloaded easily.

8.2 Logger

class torchgan.logging.**Logger**(*trainer*, *losses_list*, *metrics_list=None*, *visdom_port=8097*, *log_dir=None*, *writer=None*, *nrow=8*, *test_noise=None*)

Base Logger class. It controls the executions of all the Visualizers and is deeply integrated with the functioning of the Trainer.

Note: The `Logger` has been designed to be controlled internally by the `Trainer`. It is recommended that the user does not attempt to use it externally in any form.

Warning: This `Logger` is meant to work on the standard Visualizers available. Work is being done to support custom Visualizers in a clean way. But currently it is not possible to do so.

Parameters

- **trainer** (torchgan.trainer.Trainer) – The base trainer used for training.
- **losses_list** (*list*) – A list of the Loss Functions that need to be minimized. For a list of pre-defined losses look at `torchgan.losses`. All losses in the list must be a subclass of atleast `GeneratorLoss` or `DiscriminatorLoss`.

- **metrics_list** (*list, optional*) – List of Metric Functions that need to be logged. For a list of pre-defined metrics look at `torchgan.metrics`. All losses in the list must be a subclass of `EvaluationMetric`.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. A deafult server is started at port `8097`. So manually a new server has to be started if the post is changed. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.
- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.
- **test_noise** (*torch.Tensor, optional*) – If provided then it will be used as the noise for image sampling.
- **nrow** (*int, optional*) – Number of rows in which the image is to be stored.

close()
Turns off the tensorboard `SummaryWriter` if it were created.

get_grad_viz()
Get the GradientVisualize object.

get_loss_viz()
Get the LossVisualize object.

get_metric_viz()
Get the MetricVisualize object.

register(*visualize*, **args*, *mid_epoch=True*, ***kwargs*)
Register a new `Visualize` object with the Logger.

Parameters

- **visualize** (`torchgan.logging.Visualize`) – Class name of the visualizer.
- **mid_epoch** (*bool, optional*) – Set it to `False` if it is to be executed once the epoch is over. Otherwise it is executed after every call to the `train_iter`.

run_end_epoch(*trainer*, *epoch*, *time_duration*, **args*)
Runs the Visualizers at the end of one epoch.

Parameters

- **trainer** (`torchgan.trainer.Trainer`) – The base trainer used for training.
- **epoch** (*int*) – The epoch number which was completed.

run_mid_epoch(*trainer*, **args*)
Runs the Visualizers after every call to the `train_iter`.

Parameters **trainer** (`torchgan.trainer.Trainer`) – The base trainer used for training.

8.3 Visualization

8.3.1 Visualize

class torchgan.logging.**Visualize**(*visualize_list*, *visdom_port=8097*, *log_dir=None*, *writer=None*)

Base class for all Visualizations.

Parameters

- **visualize_list** (*list, optional*) – List of the functions needed for visualization.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. The visdom server needs to be manually started at this port else an error will be thrown and the code will crash. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.
- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.

log_console()

Console logging function. Needs to be defined in the subclass

Raises **NotImplementedError** –

log_tensorboard()

Tensorboard logging function. Needs to be defined in the subclass

Raises **NotImplementedError** –

log_visdom()

Visdom logging function. Needs to be defined in the subclass

Raises **NotImplementedError** –

step_update()

Helper function which updates the step at the end of one print iteration.

8.3.2 LossVisualize

class torchgan.logging.**LossVisualize**(*visualize_list*, *visdom_port=8097*, *log_dir=None*, *writer=None*)

This class provides the Visualizations for Generator and Discriminator Losses.

Parameters

- **visualize_list** (*list, optional*) – List of the functions needed for visualization.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. The visdom server needs to be manually started at this port else an error will be thrown and the code will crash. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.
- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.

log_console(*running_losses*)

Console logging function. This function logs the mean `generator` and `discriminator` losses.

Parameters **running_losses** (*dict*) – A dict with 2 items namely, `Running Discriminator Loss`, and `Running Generator Loss`.

log_tensorboard(*running_losses*)
Tensorboard logging function. This function logs the following:

- `Running Discriminator Loss`
- `Running Generator Loss`
- `Running Losses`
- Loss Values of the individual Losses.

Parameters **running_losses** (*dict*) – A dict with 2 items namely, `Running Discriminator Loss`, and `Running Generator Loss`.

log_visdom(*running_losses*)
Visdom logging function. This function logs the following:

- `Running Discriminator Loss`
- `Running Generator Loss`
- `Running Losses`
- Loss Values of the individual Losses.

Parameters **running_losses** (*dict*) – A dict with 2 items namely, `Running Discriminator Loss`, and `Running Generator Loss`.

8.3.3 GradientVisualize

class torchgan.logging.**GradientVisualize**(*visualize_list*, *visdom_port=8097*, *log_dir=None*, *writer=None*)
This class provides the Visualizations for the Gradients.

Parameters

- **visualize_list** (*list, optional*) – List of the functions needed for visualization.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. The visdom server needs to be manually started at this port else an error will be thrown and the code will crash. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.
- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.

log_console(*name*)
Console logging function. This function logs the mean gradients.

Parameters **name** (*str*) – Name of the model whose gradients are to be logged.

log_tensorboard(*name*)
Tensorboard logging function. This function logs the values of the individual gradients.

Parameters **name** (*str*) – Name of the model whose gradients are to be logged.

log_visdom(*name*)
Visdom logging function. This function logs the values of the individual gradients.

Parameters **name** (*str*) – Name of the model whose gradients are to be logged.

report_end_epoch()
Prints to the console at the end of the epoch.

update_grads(*name*, *model*, *eps=1e-05*)
Updates the gradient logs.

Parameters

- **name** (*str*) – Name of the model.
- **model** (*torch.nn.Module*) – Either a `torchgan.models.Generator` or a `torchgan.models.Discriminator` or their subclass.
- **eps** (*float, optional*) – Tolerance value.

8.3.4 MetricVisualize

class torchgan.logging.**MetricVisualize**(*visualize_list*, *visdom_port=8097*, *log_dir=None*, *writer=None*)
This class provides the Visualizations for Metrics.

Parameters

- **visualize_list** (*list, optional*) – List of the functions needed for visualization.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. The visdom server needs to be manually started at this port else an error will be thrown and the code will crash. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.
- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.

log_console()
Console logging function. This function logs the mean metrics.

log_tensorboard()
Tensorboard logging function. This function logs the values of the individual metrics.

log_visdom()
Visdom logging function. This function logs the values of the individual metrics.

8.3.5 ImageVisualize

class torchgan.logging.**ImageVisualize**(*trainer*, *visdom_port=8097*, *log_dir=None*, *writer=None*, *test_noise=None*, *nrow=8*)
This class provides the Logging for the Images.

Parameters

- **trainer** (`torchgan.trainer.Trainer`) – The base trainer used for training.
- **visdom_port** (*int, optional*) – Port to log using `visdom`. The visdom server needs to be manually started at this port else an error will be thrown and the code will crash. This is ignored if `VISDOM_LOGGING` is `0`.
- **log_dir** (*str, optional*) – Directory where TensorboardX should store the logs. This is ignored if `TENSORBOARD_LOGGING` is `0`.

- **writer** (*tensorboardX.SummaryWriter, optonal*) – Send a *SummaryWriter* if you don't want to start a new SummaryWriter.
- **test_noise** (*torch.Tensor, optional*) – If provided then it will be used as the noise for image sampling.
- **nrow** (*int, optional*) – Number of rows in which the image is to be stored.

log_console(*trainer*, *image*, *model*)

Saves a generated image at the end of an epoch. The path where the image is being stored is controlled by the `trainer`.

Parameters

- **trainer** (torchgan.trainer.Trainer) – The base trainer used for training.
- **image** (*Image*) – The generated image.
- **model** (*str*) – The name of the model which generated the `image`.

log_tensorboard(*trainer*, *image*, *model*)

Logs a generated image in tensorboard at the end of an epoch.

Parameters

- **trainer** (torchgan.trainer.Trainer) – The base trainer used for training.
- **image** (*Image*) – The generated image.
- **model** (*str*) – The name of the model which generated the `image`.

log_visdom(*trainer*, *image*, *model*)

Logs a generated image in visdom at the end of an epoch.

Parameters

- **trainer** (torchgan.trainer.Trainer) – The base trainer used for training.
- **image** (*Image*) – The generated image.
- **model** (*str*) – The name of the model which generated the `image`.

CHAPTER 9

torchgan.losses

This losses subpackage is a collection of popular loss functions used in the training of GANs. Currently the following losses are supported:

These losses are tested with the current available trainers. So if you need to implement you custom loss for using with the `trainer` it is recommended that you subclass the *GeneratorLoss* and *DiscriminatorLoss*.

Warning: The `override_train_ops` gets only the arguments that were received by the default `train_ops`. Hence it might not be a wise to use this very often. If this is used make sure to take into account the arguments and their order. A better alternative is to subclass the Loss and define a custom `train_ops`.

Warning: `train_ops` are designed to be used internally through the `Trainer`. Hence it is highly recommended that this function is not directly used by external sources, i.e. no call to this function is made outside the `Trainer`.

9.1 Loss

9.1.1 GeneratorLoss

class torchgan.losses.**GeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)

Base class for all generator losses.

Note: All Losses meant to be minimized for optimizing the Generator must subclass this.

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

set_arg_map(*value*)
Updates the `arg_map` for passing a different value to the `train_ops`.

Parameters **value** (*dict*) – A mapping of the `argument name` in the method signature and the variable name in the `Trainer` it corresponds to.

Note: If the `train_ops` signature is `train_ops(self, gen, disc, optimizer_generator, device, batch_size, labels=None)` then we need to map `gen` to `generator` and `disc` to `discriminator`. In this case we make the following function call `loss.set_arg_map({"gen": "generator", "disc": "discriminator"})`.

---

train_ops(*generator*, *discriminator*, *optimizer_generator*, *device*, *batch_size*, *labels=None*)
Defines the standard `train_ops` used by most losses. Losses which have a different training procedure can either `subclass` it **(recommended approach)** or make use of `override_train_ops` argument.

The `standard optimization algorithm` for the `generator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value = discriminator(fake)$
3. $loss = loss\_function(value)$
4. Backpropagate by computing $\nabla loss$
5. Run a step of the optimizer for generator

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `generator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.1.2 DiscriminatorLoss

class torchgan.losses.**DiscriminatorLoss**(*reduction='mean'*, *override_train_ops=None*)
Base class for all discriminator losses.

Note: All Losses meant to be minimized for optimizing the Discriminator must subclass this.

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

set_arg_map(*value*)

Updates the `arg_map` for passing a different value to the `train_ops`.

Parameters **value** (*dict*) – A mapping of the `argument name` in the method signature and the variable name in the `Trainer` it corresponds to.

Note: If the `train_ops` signature is `train_ops(self, gen, disc, optimizer_discriminator, device, batch_size, labels=None)` then we need to map `gen` to `generator` and `disc` to `discriminator`. In this case we make the following function call `loss.set_arg_map({"gen": "generator", "disc": "discriminator"})`.

---

train_ops(*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)

Defines the standard `train_ops` used by most losses. Losses which have a different training procedure can either `subclass` it **(recommended approach)** or make use of `override_train_ops` argument.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value_1 = discriminator(fake)$
3. $value_2 = discriminator(real)$
4. $loss = loss\_function(value_1, value_2)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.2 Least Squares Loss

9.2.1 LeastSquaresGeneratorLoss

class torchgan.losses.**LeastSquaresGeneratorLoss**(*reduction='mean', c=1.0, override_train_ops=None*)

Least Squares GAN generator loss from "Least Squares Generative Adversarial Networks by Mao et. al." paper

The loss can be described as

$$L(G) = \frac{(D(G(z)) - c)^2}{2}$$

where

- G : Generator
- D : Disrciminator
- c : target generator label
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **c** (*float, optional*) – Target generator label.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*dgz*)

Computes the loss for the given input.

Parameters **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.2.2 LeastSquaresDiscriminatorLoss

class torchgan.losses.**LeastSquaresDiscriminatorLoss**(*reduction='mean', a=0.0, b=1.0, override_train_ops=None*)

Least Squares GAN discriminator loss from "Least Squares Generative Adversarial Networks by Mao et. al." paper.

The loss can be described as:

$$L(D) = \frac{(D(x) - b)^2 + (D(G(z)) - a)^2}{2}$$

where

- G : Generator
- D : Disrciminator
- a : Target discriminator label for generated image

- b : Target discriminator label for real image

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **a** (*float, optional*) – Target discriminator label for generated image.
- **b** (*float, optional*) – Target discriminator label for real image.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*dx*, *dgz*)

Computes the loss for the given input.

Parameters

- **dx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.3 Minimax Loss

9.3.1 MinimaxGeneratorLoss

class torchgan.losses.**MinimaxGeneratorLoss**(*reduction='mean'*, *nonsaturating=True*, *override_train_ops=None*)

Minimax game generator loss from the original GAN paper "Generative Adversarial Networks by Goodfellow et. al."

The loss can be described as:

$$L(G) = log(1 - D(G(z)))$$

The nonsaturating heuristic is also supported:

$$L(G) = -log(D(G(z)))$$

where

- G : Generator
- D : Discriminator
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.

- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`
- **nonsaturating** (*bool, optional*) – Specifies whether to use the nonsaturating heuristic loss for the generator.

forward(*dgz*)

Computes the loss for the given input.

Parameters **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.3.2 MinimaxDiscriminatorLoss

class torchgan.losses.**MinimaxDiscriminatorLoss**(*label_smoothing=0.0*, *reduction='mean'*, *override_train_ops=None*)

Minimax game discriminator loss from the original GAN paper "Generative Adversarial Networks by Goodfellow et. al."

The loss can be described as:

$$L(D) = -[log(D(x)) + log(1 - D(G(z)))]$$

where

- G : Generator
- D : Discriminator
- x : A sample from the data distribution
- z : A sample from the noise prior

Parameters

- **label_smoothing** (*float, optional*) – The factor by which the labels (1 in this case) needs to be smoothened. For example, label_smoothing = 0.2 changes the value of the real labels to 0.8.
- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the mean of the output. If `sum` the elements of the output will be summed.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward(*dx*, *dgz*)

Computes the loss for the given input.

Parameters

- **dx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.4 Boundary Equilibrium Loss

9.4.1 BoundaryEquilibriumGeneratorLoss

class torchgan.losses.**BoundaryEquilibriumGeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)

Boundary Equilibrium GAN generator loss from "BEGAN : Boundary Equilibrium Generative Adversarial Networks by Berthelot et. al." paper

The loss can be described as

$$L(G) = D(G(z))$$

where

- G : Generator
- D : Discriminator

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*dgz*)

Computes the loss for the given input.

Parameters **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.4.2 BoundaryEquilibriumDiscriminatorLoss

class torchgan.losses.**BoundaryEquilibriumDiscriminatorLoss**(*reduction='mean'*, *override_train_ops=None*, *init_k=0.0*, *lambd=0.001*, *gamma=0.75*)

Boundary Equilibrium GAN discriminator loss from "BEGAN : Boundary Equilibrium Generative Adversarial Networks by Berthelot et. al." paper

The loss can be described as

$$L(D) = D(x) - k_t \times D(G(z))$$

$$k_{t+1} = k_t + \lambda \times (\gamma \times D(x) - D(G(z)))$$

where

- G : Generator
- D : Discriminator

- k_t : Running average of the balance point of G and D
- λ : Learning rate of the running average
- γ : Goal bias hyperparameter

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place ofthe default `train_ops`
- **init_k** (*float, optional*) – Initial value of the balance point `k`.
- **lambd** (*float, optional*) – Learning rate of the running average.
- **gamma** (*float, optional*) – Goal bias hyperparameter.

forward (*dx*, *dgz*)

Computes the loss for the given input.

Parameters

- **dx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns A tuple of 3 loss values, namely the `total loss`, `loss due to real data` and `loss due to fake data`.

set_k (*k=0.0*)

Change the default value of k

Parameters **k** (*float, optional*) – New value to be set.

train_ops (*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)

Defines the standard `train_ops` used by boundary equilibrium loss.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value_1 = discriminator(fake)$
3. $value_2 = discriminator(real)$
4. $loss = loss\_function(value_1, value_2)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator
7. Update the value of :math: *k*.

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.

- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

update_k(*loss_real*, *loss_fake*)

Update the running mean of k for each forward pass.

The update takes place as

$$k_{t+1} = k_t + \lambda \times (\gamma \times D(x) - D(G(z)))$$

Parameters

- **loss_real** (*float*) – $D(x)$
- **loss_fake** (*float*) – $D(G(z))$

9.5 Energy Based Loss

9.5.1 EnergyBasedGeneratorLoss

class torchgan.losses.**EnergyBasedGeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)

Energy Based GAN generator loss from "Energy Based Generative Adversarial Network by Zhao et. al." paper.

The loss can be described as:

$$L(G) = D(G(z))$$

where

- G : Generator
- D : Discriminator
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward(*dgz*)

Computes the loss for the given input.

Parameters **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_generator*, *device*, *batch_size*, *labels=None*)
This function sets the `embeddings` attribute of the `AutoEncodingDiscriminator` to `False` and calls the `train_ops` of the `GeneratorLoss`. After the call the attribute is again set to `True`.

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `generator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.5.2 EnergyBasedDiscriminatorLoss

class torchgan.losses.**EnergyBasedDiscriminatorLoss**(*reduction='mean'*, *margin=80.0*, *override_train_ops=None*)
Energy Based GAN generator loss from "Energy Based Generative Adversarial Network by Zhao et. al." paper

The loss can be described as:

$$L(D) = D(x) + max(0, m - D(G(z)))$$

where

- G : Generator
- D : Discriminator
- m : Margin Hyperparameter
- z : A sample from the noise prior

Note: The convergence of EBGAN is highly sensitive to hyperparameters. The `margin` hyperparameter as per the paper was taken as follows:

Dataset	Margin
MNIST	10.0
LSUN	80.0
CELEB A	20.0
Imagenet (128 x 128)	40.0
Imagenet (256 x 256)	80.0

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **margin** (*float, optional*) – The margin hyperparameter.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*dx*, *dgz*)

Computes the loss for the given input.

Parameters

- **dx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *batch_size*, *labels=None*)

This function sets the `embeddings` attribute of the `AutoEncodingDiscriminator` to `False` and calls the `train_ops` of the `DiscriminatorLoss`. After the call the attribute is again set to `True`.

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.5.3 EnergyBasedPullingAwayTerm

class torchgan.losses.**EnergyBasedPullingAwayTerm**(*pt_ratio=0.1*, *override_train_ops=None*)

Energy Based Pulling Away Term from "Energy Based Generative Adversarial Network by Zhao et. al." paper.

The loss can be described as:

$$f_{PT}(S) = \frac{1}{N(N-1)} \sum_{i} \sum_{j \neq i} \left(\frac{S_i^T S_j}{||S_i||\,||S_j||} \right)^2$$

where

- S : The feature output from the encoder for generated images

- N : Batch Size of the Input

Parameters

- **pt_ratio** (*float, optional*) – The weight given to the pulling away term.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward (*dgz*, *d_hid*)
Computes the loss for the given input.

Parameters

- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **d_hid** (*torch.Tensor*) – The embeddings generated by the discriminator.

Returns scalar.

train_ops (*generator*, *discriminator*, *optimizer_generator*, *device*, *batch_size*, *labels=None*)
This function extracts the hidden embeddings of the discriminator network. The furthur computation is same as the standard train_ops.

Note: For the loss to work properly, the discriminator must be a `AutoEncodingDiscriminator` and it must have a `embeddings` attribute which should be set to `True`. Also the generator `label_type` must be `none`. As a result of these constraints it advisable not to use custom models with this loss. This will be improved in future.

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `generator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.6 Wasserstein Loss

9.6.1 WassersteinGeneratorLoss

class torchgan.losses.**WassersteinGeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)
Wasserstein GAN generator loss from "Wasserstein GAN by Arjovsky et. al." paper

The loss can be described as:

$$L(G) = -f(G(z))$$

where

- G : Generator
- f : Critic/Discriminator
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the mean of the output. If `sum` the elements of the output will be summed.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward (*fgz*)

Computes the loss for the given input.

Parameters **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

9.6.2 WassersteinDiscriminatorLoss

class torchgan.losses.**WassersteinDiscriminatorLoss**(*reduction='mean', clip=None, override_train_ops=None*)

Wasserstein GAN generator loss from "Wasserstein GAN by Arjovsky et. al." paper

The loss can be described as:

$$L(D) = f(G(z)) - f(x)$$

where

- G : Generator
- f : Critic/Discriminator
- x : A sample from the data distribution
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the mean of the output. If `sum` the elements of the output will be summed.
- **clip** (*tuple, optional*) – Tuple that specifies the maximum and minimum parameter clamping to be applied, as per the original version of the Wasserstein loss without Gradient Penalty.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward (*fx*, *fgz*)
Computes the loss for the given input.

Parameters

- **fx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **fgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops (*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)
Defines the standard `train_ops` used by wasserstein discriminator loss.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. Clamp the discriminator parameters to satisfy $lipschitz\ condition$
2. $fake = generator(noise)$
3. $value_1 = discriminator(fake)$
4. $value_2 = discriminator(real)$
5. $loss = loss\_function(value_1, value_2)$
6. Backpropagate by computing $\nabla loss$
7. Run a step of the optimizer for discriminator

Parameters

- **generator** (*torchgan.models.Generator*) – The model to be optimized.
- **discriminator** (*torchgan.models.Discriminator*) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.6.3 WassersteinGradientPenalty

class torchgan.losses.**WassersteinGradientPenalty** (*reduction='mean'*, *lambd=10.0*, *override_train_ops=None*)
Gradient Penalty for the Improved Wasserstein GAN discriminator from "Improved Training of Wasserstein GANs by Gulrajani et. al." paper

The gradient penalty is calculated as:

The gradient being taken with respect to x

where

- G : Generator
- D : Disrciminator/Critic
- λ : Scaling hyperparameter
- x : Interpolation term for the gradient penalty

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the mean of the output. If `sum` the elements of the output will be summed.
- **lambd** (*float,optional*) – Hyperparameter lambda for scaling the gradient penalty.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward(*interpolate*, *d_interpolate*)
Computes the loss for the given input.

Parameters

- **interpolate** (*torch.Tensor*) – It must have the dimensions (N, *) where * means any number of additional dimensions.
- **d_interpolate** (*torch.Tensor*) – Output of the `discriminator` with `interpolate` as the input. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)
Defines the standard `train_ops` used by the Wasserstein Gradient Penalty.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $interpolate = \epsilon \times real + (1 - \epsilon) \times fake$
3. $d\_interpolate = discriminator(interpolate)$
4. $loss = \lambda loss\_function(interpolate, d\_interpolate)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized.
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.

- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.7 Mutual Information Penalty

class torchgan.losses.**MutualInformationPenalty**(*lambd=1.0*, *reduction='mean'*, *override_train_ops=None*)

Mutual Information Penalty as defined in "InfoGAN : Interpretable Representation Learning by Information Maximising Generative Adversarial Nets by Chen et. al." paper

The loss is the variational lower bound of the mutual information between the latent codes and the generator distribution and is defined as

$$L(G, Q) = log(Q|x)$$

where

- x is drawn from the generator distribution G(z,c)
- c drawn from the latent code prior $P(c)$

Parameters

- **lambd** (*float, optional*) – The scaling factor for the loss.
- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the mean of the output. If `sum` the elements of the output will be summed.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

forward(*c_dis*, *c_cont*, *dist_dis*, *dist_cont*)

Computes the loss for the given input.

Parameters

- **c_dis** (*int*) – The discrete latent code sampled from the prior.
- **c_cont** (*int*) – The continuous latent code sampled from the prior.
- **dist_dis** (*torch.distributions.Distribution*) – The auxilliary distribution $Q(c|x)$ over the discrete latent code output by the discriminator.
- **dist_cont** (*torch.distributions.Distribution*) – The auxilliary distribution $Q(c|x)$ over the continuous latent code output by the discriminator.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_generator*, *optimizer_discriminator*, *dis_code*, *cont_code*, *device*, *batch_size*)

Defines the standard `train_ops` used by most losses. Losses which have a different training procedure can either `subclass` it **(recommended approach)** or make use of `override_train_ops` argument.

The `standard optimization algorithm` for the `generator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value = discriminator(fake)$
3. $loss = loss\_function(value)$
4. Backpropagate by computing $\nabla loss$
5. Run a step of the optimizer for generator

Parameters

- **generator** (torchgan.models.Generator) – The model to be optimized.
- **discriminator** (torchgan.models.Discriminator) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the parameters of the generator.
- **device** (*torch.device*) – Device on which the generator and discriminator is present.
- **batch_size** (*int*) – Batch Size of the data infered from the DataLoader by the Trainer.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.8 Dragan Loss

9.8.1 DraganGradientPenalty

class torchgan.losses.**DraganGradientPenalty**(*reduction='mean', lambd=10.0, k=1.0, override_train_ops=None*)

Gradient Penalty for the DRAGAN discriminator from "On Convergence and Stability of GANs by Kodali et. al." paper

The gradient penalty is calculated as:

$$\lambda \times (||grad(D(x))||_2 - k)^2$$

The gradient being taken with respect to x

where

- G : Generator
- D : Disrciminator
- λ : Scaling hyperparameter
- x : Interpolation term for the gradient penalty
- k : Constant

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If none no reduction will be applied. If mean the outputs are averaged over batch size. If sum the elements of the output are summed.

- **lambd** (*float, optional*) – Hyperparameter λ for scaling the gradient penalty.
- **k** (*float, optional*) – Constant.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*interpolate*, *d_interpolate*)

Computes the loss for the given input.

Parameters

- **interpolate** (*torch.Tensor*) – It must have the dimensions (N, *) where * means any number of additional dimensions.
- **d_interpolate** (*torch.Tensor*) – Output of the `discriminator` with `interpolate` as the input. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)

Defines the standard `train_ops` used by the DRAGAN Gradient Penalty.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. $interpolate = real + \frac{1}{2} \times (1 - \alpha) \times std(real) \times \beta$
2. $d\_interpolate = discriminator(interpolate)$
3. $loss = loss\_function(interpolate, d\_interpolate)$
4. Backpropagate by computing $\nabla loss$
5. Run a step of the optimizer for discriminator

Parameters

- **generator** (*torchgan.models.Generator*) – The model to be optimized.
- **discriminator** (*torchgan.models.Discriminator*) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.9 Auxillary Classifier Loss

9.9.1 AuxiliaryClassifierGeneratorLoss

class torchgan.losses.**AuxiliaryClassifierGeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)

Auxiliary Classifier GAN (ACGAN) loss based on a from "Conditional Image Synthesis With Auxiliary Classifier GANs by Odena et. al. " paper

Parameters **reduction** (*str, optional*) –

Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.

override_train_ops (function, optional): A function is passed to this argument, if the default `train_ops` is not to be used.

train_ops(*generator*, *discriminator*, *optimizer_generator*, *device*, *batch_size*, *labels=None*)

Defines the standard `train_ops` used by the Auxiliary Classifier generator loss.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows (label_g and label_d both could be either real labels or generated labels):

1. $fake = generator(noise, label_g)$
2. $value_1 = classifier(fake, label_g)$
3. $value_2 = classifier(real, label_d)$
4. $loss = loss\_function(value_1, label_g) + loss\_function(value_2, label_d)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized. For ACGAN, it must require labels for training
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.9.2 AuxiliaryClassifierDiscriminatorLoss

class torchgan.losses.**AuxiliaryClassifierDiscriminatorLoss**(*reduction='mean'*, *override_train_ops=None*)

Auxiliary Classifier GAN (ACGAN) loss based on a from "Conditional Image Synthesis With Auxiliary Classifier GANs by Odena et. al. " paper

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – A function is passed to this argument, if the default `train_ops` is not to be used.

train_ops(*generator*, *discriminator*, *optimizer_discriminator*, *real_inputs*, *device*, *labels=None*)

Defines the standard `train_ops` used by the Auxiliary Classifier discriminator loss.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows (label_g and label_d both could be either real labels or generated labels):

1. $fake = generator(noise, label_g)$
2. $value_1 = classifier(fake, label_g)$
3. $value_2 = classifier(real, label_d)$
4. $loss = loss\_function(value_1, label_g) + loss\_function(value_2, label_d)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator

Parameters

- **generator** (`torchgan.models.Generator`) – The model to be optimized. For ACGAN, it must require labels for training
- **discriminator** (`torchgan.models.Discriminator`) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.10 Feature Matching Loss

9.10.1 FeatureMatchingGeneratorLoss

class torchgan.losses.**FeatureMatchingGeneratorLoss**(*reduction='mean'*, *override_train_ops=None*)

Feature Matching Generator loss from "Improved Training of GANs by Salimans et. al." paper

The loss can be described as:

$$L(G) = ||f(x) - f(G(z))||_2$$

where

- G : Generator
- f : An intermediate activation from the discriminator
- z : A sample from the noise prior

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If `none` no reduction will be applied. If `mean` the outputs are averaged over batch size. If `sum` the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default `train_ops`

forward(*fx*, *fgz*)

Computes the loss for the given input.

Parameters

- **dx** (*torch.Tensor*) – Output of the Discriminator with real data. It must have the dimensions (N, *) where * means any number of additional dimensions.
- **dgz** (*torch.Tensor*) – Output of the Discriminator with generated data. It must have the dimensions (N, *) where * means any number of additional dimensions.

Returns scalar if reduction is applied else Tensor with dimensions (N, *).

train_ops(*generator*, *discriminator*, *optimizer_generator*, *real_inputs*, *device*, *labels=None*)

Defines the standard `train_ops` used for feature matching.

The `standard optimization algorithm` for the `generator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value_1 = discriminator(fake)$ **where** $value_1$ **is an activation of an intermediate** discriminator layer
3. $value_2 = discriminator(real)$ **where** $value_2$ **is an activation of the same intermediate** discriminator layer
4. $loss = loss\_function(value_1, value_2)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for generator

Parameters

- **generator** (torchgan.models.Generator) – The model to be optimized.
- **discriminator** (torchgan.models.Discriminator) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the parameters of the generator.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the discriminator.
- **device** (*torch.device*) – Device on which the generator and discriminator is present.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.11 Historical Averaging

9.11.1 HistoricalAverageGeneratorLoss

class torchgan.losses.**HistoricalAverageGeneratorLoss**(*reduction='elementwise_mean'*, *override_train_ops=None*, *lambd=1.0*)

Historical Average Generator Loss from "Improved Techniques for Training GANs by Salimans et. al." paper

The loss can be described as

$$|| - \frac{1}{t}\sum_{i=1}^{t}[i]||^2$$

where

- G : Generator
- **math** *vtheta[i]* : Generator Parameters at Past Timestep :math: i

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If none no reduction will be applied. If mean the outputs are averaged over batch size. If sum the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default train_ops
- **lambd** (*float, optional*) – Hyperparameter lambda for scaling the Historical Average Penalty

train_ops(*generator*, *optimizer_generator*)

Defines the standard train_ops used by most losses. Losses which have a different training procedure can either subclass it **(recommended approach)** or make use of override_train_ops argument.

The standard optimization algorithm for the generator defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value = discriminator(fake)$

3. $loss = loss\_function(value)$
4. Backpropagate by computing $\nabla loss$
5. Run a step of the optimizer for generator

Parameters

- **generator** (torchgan.models.Generator) – The model to be optimized.
- **discriminator** (torchgan.models.Discriminator) – The discriminator which judges the performance of the generator.
- **optimizer_generator** (*torch.optim.Optimizer*) – Optimizer which updates the parameters of the generator.
- **device** (*torch.device*) – Device on which the generator and discriminator is present.
- **batch_size** (*int*) – Batch Size of the data infered from the DataLoader by the Trainer.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

9.11.2 HistoricalAverageDiscriminatorLoss

class torchgan.losses.**HistoricalAverageDiscriminatorLoss**(*reduction='elementwise_mean'*, *override_train_ops=None*, *lambd=1.0*)

Historical Average Discriminator Loss from "Improved Techniques for Training GANs by Salimans et. al." paper

The loss can be described as

$$|| - \frac{1}{t}\sum_{i=1}^{t}[i]||^2$$

where

- G : Discriminator
- **math** *vtheta[i]* : Discriminator Parameters at Past Timestep :math: i

Parameters

- **reduction** (*str, optional*) – Specifies the reduction to apply to the output. If none no reduction will be applied. If mean the outputs are averaged over batch size. If sum the elements of the output are summed.
- **override_train_ops** (*function, optional*) – Function to be used in place of the default train_ops
- **lambd** (*float, optional*) – Hyperparameter lambda for scaling the Historical Average Penalty

train_ops(*discriminator*, *optimizer_discriminator*)

Defines the standard train_ops used by most losses. Losses which have a different training procedure can either subclass it **(recommended approach)** or make use of override_train_ops argument.

The `standard optimization algorithm` for the `discriminator` defined in this train_ops is as follows:

1. $fake = generator(noise)$
2. $value_1 = discriminator(fake)$
3. $value_2 = discriminator(real)$
4. $loss = loss\_function(value_1, value_2)$
5. Backpropagate by computing $\nabla loss$
6. Run a step of the optimizer for discriminator

Parameters

- **generator** (*torchgan.models.Generator*) – The model to be optimized.
- **discriminator** (*torchgan.models.Discriminator*) – The discriminator which judges the performance of the generator.
- **optimizer_discriminator** (*torch.optim.Optimizer*) – Optimizer which updates the `parameters` of the `discriminator`.
- **real_inputs** (*torch.Tensor*) – The real data to be fed to the `discriminator`.
- **device** (*torch.device*) – Device on which the `generator` and `discriminator` is present.
- **batch_size** (*int*) – Batch Size of the data infered from the `DataLoader` by the `Trainer`.
- **labels** (*torch.Tensor, optional*) – Labels for the data.

Returns Scalar value of the loss.

CHAPTER 10

torchgan.metrics

This subpackage provides various metrics that are available to judge the performance of GANs. Currently available metrics are:

10.1 Metric

10.1.1 EvaluationMetric

class torchgan.metrics.**EvaluationMetric**

Base class for all Evaluation Metrics

calculate_score(*x*)

Subclasses must override this function and provide their own score calculation.

Raises **NotImplementedError** – If the subclass doesn't override this function.

metric_ops(*generator*, *discriminator*, ***kwargs*)

Subclasses must override this function and provide their own metric evaluation ops.

Raises **NotImplementedError** – If the subclass doesn't override this function.

preprocess(*x*)

Subclasses must override this function and provide their own preprocessing pipeline.

Raises **NotImplementedError** – If the subclass doesn't override this function.

set_arg_map(*value*)

Updates the `arg_map` for passing a different value to the `metric_ops`.

Parameters **value** (*dict*) – A mapping of the `argument name` in the method signature and the variable name in the `Trainer` it corresponds to.

Note: If the `metric_ops` signature is `metric_ops(self, gen, disc)` then we need to map `gen` to `generator` and `disc` to `discriminator`. In this case we make the following function call `metric.set_arg_map({"gen": "generator", "disc": "discriminator"})`.

10.2 Classifier Score

class torchgan.metrics.**ClassifierScore**(*classifier=None*, *transform=None*, *sample_size=1*)

Computes the Classifier Score of a Model. Also popularly known as the Inception Score. The `classifier` can be any model. It also supports models outside of torchvision models. For more details on how to use custom trained models look up the tutorials.

Parameters

- **classifier** (*torch.nn.Module, optional*) – The model to be used as a base to compute the classifier score. If `None` is passed the pretrained `torchvision.models.inception_v3` is used.

  Note: Ensure that the classifier is on the same `device` as the Trainer to avoid sudden crash.

- **transform** (*torchvision.transforms, optional*) – Transformations applied to the image before feeding it to the classifier. Look up the documentation of the torchvision models for this transforms.
- **sample_size** (*int*) – Batch Size for calculation of Classifier Score.

calculate_score(*x*)

Computes the Inception Score for the Input.

Parameters **x** (*torch.Tensor*) – Image in tensor format

Returns The Inception Score.

metric_ops(*generator*, *device*)

Defines the set of operations necessary to compute the ClassifierScore.

Parameters

- **generator** (`torchgan.models.Generator`) – The generator which needs to be evaluated.
- **device** (*torch.device*) – Device on which the generator is present.

Returns The Classifier Score (scalar quantity)

preprocess(*x*)

Preprocessor for the Classifier Score. It transforms the image as per the transform requirements and feeds it to the classifier.

Parameters **x** (*torch.Tensor*) – Image in tensor format

Returns The output from the classifier.

CHAPTER 11

torchgan.models

This models subpackage is a collection of popular GAN architectures. It has the support for existing architectures and provides a base class for extending to any form of new architecture. Currently the following models are supported:

You can construct a new model by simply calling its constructor.

```
>>> import torchgan.models as models
>>> dcgan_discriminator = DCGANDiscriminator()
>>> dcgan_generator = DCGANGenerator()
```

All models follow the same structure. There are additional customization options. Look into the individual documentation for such capabilities.

11.1 Vanilla GAN

11.1.1 Generator

class torchgan.models.**Generator**(*encoding_dims*, *label_type='none'*)

Base class for all Generator models. All Generator models must subclass this.

Parameters

- **encoding_dims** (*int*) – Dimensions of the sample from the noise prior.
- **label_type** (*str, optional*) – The type of labels expected by the Generator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

_weight_initializer()

Default weight initializer for all generator models. Models that require custom weight initialization can override this method

sampler(*sample_size*, *device*)

Function to allow sampling data at inference time. Models requiring input in any other format must override it in the subclass.

Parameters

- **sample_size** (*int*) – The number of images to be generated
- **device** (*torch.device*) – The device on which the data must be generated

Returns A list of the items required as input

11.1.2 Discriminator

class torchgan.models.**Discriminator**(*input_dims*, *label_type='none'*)

Base class for all Discriminator models. All Discriminator models must subclass this.

Parameters

- **input_dims** (*int*) – Dimensions of the input.
- **label_type** (*str, optional*) – The type of labels expected by the Discriminator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

_weight_initializer()

Default weight initializer for all disciminator models. Models that require custom weight initialization can override this method

11.2 Deep Convolutional GAN

11.2.1 DCGANGenerator

class torchgan.models.**DCGANGenerator**(*encoding_dims=100*, *out_size=32*, *out_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*, *label_type='none'*)

Deep Convolutional GAN (DCGAN) generator from "Unsupervised Representation Learning With Deep Convolutional Generative Aversarial Networks by Radford et. al. " paper

Parameters

- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **out_size** (*int, optional*) – Height and width of the input image to be generated. Must be at least 16 and should be an exact power of 2.
- **out_channels** (*int, optional*) – Number of channels in the output Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.
- **label_type** (*str, optional*) – The type of labels expected by the Generator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

forward(*x*, *feature_matching=False*)

Calculates the output tensor on passing the encoding `x` through the Generator.

Parameters

- **x** (*torch.Tensor*) – A 2D torch tensor of the encoding sampled from a probability distribution.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 4D torch.Tensor of the generated image.

11.2.2 DCGANDiscriminator

class torchgan.models.**DCGANDiscriminator**(*in_size=32*, *in_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*, *label_type='none'*)

Deep Convolutional GAN (DCGAN) discriminator from "Unsupervised Representation Learning With Deep Convolutional Generative Aversarial Networks by Radford et. al. " paper

Parameters

- **in_size** (*int, optional*) – Height and width of the input image to be evaluated. Must be at least 16 and should be an exact power of 2.
- **in_channels** (*int, optional*) – Number of channels in the input Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.
- **label_type** (*str, optional*) – The type of labels expected by the Generator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

forward(*x*, *feature_matching=False*)

Calculates the output tensor on passing the image `x` through the Discriminator.

Parameters

- **x** (*torch.Tensor*) – A 4D torch tensor of the image.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 1D torch.Tensor of the probability of each image being real.

11.3 Conditional GAN

11.3.1 ConditionalGANGenerator

class torchgan.models.**ConditionalGANGenerator**(*num_classes*, *encoding_dims=100*, *out_size=32*, *out_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*)

Conditional GAN (CGAN) generator based on a DCGAN model from "Conditional Generative Adversarial Nets by Mirza et. al. " paper

Parameters

- **num_classes** (*int*) – Total classes present in the dataset.
- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **out_size** (*int, optional*) – Height and width of the input image to be generated. Must be at least 16 and should be an exact power of 2.
- **out_channels** (*int, optional*) – Number of channels in the output Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.

- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

forward(*z*, *y*)

Calculates the output tensor on passing the encoding `z` through the Generator.

Parameters

- **z** (*torch.Tensor*) – A 2D torch tensor of the encoding sampled from a probability distribution.
- **y** (*torch.Tensor*) – The labels corresponding to the encoding `z`.

Returns A 4D torch.Tensor of the generated Images conditioned on `y`.

sampler(*sample_size*, *device*)

Function to allow sampling data at inference time. Models requiring input in any other format must override it in the subclass.

Parameters

- **sample_size** (*int*) – The number of images to be generated
- **device** (*torch.device*) – The device on which the data must be generated

Returns A list of the items required as input

11.3.2 ConditionalGANDiscriminator

class torchgan.models.**ConditionalGANDiscriminator**(*num_classes*, *in_size=32*, *in_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*)

Condititional GAN (CGAN) discriminator based on a DCGAN model from "Conditional Generative Adversarial Nets by Mirza et. al. " paper

Parameters

- **num_classes** (*int*) – Total classes present in the dataset.
- **in_size** (*int, optional*) – Height and width of the input image to be evaluated. Must be at least 16 and should be an exact power of 2.
- **in_channels** (*int, optional*) – Number of channels in the input Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

forward(*x*, *y*, *feature_matching=False*)
Calculates the output tensor on passing the image `x` through the Discriminator.

Parameters

- **x** (*torch.Tensor*) – A 4D torch tensor of the image.
- **y** (*torch.Tensor*) – Labels corresponding to the images `x`.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 1D torch.Tensor of the probability of each image being real.

11.4 InfoGAN

11.4.1 InfoGANGenerator

class torchgan.models.**InfoGANGenerator**(*dim_dis*, *dim_cont*, *encoding_dims=100*, *out_size=32*, *out_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*)
Generator for InfoGAN based on the Deep Convolutional GAN (DCGAN) architecture, from "InfoGAN : Interpretable Representation Learning With Information Maximizing Generative Aversarial Nets by Chen et. al. " paper

Parameters

- **dim_dis** (*int*) – Dimension of the discrete latent code sampled from the prior.
- **dim_cont** (*int*) – Dimension of the continuous latent code sampled from the prior.
- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **out_size** (*int, optional*) – Height and width of the input image to be generated. Must be at least 16 and should be an exact power of 2.
- **out_channels** (*int, optional*) – Number of channels in the output Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

Example

```
>>> import torchgan.models as models
>>> G = models.InfoGANGenerator(10, 30)
>>> z = torch.randn(10, 100)
```

(continues on next page)

(continued from previous page)

```
>>> c_cont = torch.randn(10, 10)
>>> c_dis = torch.randn(10, 30)
>>> x = G(z, c_cont, c_dis)
```

forward(*z*, *c_dis=None*, *c_cont=None*)

Calculates the output tensor on passing the encoding `x` through the Generator.

Parameters

- **x** (*torch.Tensor*) – A 2D torch tensor of the encoding sampled from a probability distribution.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 4D torch.Tensor of the generated image.

11.4.2 InfoGANDiscriminator

class torchgan.models.**InfoGANDiscriminator**(*dim_dis*, *dim_cont*, *in_size=32*, *in_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*, *latent_nonlinearity=None*)

Discriminator for InfoGAN based on the Deep Convolutional GAN (DCGAN) architecture, from "InfoGAN : Interpretable Representation Learning With Information Maximizing Generative Aversarial Nets by Chen et. al. " paper

The approximate conditional probability distribution over the latent code Q(c|x) is chosen to be a factored Gaussian for the continuous latent code and a Categorical distribution for the discrete latent code

Parameters

- **dim_dis** (*int*) – Dimension of the discrete latent code sampled from the prior.
- **dim_cont** (*int*) – Dimension of the continuous latent code sampled from the prior.
- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **in_size** (*int, optional*) – Height and width of the input image to be evaluated. Must be at least 16 and should be an exact power of 2.
- **in_channels** (*int, optional*) – Number of channels in the input Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.
- **latent_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the `dist_conv`. Defaults to `LeakyReLU(0.2)` when None is passed.

Example

```
>>> import torchgan.models as models
>>> D = models.InfoGANDiscriminator(10, 30)
>>> x = torch.randn(10, 3, 32, 32)
>>> score, q_categorical, q_gaussian = D(x, return_latents=True)
```

forward(*x*, *return_latents=False*, *feature_matching=False*)

Calculates the output tensor on passing the image `x` through the Discriminator.

Parameters

- **x** (*torch.Tensor*) – A 4D torch tensor of the image.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 1D torch.Tensor of the probability of each image being real.

11.5 AutoEncoders

11.5.1 AutoEncodingGenerator

class torchgan.models.**AutoEncodingGenerator**(*encoding_dims=100*, *out_size=32*, *out_channels=3*, *step_channels=64*, *scale_factor=2*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*, *label_type='none'*)

Autoencoding Generator for Boundary Equilibrium GAN (BEGAN) from "BEGAN : Boundary Equilibrium Generative Adversarial Networks by Berthelot et. al." paper

Parameters

- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **out_size** (*int, optional*) – Height and width of the input image to be generated. Must be at least 16 and should be an exact power of 2.
- **out_channels** (*int, optional*) – Number of channels in the output Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **scale_factor** (*int, optional*) – The scale factor is used to infer properties of the model like `upsample_pad`, `upsample_filters`, `upsample_stride` and `upsample_output_pad`.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

- **label_type** (*str, optional*) – The type of labels expected by the Generator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

forward(*z*)

Calculates the output tensor on passing the encoding `z` through the Generator.

Parameters **z** (*torch.Tensor*) – A 2D torch tensor of the encoding sampled from a probability distribution.

Returns A 4D torch.Tensor of the generated image.

11.5.2 AutoEncodingDiscriminator

class torchgan.models.**AutoEncodingDiscriminator**(*in_size=32*, *in_channels=3*, *encoding_dims=100*, *step_channels=64*, *scale_factor=2*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*, *energy=True*, *embeddings=False*, *label_type='none'*)

Autoencoding Generator for Boundary Equilibrium GAN (BEGAN) from "BEGAN : Boundary Equilibrium Generative Adversarial Networks by Berthelot et. al." paper

Parameters

- **in_size** (*int, optional*) – Height and width of the input image to be evaluated. Must be at least 16 and should be an exact power of 2.
- **in_channels** (*int, optional*) – Number of channels in the input Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **scale_factor** (*int, optional*) – The scale factor is used to infer properties of the model like `downsample_pad`, `downsample_filters` and `downsample_stride`.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.
- **energy** (*bool, optional*) – If set to True returns the energy instead of the decoder output.
- **embeddings** (*bool, optional*) – If set to True the embeddings will be returned.
- **label_type** (*str, optional*) – The type of labels expected by the Generator. The available choices are 'none' if no label is needed, 'required' if the original labels are needed and 'generated' if labels are to be sampled from a distribution.

forward(*x*, *feature_matching=False*)

Calculates the output tensor on passing the image `x` through the Discriminator.

Parameters

- **x** (*torch.Tensor*) – A 4D torch tensor of the image.

- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 1D torch.Tensor of the energy value of each image.

11.6 Auxiliary Classifier GAN

11.6.1 ACGANGenerator

class torchgan.models.**ACGANGenerator**(*num_classes*, *encoding_dims=100*, *out_size=32*, *out_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*)

Auxiliary Classifier GAN (ACGAN) generator based on a DCGAN model from "Conditional Image Synthesis With Auxiliary Classifier GANs by Odena et. al. " paper

Parameters

- **num_classes** (*int*) – Total classes present in the dataset.
- **encoding_dims** (*int, optional*) – Dimension of the encoding vector sampled from the noise prior.
- **out_size** (*int, optional*) – Height and width of the input image to be generated. Must be at least 16 and should be an exact power of 2.
- **out_channels** (*int, optional*) – Number of channels in the output Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

forward(*z*, *y*)

Calculates the output tensor on passing the encoding `z` through the Generator.

Parameters

- **z** (*torch.Tensor*) – A 2D torch tensor of the encoding sampled from a probability distribution.
- **y** (*torch.Tensor*) – The labels corresponding to the encoding `z`.

Returns A 4D torch.Tensor of the generated Images conditioned on `y`.

sampler(*sample_size*, *device*)

Function to allow sampling data at inference time. Models requiring input in any other format must override it in the subclass.

Parameters

- **sample_size** (*int*) – The number of images to be generated
- **device** (*torch.device*) – The device on which the data must be generated

Returns A list of the items required as input

11.6.2 ACGANDiscriminator

class torchgan.models.**ACGANDiscriminator**(*num_classes*, *in_size=32*, *in_channels=3*, *step_channels=64*, *batchnorm=True*, *nonlinearity=None*, *last_nonlinearity=None*)

Auxiliary Classifier GAN (ACGAN) discriminator based on a DCGAN model from "Conditional Image Synthesis With Auxiliary Classifier GANs by Odena et. al. " paper

Parameters

- **num_classes** (*int*) – Total classes present in the dataset.
- **in_size** (*int, optional*) – Height and width of the input image to be evaluated. Must be at least 16 and should be an exact power of 2.
- **in_channels** (*int, optional*) – Number of channels in the input Tensor.
- **step_channels** (*int, optional*) – Number of channels in multiples of which the DCGAN steps up the convolutional features. The step up is done as dim $z \to d \to 2 \times d \to 4 \times d \to 8 \times d$ where d = step_channels.
- **batchnorm** (*bool, optional*) – If True, use batch normalization in the convolutional layers of the generator.
- **nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the intermediate convolutional layers. Defaults to `LeakyReLU(0.2)` when None is passed.
- **last_nonlinearity** (*torch.nn.Module, optional*) – Nonlinearity to be used in the final convolutional layer. Defaults to `Tanh()` when None is passed.

forward(*x*, *mode='discriminator'*, *feature_matching=False*)

Calculates the output tensor on passing the image `x` through the Discriminator.

Parameters

- **x** (*torch.Tensor*) – A 4D torch tensor of the image.
- **mode** (*str, optional*) – Option to choose the mode of the ACGANDiscriminator. Setting it to 'discriminator' gives the probability of the image being fake/real, 'classifier' allows it to make a prediction about the class of the image and anything else leads to returning both the values.
- **feature_matching** (*bool, optional*) – Returns the activation from a predefined intermediate layer.

Returns A 1D torch.Tensor of the probability of each image being real.

CHAPTER 12

torchgan.trainer

This subpackage provides ability to perform end to end training capabilities of the Generator and Discriminator models. It provides strong visualization capabilities using tensorboardX. Most of the cases can be handled elegantly with the default trainer itself. But if incase you need to *subclass* the trainer for any reason follow the docs closely.

12.1 Base Trainer

class torchgan.trainer.**BaseTrainer**(*losses_list*, *metrics_list=None*, *device=<sphinx.ext.autodoc.importer._MockObject object>*, *ncritic=1*, *epochs=5*, *sample_size=8*, *checkpoints='./model/gan'*, *retain_checkpoints=5*, *recon='./images'*, *log_dir=None*, *test_noise=None*, *nrow=8*, ***kwargs*)

Base Trainer for TorchGANs.

Warning: This trainer is meant to form the base for all other Trainers. This is not meant for direct usage.

Features provided by this Base Trainer are:

- Loss and Metrics Logging via the `Logger` class.
- Generating Image Samples.
- Saving models at the end of every epoch and loading of previously saved models.
- Highly flexible and allows changing hyperparameters by simply adjusting the arguments.

Most of the functionalities provided by the Trainer are flexible enough and can be customized by simply passing different arguments. You can train anything from a simple DCGAN to complex CycleGANs without ever having to subclass this `Trainer`.

Parameters

- **losses_list** (*list*) – A list of the Loss Functions that need to be minimized. For a list of pre-defined losses look at `torchgan.losses`. All losses in the list must be a subclass of atleast `GeneratorLoss` or `DiscriminatorLoss`.
- **metrics_list** (*list, optional*) – List of Metric Functions that need to be logged. For a list of pre-defined metrics look at `torchgan.metrics`. All losses in the list must be a subclass of `EvaluationMetric`.
- **device** (*torch.device, optional*) – Device in which the operation is to be carried out. If you are using a CPU machine make sure that you change it for proper functioning.
- **ncritic** (*int, optional*) – Setting it to a value will make the discriminator train that many times more than the generator. If it is set to a negative value the generator will be trained that many times more than the discriminator.
- **sample_size** (*int, optional*) – Total number of images to be generated at the end of an epoch for logging purposes.
- **epochs** (*int, optional*) – Total number of epochs for which the models are to be trained.
- **checkpoints** (*str, optional*) – Path where the models are to be saved. The naming convention is if checkpoints is `./model/gan` then models are saved as `./model/gan0.model` and so on.
- **retain_checkpoints** (*int, optional*) – Total number of checkpoints that should be retained. For example, if the value is set to 3, we save at most 3 models and start rewriting the models after that.
- **recon** (*str, optional*) – Directory where the sampled images are saved. Make sure the directory exists from beforehand.
- **log_dir** (*str, optional*) – The directory for logging tensorboard. It is ignored if TENSORBOARD_LOGGING is 0.
- **test_noise** (*torch.Tensor, optional*) – If provided then it will be used as the noise for image sampling.
- **nrow** (*int, optional*) – Number of rows in which the image is to be stored.

Any other argument that you need to store in the object can be simply passed via keyword arguments.

complete(***kwargs*)

Marks the end of training. It saves the final model and turns off the logger.

Note: It is not necessary to call this function. If it is not called the logger is kept alive in the background. So it might be considered a good practice to call this function.

---

eval_ops(***kwargs*)

Runs all evaluation operations at the end of every epoch. It calls all the metric functions that are passed to the Trainer.

load_model(*load_path=''*, *load_items=None*)

Function to load the model and some necessary information along with it. List of items loaded:

- Epoch
- Model States
- Optimizer States
- Loss Information

- Loss Objects
- Metric Objects
- Loss Logs

Warning: An Exception is raised if the model could not be loaded. Make sure that the model being loaded was saved previously by `torchgan Trainer` itself. We currently do not support loading any other form of models but this might be improved in the future.

Parameters

- **load_path** (*str, optional*) – Path from which the model is to be loaded.
- **load_items** (*str, list, optional*) – Pass the variable name of any other item you want to load. If the item cannot be found then a warning will be thrown and model will start to train from scratch. So make sure that item was saved.

optim_ops()
Runs all the schedulers at the end of every epoch.

save_model(*epoch*, *save_items=None*)
Function saves the model and some necessary information along with it. List of items stored for future reference:

- Epoch
- Model States
- Optimizer States
- Loss Information
- Loss Objects
- Metric Objects
- Loss Logs

The save location is printed when this function is called.

Parameters

- **epoch** (*int, optional*) – Epoch Number at which the model is being saved
- **save_items** (*str, list, optional*) – Pass the variable name of any other item you want to save. The item must be present in the *__dict__* else training will come to an abrupt end.

train(*data_loader*, ***kwargs*)
Uses the information passed by the user while creating the object and trains the model. It iterates over the epochs and the DataLoader and calls the functions for training the models and logging the required variables.

Note: Even though `__call__` calls this function, it is best if `train` is not called directly. When `__call__` is invoked, we infer the `batch_size` from the `data_loader`. Also, we are certain not going to change the interface of the `__call__` function so it gives the user a stable API, while we can change the flow of execution of `train` in future.

> **Warning:** The user should never try to change this function in subclass. It is too delicate and changing affects every other function present in this `Trainer` class.

This function controls the execution of all the components of the `Trainer`. It controls the `logger`, `train_iter`, `save_model`, `eval_ops` and `optim_ops`.

Parameters **`data_loader`** (*torch.utils.data.DataLoader*) – A DataLoader for the trainer to iterate over and train the models.

`train_iter`()

Calls the train_ops of the loss functions. This is the core function of the Trainer. In most cases you will never have the need to extend this function. In extreme cases simply extend `train_iter_custom`.

> **Warning:** This function is needed in this exact state for the Trainer to work correctly. So it is highly recommended that this function is not changed even if the `Trainer` is subclassed.

Returns An NTuple of the `generator loss`, `discriminator loss`, `number of times the generator was trained` and the `number of times the discriminator was trained`.

`train_iter_custom`()

Function that needs to be extended if `train_iter` is to be modified. Use this function to perform any sort of initialization that need to be done at the beginning of any train iteration. Refer the model zoo and tutorials for more details on how to write this function.

12.2 Trainer

class torchgan.trainer.**Trainer**(*models*, *losses_list*, *metrics_list=None*, *device=<sphinx.ext.autodoc.importer._MockObject object>*, *ncritic=1*, *epochs=5*, *sample_size=8*, *checkpoints='./model/gan'*, *retain_checkpoints=5*, *recon='./images'*, *log_dir=None*, *test_noise=None*, *nrow=8*, ***kwargs*)

Standard Trainer for various GANs. This has been designed to work only on one GPU in case you are using a GPU.

Most of the functionalities provided by the Trainer are flexible enough and can be customized by simply passing different arguments. You can train anything from a simple DCGAN to complex CycleGANs without ever having to subclass this `Trainer`.

Parameters

- **`models`** (*dict*) – A dictionary containing a mapping between the variable name, storing the `generator`, `discriminator` and any other model that you might want to define, with the function and arguments that are needed to construct the model. Refer to the examples to see how to define complex models using this API.
- **`losses_list`** (*list*) – A list of the Loss Functions that need to be minimized. For a list of pre-defined losses look at `torchgan.losses`. All losses in the list must be a subclass of atleast `GeneratorLoss` or `DiscriminatorLoss`.
- **`metrics_list`** (*list, optional*) – List of Metric Functions that need to be logged. For a list of pre-defined metrics look at `torchgan.metrics`. All losses in the list must be a subclass of `EvaluationMetric`.

- **device** (*torch.device, optional*) – Device in which the operation is to be carried out. If you are using a CPU machine make sure that you change it for proper functioning.
- **ncritic** (*int, optional*) – Setting it to a value will make the discriminator train that many times more than the generator. If it is set to a negative value the generator will be trained that many times more than the discriminator.
- **sample_size** (*int, optional*) – Total number of images to be generated at the end of an epoch for logging purposes.
- **epochs** (*int, optional*) – Total number of epochs for which the models are to be trained.
- **checkpoints** (*str, optional*) – Path where the models are to be saved. The naming convention is if checkpoints is `./model/gan` then models are saved as `./model/gan0.model` and so on.
- **retain_checkpoints** (*int, optional*) – Total number of checkpoints that should be retained. For example, if the value is set to 3, we save at most 3 models and start rewriting the models after that.
- **recon** (*str, optional*) – Directory where the sampled images are saved. Make sure the directory exists from beforehand.
- **log_dir** (*str, optional*) – The directory for logging tensorboard. It is ignored if TENSORBOARD_LOGGING is 0.
- **test_noise** (*torch.Tensor, optional*) – If provided then it will be used as the noise for image sampling.
- **nrow** (*int, optional*) – Number of rows in which the image is to be stored.

Any other argument that you need to store in the object can be simply passed via keyword arguments.

Example

```
>>> dcgan = Trainer(
            {"generator": {"name": DCGANGenerator, "args": {"out_channels": 1,
↪"step_channels":
                           16}, "optimizer": {"name": Adam, "args": {"lr": 0.0002,
                           "betas": (0.5, 0.999)}}},
             "discriminator": {"name": DCGANDiscriminator, "args": {"in_channels
↪": 1,
                                "step_channels": 16}, "optimizer": {"var": "opt_
↪discriminator",
                                "name": Adam, "args": {"lr": 0.0002, "betas": (0.5,
↪ 0.999)}}}},
            [MinimaxGeneratorLoss(), MinimaxDiscriminatorLoss()],
            sample_size=64, epochs=20)
```

12.3 Parallel Trainer

class torchgan.trainer.**ParallelTrainer**(*models*, *losses_list*, *devices*, *metrics_list=None*, *ncritic=1*, *epochs=5*, *sample_size=8*, *checkpoints='./model/gan'*, *retain_checkpoints=5*, *recon='./images'*, *log_dir=None*, *test_noise=None*, *nrow=8*, ***kwargs*)

MultiGPU Trainer for GANs. Use the `Trainer` class for training on a single GPU or a CPU machine.

Parameters

- **models** (*dict*) – A dictionary containing a mapping between the variable name, storing the `generator`, `discriminator` and any other model that you might want to define, with the function and arguments that are needed to construct the model. Refer to the examples to see how to define complex models using this API.
- **losses_list** (*list*) – A list of the Loss Functions that need to be minimized. For a list of pre-defined losses look at `torchgan.losses`. All losses in the list must be a subclass of atleast `GeneratorLoss` or `DiscriminatorLoss`.
- **devices** (*list*) – Devices in which the operations are to be carried out. If you are using a CPU machine or a single GPU machine use the Trainer class.
- **metrics_list** (*list, optional*) – List of Metric Functions that need to be logged. For a list of pre-defined metrics look at `torchgan.metrics`. All losses in the list must be a subclass of `EvaluationMetric`.
- **ncritic** (*int, optional*) – Setting it to a value will make the discriminator train that many times more than the generator. If it is set to a negative value the generator will be trained that many times more than the discriminator.
- **sample_size** (*int, optional*) – Total number of images to be generated at the end of an epoch for logging purposes.
- **epochs** (*int, optional*) – Total number of epochs for which the models are to be trained.
- **checkpoints** (*str, optional*) – Path where the models are to be saved. The naming convention is if checkpoints is `./model/gan` then models are saved as `./model/gan0.model` and so on.
- **retain_checkpoints** (*int, optional*) – Total number of checkpoints that should be retained. For example, if the value is set to 3, we save at most 3 models and start rewriting the models after that.
- **recon** (*str, optional*) – Directory where the sampled images are saved. Make sure the directory exists from beforehand.
- **log_dir** (*str, optional*) – The directory for logging tensorboard. It is ignored if TENSORBOARD_LOGGING is 0.
- **test_noise** (*torch.Tensor, optional*) – If provided then it will be used as the noise for image sampling.
- **nrow** (*int, optional*) – Number of rows in which the image is to be stored.

Any other argument that you need to store in the object can be simply passed via keyword arguments.

Example

```
>>> dcgan = ParallelTrainer(
            {"generator": {"name": DCGANGenerator, "args": {"out_channels": 1,
↪"step_channels":
                           16}, "optimizer": {"name": Adam, "args": {"lr": 0.0002,
                           "betas": (0.5, 0.999)}}},
             "discriminator": {"name": DCGANDiscriminator, "args": {"in_channels
↪": 1,
                               "step_channels": 16}, "optimizer": {"var": "opt_
↪discriminator",
                               "name": Adam, "args": {"lr": 0.0002, "betas": (0.5,
↪ 0.999)}}}},
            [MinimaxGeneratorLoss(), MinimaxDiscriminatorLoss()],
            [0, 1, 2],
            sample_size=64, epochs=20)
```

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

O

P

R

S

T

U

V

W