Multi-view-AE

Release 0.1

Ana Lawry Aguila & Alejandra Jayme

Feb 16, 2024

USING MULTI-VIEW-AE

Multi-view-AE is a Python library for multi-view autoencoder models. Please see the user guides below.

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

Installation instructions for the `multi-view-AE` library.

Clone the below repository and move to folder:

```
git clone https://github.com/alawryaguila/multi-view-AE
cd multi-view-AE
```

Create the customised python environment:

```
conda create --name mvae python=3.9
```

Activate python environment:

```
conda activate mvae
```

Install the `multi-view-AE` package:

```
pip install ./
```

1.2 Models implemented

Below is a table with the models contained within this repository and links to the original papers.

[1] Antelmi, Luigi & Ayache, Nicholas & Robert, Philippe & Lorenzi, Marco. (2019). Sparse Multi-Channel Variational Autoencoder for the Joint Analysis of Heterogeneous Data.

[2] Sohn, K., Lee, H., & Yan, X. (2015). Learning Structured Output Representation using Deep Conditional Generative Models. NIPS.

[3] Wang, Weiran & Lee, Honglak & Livescu, Karen. (2016). Deep Variational Canonical Correlation Analysis.

[4] Yuge Shi, N. Siddharth, Brooks Paige, and Philip H. S. Torr. 2019. Variational mixture-of-experts autoencoders for multi-modal deep generative models. Proceedings of the 33rd International Conference on Neural Information Processing Systems. Curran Associates Inc., Red Hook, NY, USA, Article 1408, 15718–15729.

[5] Wu, Mike & Goodman, Noah. (2018). Multimodal Generative Models for Scalable Weakly-Supervised Learning.

[6] Suzuki, Masahiro and Nakayama, Kotaro and Matsuo, Yutaka. (2016). Joint Multimodal Learning with Deep Generative Models.

[7] Sutter, Thomas & Daunhawer, Imant & Vogt, Julia. (2021). Generalized Multimodal ELBO.

[8] Hwang, HyeongJoo and Kim, Geon-Hyeong and Hong, Seunghoon and Kim, Kee-Eung. Multi-View Representation Learning via Total Correlation Objective. 2021. NeurIPS

[9] Sutter, Thomas & Daunhawer, Imant & Vogt, Julia. (2021). Multimodal Generative Learning Utilizing Jensen-Shannon-Divergence. Advances in Neural Information Processing Systems. 33.

[10] Lawry Aguila, A., Chapman, J., Altmann, A. (2023). Multi-modal Variational Autoencoders for normative modelling across multiple imaging modalities. arXiv

[11] Cao, Y., & Fleet, D. (2014). Generalized Product of Experts for Automatic and Principled Fusion of Gaussian Process Predictions. arXiv.

[12] Zbontar, J., Jing, L., Misra, I., LeCun, Y., & Deny, S. (2021). Barlow Twins: Self-Supervised Learning via Redundancy Reduction. International Conference on Machine Learning.

[13] Chapman et al., (2021). CCA-Zoo: A collection of Regularized, Deep Learning based, Kernel, and Probabilistic CCA methods in a scikit-learn style framework. Journal of Open Source Software, 6(68), 3823, https://doi.org/10.21105/joss.03823

[14] Lee, M., Pavlovic, V. (2020). Private-Shared Disentangled Multimodal VAE for Learning of Hybrid Latent Representations. arXiv.

1.3 User Guide

User guide for initialising and running models from the `multi-view-AE` library.

1.3.1 Initialise model

```
from multiviewae import mVAE, mcVAE
from torchvision import datasets, transforms

MNIST_1 = datasets.MNIST('./data/MNIST', train=True, download=True, transform=transforms.
↪Compose([
       transforms.ToTensor(),
]))



data_1 = MNIST_1.train_data[:, :, :14].reshape(-1,392).float()/255.
data_2 = MNIST_1.train_data[:, :, 14:].reshape(-1,392).float()/255.

mvae = mVAE(cfg="./examples/config/example_mnist.yaml",
     input_dim=[392, 392],
     z_dim=64)

mcvae = mcVAE(cfg="./examples/config/example_mnist.yaml",
     input_dim=[392, 392],
     z_dim=64)
```

The dimensions of the input data, `input_dim`, must be provided however the path to a configuration file, `cfg`, and number of latent dimensions, `z_dim`, are optional. Setting `z_dim` will override the value given in the configuration file.

1.3.2 Model fit

```
mvae.fit(data_1, data_2,  max_epochs=500, batch_size=1000)
mcvae.fit(data_1, data_2,  max_epochs=500, batch_size=1000)
```

When fitting the model, the user must provide input each view of the training data. The user can optionally provide the `max_epochs` and `batch_size`. These would override the settings in the configuration file.

1.3.3 Model predictions

We can use a trained model to predict the latent dimensions or reconstructions. The structure of the latent and reconstruction list will depend on the type of model. Below shows an example for joint, `MVAE`, and coordinate, `mcVAE`, multi-view VAE models.

```
MNIST_1 = datasets.MNIST('./data/MNIST', train=False, download=True,␣
↪transform=transforms.Compose([
     transforms.ToTensor()]))

data_test_1 = MNIST_1.test_data[:, :, :14].reshape(-1,392).float()/255.
data_test_2 = MNIST_1.test_data[:, :, 14:].reshape(-1,392).float()/255.

mvae_latent = mvae.predict_latents(data_test_1, data_test_2)

mcvae_latent = mcvae.predict_latents(data_test_1, data_test_2)

mcvae_latent_view1, mcvae_latent_view2 = mcvae_latent[0], mcvae_latent[1]

mvae_reconstruction = mvae.predict_reconstruction(data_test_1, data_test_2)

mvae_reconstruction_view1 = mvae_reconstruction[0][0] #view 1 reconstruction from joint␣
↪latent
mvae_reconstruction_view2 = mvae_reconstruction[0][1] #view 2 reconstruction from joint␣
↪latent

mcvae_reconstruction = mcvae.predict_reconstruction(data_test_1, data_test_2)

mcvae_reconstruction_view1_latent1 = mcvae_reconstruction[0][0] #view 1 reconstruction␣
↪from latent 1
mcvae_reconstruction_view2_latent1 = mcvae_reconstruction[0][1] #view 2 reconstruction␣
↪from latent 1

mcvae_reconstruction_view1_latent2 = mcvae_reconstruction[1][0] #view 1 reconstruction␣
↪from latent 2
mcvae_reconstruction_view2_latent2 = mcvae_reconstruction[1][1] #view 2 reconstruction␣
↪from latent 2
```

1.3.4 Model results

We can explore the model results, for example using `matplotlib`.

```
import matplotlib.pyplot as plt #NOTE: matplotlib is not installed with the library and␣
↪must be installed separately

#Reconstruction plots - how well can the VAE do same view reconstruction?

data_sample = data_test_1[20]
#indices: view 1 latent, view 1 decoder, sample 21
pred_sample = mcvae_reconstruction_view1_latent1[20]

fig, axarr = plt.subplots(1, 2)
axarr[0].imshow(data_sample.reshape(28,14))
axarr[1].imshow(pred_sample.reshape(28,14))
plt.show()
plt.close()

#Reconstruction plots - how well can the VAE do cross view reconstruction?

#indices: view 1 latent, view 2 decoder, sample 21
data_sample = data_test_2[20]
pred_sample = mcvae_reconstruction_view2_latent1[20]

fig, axarr = plt.subplots(1, 2)
axarr[0].imshow(data_sample.reshape(28,14))
axarr[1].imshow(pred_sample.reshape(28,14))
plt.show()
plt.close()
```

1.3.5 Model loading

Trained models can be loaded from the specified path.

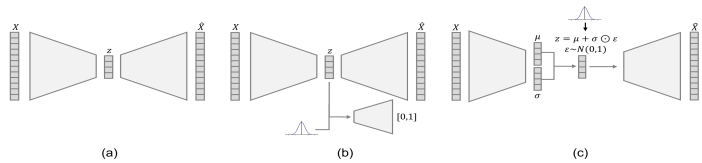
```
from multiviewae import mVAE
from os.path import join

#change the path below to your model path
mvae = mVAE.load_from_checkpoint(join('path/to/model', 'model.ckpt'))
```

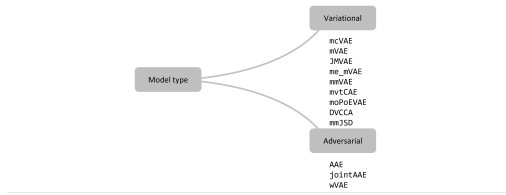
1.4 Model frameworks

There is one vanilla multi-view autoencoder framework, the `AE` model. The remaining models can be grouped into adversarial and variational models which are models regularised by matching the aggregate or marginal encoding distributions, respectively, to a prior distribution. The figure below shows the frameworks for the (a) vanilla autoencoder (b) adversarial autoencoder and (c) variational autoencoder for a single view.

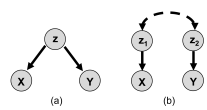
The figure below shows the adversarial and variational groupings of models within the `multi-view-AE` framework.

When extending autoencoders to multiple views, we can assume two latent models. The figure below shows the latent model for (a) a coordinated model which assumes separate latents for each view which are associated with each other





and (b) a joint model which assumes a shared latent across views for data X and Y.



1.5 Parameter settings and configurations

This file containing information on parameters settings in the configuration files, how to set them, and allowed combinations of parameters.

1.5.1 How to specify the yaml configuration file

Most parameters (with the exception of those discussed in the User Guide) are set using configuration yaml files. There are some example yaml files given in the `multi-view-AE/tests/user_config/` folder. To specify a configuration file, the user must specify the absolute or relative path to the yaml file when initialising the relevant model:

```
from multiviewae import mVAE

mvae = mVAE(cfg='./config_folder/test_config.yaml',
      input_dim=[20, 20],
      z_dim=2)
```

If no configuration file is specified, the default configuration for that model class is used. These can be found in the `multi-view-AE/multiviewae/configs/model_type/` folder.

1.5.2 Configuration file structure

The configuration file has the following model parameter groupings which can be edited by the user.

Model

The global model parameter settings.

```
model:
  save_model: True

  seed_everything: True
  seed: 42

  z_dim: 5
  learning_rate: 0.001

  sparse: False
```

There are also a number of model specific parameters which are set in the yaml files in the `multi-view-AE/multiviewae/configs/model_type/` folder.

Datamodule

The parameters for the PyTorch data module class to access and process the data.

```
datamodule:
  _target_: multiviewae.base.dataloaders.MultiviewDataModule

  batch_size: null
  is_validate: True

  train_size: 0.9
```

When the batch size is set to `null` the full batch is used for training at each epoch.

MLP Encoder

The encoder function parameters. The default encoder function is a MLP encoder network:

```
encoder:
  default:
      _target_: multiviewae.architectures.mlp.Encoder

      hidden_layer_dim: []
      bias: True
      non_linear: False

      enc_dist:
        _target_: multiviewae.base.distributions.Default
```

The `encoder._target_` parameter specifies the encoder function class of which the in-built options include: `multiviewae.architectures.mlp.Encoder` and `multiviewae.architectures.mlp.VariationalEncoder`.

The `encoder.enc_dist._target_` parameter specifies the encoding distribution class of which the in-built options include: `multiviewae.base.distributions.Default`, `multiviewae.base.distributions.Normal` and `multiviewae.base.distributions.MultivariateNormal`. The `multiviewae.base.distributions.Default` class is used for the vanilla autoencoder and adversarial autoencoder implementations where no distribution is specified.

The user can specify separate parameters for the encoder network of each view. For example:

```
encoder:
  enc0:
      _target_: multiviewae.architectures.mlp.Encoder

      hidden_layer_dim: [12, 6]
      bias: True
      non_linear: False

      enc_dist:
        _target_: multiviewae.base.distributions.Default
  enc1:
      _target_: multiviewae.architectures.mlp.Encoder

      hidden_layer_dim: [50, 6]
      bias: True
      non_linear: True

      enc_dist:
        _target_: multiviewae.base.distributions.Default
```

where `enc0` and `enc1` provide the parameters for view 0 encoder and view 1 encoder respectively. If no view specific parameters are provided, the default network parameters are used.

NOTE: The `default` encoder parameters are used for joint encoding distributions.

CNN Encoder

Alternatively, the user can specify a CNN architecture by setting the `encoder._target_` parameter:

```
encoder:
  default:
      _target_: multiviewae.architectures.cnn.Encoder

      layer0:
        layer: Conv2d
        in_channels: 1
        out_channels: 8
        kernel_size: 4
        stride: 2
        padding: 1

      layer1:
        layer: Conv2d
```

(continues on next page)

(continued from previous page)

```
        in_channels: 8
        out_channels: 16
        kernel_size: 4
        stride: 2
        padding: 1

      layer2:
        layer: Conv2d
        in_channels: 16
        out_channels: 32
        kernel_size: 4
        stride: 2
        padding: 1

      layer3:
        layer: Conv2d
        in_channels: 32
        out_channels: 64
        kernel_size: 4
        stride: 2
        padding: 0

      layer5:
        layer: AdaptiveAvgPool2d
        output_size: 1

      layer6:
        layer: Flatten
        start_dim: 1

      layer7:
        layer: Linear
        in_features: 64
        out_features: 128

      bias: True
      non_linear: False

      enc_dist:
        _target_: multiviewae.base.distributions.Default
```

In-built options include: `multiviewae.architectures.cnn.Encoder` and `multiviewae.architectures.cnn.VariationalEncoder`. As with the MLP architectures, the user can chose to set view specific parameters. Each layer can be `torch.nn` `Conv2d` layers or any suitable 2D pooling or padding layers.

NOTE: The user is responsible for ensuring that the CNN encoder and decoder network architectures are compatible and create an output tensor of the correct dimensionality.

MLP Decoder

The decoder function parameters. The default decoder function is a MLP decoder network:

```
decoder:
  default:
      _target_: multiviewae.architectures.mlp.Decoder

      hidden_layer_dim: []
      bias: True
      non_linear: False

      dec_dist:
        _target_: multiviewae.base.distributions.Default
```

The `decoder._target_` parameter specifies the encoder function class of which the in-built options include: `multiviewae.architectures.mlp.Decoder` and `multiviewae.models.layers.VariationalDecoder`.

The `decoder.dec_dist._target_` parameter specifies the decoding distribution class of which the in-built options include: `multiviewae.base.distributions.Default`, `multiviewae.base.distributions.Normal`, `multiviewae.base.distributions.MultivariateNormal`, `multiviewae.base.distributions.Laplace` and `multiviewae.base.distributions.Bernoulli`. The `multiviewae.base.distributions.Default` class is used for the vanilla autoencoder and adversarial autoencoder implementations where no distribution is specified.

The user can specify separate parameters for the encoder network of each view. For example:

```
decoder:
  dec0:
      _target_: multiviewae.architectures.mlp.Decoder

      hidden_layer_dim: [6, 12]
      bias: True
      non_linear: False

      dec_dist:
        _target_: multiviewae.base.distributions.Default
  dec1:
      _target_: multiviewae.architectures.mlp.Decoder

      hidden_layer_dim: [6, 50]
      bias: True
      non_linear: True

      dec_dist:
        _target_: multiviewae.base.distributions.Default
```

where `dec0` and `dec1` provide the parameters for view 0 decoder and view 1 decoder respectively. If no view specific parameters are provided, the default network parameters are used.

CNN Decoder

Alternatively, the user can specify a CNN architecture by setting the `encoder._target_` parameter:

```
decoder:
  default:
      _target_: multiviewae.architectures.cnn.Decoder

      layer0:
        layer: Linear
        out_features: 128

      layer1:
        layer: Linear
        in_features: 128
        out_features: 64

      layer2:
        layer: Unflatten
        dim: 1
        unflattened_size: [64, 1, 1]

      layer3:
        layer: ConvTranspose2d
        in_channels: 64
        out_channels: 32
        kernel_size: 4
        stride: 2
        padding: 0

      layer4:
        layer: ConvTranspose2d
        in_channels: 32
        out_channels: 16
        kernel_size: 4
        stride: 2
        padding: 1

      layer5:
        layer: ConvTranspose2d
        in_channels: 16
        out_channels: 8
        kernel_size: 4
        stride: 2
        padding: 1

      layer6:
        layer: ConvTranspose2d
        in_channels: 8
        out_channels: 1
        kernel_size: 4
        stride: 2
        padding: 1
```

(continues on next page)

(continued from previous page)

```
    bias: True
    non_linear: False

    dec_dist:
      _target_: multiviewae.base.distributions.Default
```

NOTE: The user is responsible for ensuring that the CNN encoder and decoder network architectures are compatible and create an output tensor of the correct dimensionality.

Prior

The parameters of the prior distribution for variational models.

```
prior:
  _target_: multiviewae.base.distributions.Normal
  loc: 0
  scale: 1
```

The prior can take the form of a univariate gaussian, `multiviewae.base.distributions.Normal`, or multivariate gaussian, `multiviewae.base.distributions.MultivariateNormal`, with diagonal covariance matrix with variances given by the `scale` parameter.

Trainer

The parameters for the PyTorch trainer. Please see the PyTorch Lightning documentation for more information on the parameter settings.

```
trainer:
  _target_: pytorch_lightning.Trainer

  accelerator: "auto"

  max_epochs: 10

  deterministic: false
  log_every_n_steps: 2
```

Callbacks

Parameters for the PyTorchLightning callbacks. Please see the PyTorch Lightning documentation for more information on the parameter settings.

```
callbacks:
  model_checkpoint:
    _target_: pytorch_lightning.callbacks.ModelCheckpoint
    monitor: "val_loss"
    mode: "min"
    save_last: True
    dirpath: ${out_dir}
```

(continues on next page)

(continued from previous page)

```
  early_stopping:
    _target_: pytorch_lightning.callbacks.EarlyStopping
    monitor: "val_loss"
    mode: "min"
    patience: 50
    min_delta: 0.001
    verbose: True
```

Only the `model_checkpoint` and `early_stopping` callbacks are used in the `multi-view-AE` library. However for more callback options, please refer to the PyTorch Lightning documentation.

Logger

The parameters of the logger file.

```
logger:
  _target_: pytorch_lightning.loggers.tensorboard.TensorBoardLogger

  save_dir: ${out_dir}/logs
```

In the `multi-view-AE` we use TensorBoard for logging. However, the user is free to use whichever logging framework their prefer. **NOTE:** other logging frameworks have not been tested.

1.5.3 Changing parameter settings

Only the grouping header, sub header and the parameters the user wishes to change need to be specified in the users yaml file. The default model parameters are used for the remaining parameters. For example, to change the number of hidden layers for the encoder and decoder networks the user can use the following yaml file:

```
encoder:
  hidden_layer_dim: [10, 5]

decoder:
  hidden_layer_dim: [10, 5]
```

NOTE: An exception to this rule are the Pytorch callbacks where all the parameters for the relevant callback must be specified again in the user configuration file. For example to change the early stopping patience to `100` of the following callback:

```
callbacks:
  early_stopping:
    _target_: pytorch_lightning.callbacks.EarlyStopping
    monitor: "val_loss"
    mode: "min"
    patience: 50
    min_delta: 0.001
    verbose: True
```

The user must add the following section to their yaml file:

```yaml
callbacks:
  early_stopping:
    _target_: pytorch_lightning.callbacks.EarlyStopping
    monitor: "val_loss"
    mode: "min"
    patience: 100
    min_delta: 0.001
    verbose: True
```

1.5.4 Target classes

There are a number of model classes specified in the configuration file, namely; the encoder and decoder functions, the encoder, decoder, and prior distributions for variational models, and the discriminator function for adversarial models. There are a number of existing classes built into the `multi-view-AE` framework for the user to chose from. Alternatively, the user can use their own classes and specify them in the yaml file:

```yaml
encoder:
  _target_: encoder_folder.user_encoder

decoder:
  _target_: decoder_folder.user_decoder
```

However, for these classes to work with the `multi-view-AE` framework, user class implementations must follow the same structure as existing classes. For example, an `encoder` implementation must have a `forward` method.

1.5.5 Allowed parameter combinations

Some parameter combinations are not compatible in the `multi-view-AE` framework. If an incorrect parameter combination is given in the configuration file, either a warning or error is raised depending on whether the parameter choices can be ignored or would impede the model from functioning correctly.

1.6 Limitations and user implementations

There are a number of limitations and allowed parameter combinations within the `multi-view-AE` framework. These restrictions are set in the `multiviewae/base/validation.py` file and will need to be updated should the user wish to add their own implementations. Allowed parameter types are also set in the `multiviewae/base/validation.py` file.

1.6.1 Limitations and allowed parameter combinations

Distribution classes

It should be noted that currently the multivariate normal class, `multiviewae.base.distributions.MultivariateNormal`, implements a multivariate gaussian with a diagonal covariance matrix. Further work will involve implementing a multivariate normal class where the off-diagonal elements of the covariance matrix can be specified or learnt.

Encoder distribution

The `multiviewae.base.distributions.Default` class must be used for the vanilla autoencoder and adversarial autoencoder implementations where no distribution is specified.

Either the `multiviewae.base.distributions.Normal` or `multiviewae.base.distributions.MultivariateNormal` classes must be used for variational models.

For adversarial autoencoders with gaussian posterior, i.e. gaussian encoding distributions, the `multiviewae.base.distributions.Normal` or `multiviewae.base.distributions.MultivariateNormal` classes can be used if their are coupled with a variational encoder architecture, e.g. `multiviewae.architectures.mlp.VariationalEncoder`.

Encoder and prior distribution combinations

Currently, the encoder distribution must be the same as the prior distribution.

Models which support CNN architectures

Many of the autoencoder models in the `multi-view-AE` support CNN encoder and decoder network architectures. The `JMVAE` and `MMVAE` models do not currently support these architectures. This will be addressed in further work.

1.6.2 Adding user designed classes

Users are able to implement their own network architectures, datamodules, datasets and distributions. This should cover the majority of classes the user should wish to edit and offers lots of flexibility when implementing a model. For any other classes, users must have access to the source code and the class must be added to the supported classes in the `multiviewae/base/validation.py` file.

User designed network architectures

User designed MLP network classes must be implemented in a `mlp.py` file and named one of; `Encoder`, `VariationalEncoder`, `Decoder`, and `VariationalDecoder` depending on the network type. CNN network classes must be implemented in a `cnn.py` file and named one of; `Encoder`, `VariationalEncoder`, and `Decoder` depending on the network type.

Networks must except and return the same parameters as the respective `multi-view-AE` counterpart. For example, variational encoder networks must return `mu` and `logvar` in the form of a `Torch.tensor`. Please see the *Architectures* section for information on input and output parameters of encoder and decoder networks.

Implemented classes must have a `forward` method.

User designed datamodules and datasets

User designed datamodules must implement a `setup` method, a `train_dataloader` method and a `val_dataloader` method and must accept the same parameters as the `multi-view-AE` counterpart. The datamodule class name must end in `DataModule`. User designed datasets must implement a `__getitem__` method and must accept the same parameters as the `multi-view-AE` counterpart. The dataset class name must end in `Dataset`. **NOTE** User implemented datasets must also provide a `is_path_ds` parameter to indicate whether the input data is a path to the data for the data to be loaded when the `__getitem__` method is called or whether the data is stored in memory.

User designed distributions

User designed distributions must implement a `log_likelihood` and `_sample` method.

1.7 Models

class `multiviewae.models.AE`(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multi-view Autoencoder model with a separate latent representation for each view.

Parameters

- **cfg** (*str*) – Path to configuration file.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of latent dimensions for each view of type torch.Tensor.

Return type
z (list)

decode(*z*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
list of data reconstructions.

Return type
x_recon (list)

forward(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing list of data reconstructions (x_recon) and latent dimensions (z).

Return type
fwd_rtn (dict)

loss_function(*x*, *fwd_rtn*)

Calculate reconstruction loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.

- **fwd_rtn** (*dict*) – dictionary containing list of data reconstructions (x_recon) and latent dimensions (z).

Returns
dictionary containing reconstruction loss.

Return type
losses (dict)

class multiviewae.models.**mAAE**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multi-view Adversarial Autoencoder model with a separate latent representation for each view.

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - eps (float): Value added for numerical stability.
  - discriminator._target_ (multiviewae.architectures.mlp.Discriminator): Discriminator network class.
  - discriminator.hidden_layer_dim (list): Number of nodes per hidden layer.
  - discriminator.bias (bool): Whether to include a bias term in hidden layers.
  - discriminator.non_linear (bool): Whether to include a ReLU() function between layers.
  - discriminator.dropout_threshold (float): Dropout threshold of layers.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of latent dimensions for each view of type torch.Tensor.

Return type
z (list)

decode(*z*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
list of decoding distributions.

Return type
px_zs (list)

disc(*z*)

Forward pass of "real" samples from gaussian prior and "fake" samples from encoders through the discriminator network.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
Discriminator network output for "real" samples. d_fake (list): list of discriminator network output for "fake" samples.

Return type
d_real (torch.Tensor)

`forward_recon`(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing decoding distributions (px_zs) and latent dimensions (z).

Return type
fwd_rtn (dict)

`forward_discrim`(*x*)

Apply encode and disc methods to input data to generate discriminator prediction on the latent dimensions and train discriminator parameters.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing discriminator output from "real" samples (d_real), discriminator output from "fake" samples (d_fake), and latent dimensions (z).

Return type
fwd_rtn (dict)

`forward_gen`(*x*)

Apply encode and disc methods to input data to generate discriminator prediction on the latent dimensions and train encoder parameters.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
fwd_rtn (dict): dictionary containing discriminator output from "fake" samples (d_fake) and latent dimensions (z).

Return type
fwd_rtn (dict)

`recon_loss`(*x*, *fwd_rtn*)

Calculate reconstruction loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – fwd_rtn from the forward_recon method.

Returns
Reconstruction error.

Return type
ll (torch.Tensor)

generator_loss(*fwd_rtn*)

Calculate the generator loss.

Parameters
fwd_rtn (*dict*) – fwd_rtn from the forward_gen method.

Returns
Generator loss.

Return type
gen_loss (torch.Tensor)

discriminator_loss(*fwd_rtn*)

Calculate the discriminator loss.

Parameters
fwd_rtn (*dict*) – fwd_rtn from the forward_discrim method.

Returns
Discriminator loss.

Return type
disc_loss (torch.Tensor)

class multiviewae.models.**mWAE**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multi-view Adversarial Autoencoder model with wasserstein loss.

Wasserstein autoencoders: https://arxiv.org/abs/1711.01558

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - discriminator._target_ (multiviewae.architectures.mlp.Discriminator): Discriminator network class.
  - discriminator.hidden_layer_dim (list): Number of nodes per hidden layer.
  - discriminator.bias (bool): Whether to include a bias term in hidden layers.
  - discriminator.non_linear (bool): Whether to include a ReLU() function between layers.
  - discriminator.dropout_threshold (float): Dropout threshold of layers.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of latent dimensions for each view of type torch.Tensor.

Return type
z (list)

decode(*z*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
list of decoding distributions.

Return type
px_zs (list)

`disc`(*z*)

Forward pass of "real" samples from gaussian prior and "fake" samples from encoders through the discriminator network.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
Discriminator network output for "real" samples. d_fake (list): list of discriminator network output for "fake" samples.

Return type
d_real (torch.Tensor)

`forward_recon`(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing decoding distributions (px_zs) and latent dimensions (z).

Return type
fwd_rtn (dict)

`forward_discrim`(*x*)

Apply encode and disc methods to input data to generate discriminator prediction on the latent dimensions and train discriminator parameters.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing discriminator output from "real" samples (d_real), discriminator output from "fake" samples (d_fake), and latent dimensions (z).

Return type
fwd_rtn (dict)

`forward_gen`(*x*)

Apply encode and disc methods to input data to generate discriminator prediction on the latent dimensions and train encoder parameters.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
fwd_rtn (dict): dictionary containing discriminator output from "fake" samples (d_fake) and latent dimensions (z).

Return type
fwd_rtn (dict)

recon_loss(*x*, *fwd_rtn*)

Calculate reconstruction loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – fwd_rtn from the forward_recon method.

Returns
Reconstruction error.

Return type
ll (torch.Tensor)

generator_loss(*fwd_rtn*)

Calculate the generator loss.

Parameters
fwd_rtn (*dict*) – fwd_rtn from the forward_gen method.

Returns
Generator loss.

Return type
gen_loss (torch.Tensor)

discriminator_loss(*fwd_rtn*)

Calculate the discriminator loss.

Parameters
fwd_rtn (*dict*) – fwd_rtn from the forward_discrim method.

Returns
Discriminator loss.

Return type
disc_loss (torch.Tensor)

class multiviewae.models.**mcVAE**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multi-Channel Variational Autoencoder and Sparse Multi-Channel Variational Autoencoder.

Code is based on: https://github.com/ggbioing/mcvae

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.sparse (bool): Whether to enforce sparsity of the encoding distribution.
  - model.threshold (float): Dropout threshold applied to the latent dimensions. Default is 0.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.

- decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Antelmi, Luigi & Ayache, Nicholas & Robert, Philippe & Lorenzi, Marco. (2019). Sparse Multi-Channel Variational Autoencoder for the Joint Analysis of Heterogeneous Data.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of encoding dimensions for each view.

Return type
qz_xs (list)

decode(*qz_xs*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
A nested list of decoding distributions. The outer list has a n_view element indicating latent dimensions index. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
px_zs (list)

forward(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding (qz_xs) and decoding (px_zs) distributions.

Return type
fwd_rtn (dict)

loss_function(*x*, *fwd_rtn*)

Calculate mcVAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the mcVAE loss.

Return type
losses (dict)

`calc_kl`(*qz_xs*)

Calculate mcVAE KL-divergence loss.

Parameters
qz_xs (*list*) – list of encoding distributions.

Returns
KL-divergence loss across all views.

Return type
(torch.Tensor)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

class `multiviewae.models.`**`mVAE`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multimodal Variational Autoencoder (MVAE).

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.join_type (str): Method of combining encoding distributions.
  - model.warmup (int): KL term weighted by beta linearly increased to 1 over this many epochs.
  - model.use_prior (bool): Whether to use a prior expert when combining encoding distributions.
  - model.sparse (bool): Whether to enforce sparsity of the encoding distribution.
  - model.threshold (float): Dropout threshold applied to the latent dimensions. Default is 0.
  - model.weight_ll (bool): Whether to weight the log-likelihood loss by 1/n_views.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.

- decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Wu, M., & Goodman, N.D. (2018). Multimodal Generative Models for Scalable Weakly-Supervised Learning. NeurIPS.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list of joint encoding distribution.

Return type
(list)

encode_subset(*x*, *subset*)

Forward pass through encoder networks for the specified subset.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list of joint encoding distribution.

Return type
(list)

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*qz_x*, *subset*)

Forward pass of joint latent dimensions through decoder networks for the specified subset.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

`forward`(*x*)

Apply encode and decode methods to input data to generate the joint latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

`calc_kl`(*qz_x*)

Calculate KL-divergence loss.

Parameters
qz_xs (*list*) – Single element list containing joint encoding distribution.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate Multimodal VAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the MVAE loss.

Return type
losses (dict)

`calc_nll`(*x*, *K=1000*, *batch_size_K=100*)

Calculate negative log-likelihood used to evaluate model performance.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Negative log-likelihood.

Return type
nll (torch.Tensor)

class `multiviewae.models.JMVAE`(*cfg=None*, *input_dim=None*, *z_dim=None*)

JMVAE-kl.

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters: - alpha (float): Weighting of KL-divergence loss from individual encoders. - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of Encoder to use. - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution. - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use. - decoder.default.init_logvar(int, float): Initial value for log variance of decoder. - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Suzuki, Masahiro & Nakayama, Kotaro & Matsuo, Yutaka. (2016). Joint Multimodal Learning with Deep Generative Models.

encode(*x*)

Forward pass through joint encoder network.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list containing joint encoding distribution, qz_xy.

Return type
(list)

encode_separate(*x*)

Forward pass through separate encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Encoding distribution for modality X. qz_y: Encoding distribution for modality Y.

Return type
qz_x

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a 2 element list with the position in the list indicating the decoder index.

Return type
(list)

`forward`(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

`calc_kl`(*qz_xy*, *qz_x*, *qz_y*)

Calculate JMVAE-kl KL-divergence loss.

Parameters

- **qz_xy** (*list*) – Single element list containing shared encoding distribution.
- **qz_x** (*list*) – Single element list containing encoding distribution for modality X.
- **qz_y** (*list*) – Single element list containing encoding distribution for modality Y.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate JMVAE-kl loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the JMVAE loss.

Return type
losses (dict)

`calc_nll`(*x*, *K=1000*, *batch_size_K=100*)

Calculate negative log-likelihood used to evaluate model performance.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Negative log-likelihood.

Return type
nll (torch.Tensor)

class `multiviewae.models.me_mVAE`(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multimodal Variational Autoencoder (MVAE).

Loss optimises the ELBO term from the joint posterior distribution, as well as the separate ELBO terms for each view. me_mVAE stands for multi ELBO Multimodal VAE

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.join_type (str): Method of combining encoding distributions.
  - model.warmup (int): KL term weighted by beta linearly increased to 1 over this many epochs.
  - model.use_prior (bool): Whether to use a prior expert when combining encoding distributions.
  - model.sparse (bool): Whether to enforce sparsity of the encoding distribution.
  - model.threshold (float): Dropout threshold applied to the latent dimensions. Default is 0.
  - model.weight_kld (bool): Whether to weight the KL term by the number of views.
  - model.weight_ll (bool): Whether to weight the log-likelihood term by the number of views.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Wu, M., & Goodman, N.D. (2018). Multimodal Generative Models for Scalable Weakly-Supervised Learning. NeurIPS.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list of joint encoding distribution. (list): List containing separate encoding distributions.

Return type
(list)

encode_subset(*x*, *subset*)

Forward pass through encoder networks for the specified subset.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list of joint encoding distribution.

Return type
(list)

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*qz_x*, *subset*)

Forward pass of joint latent dimensions through decoder networks for the specified subset.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_separate(*qz_xs*)

Forward pass of each view specific latent dimensions through the respective decoder network.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the view specific latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

`forward`(*x*)

Apply encode, decode, encode_separate and decode_separate methods to input data to generate the joint and modality specific latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

`calc_kl`(*qz_xs*)

Calculate KL-divergence loss.

Parameters
qz_xs (*list*) – list of encoding distributions.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate multi ELBO Multimodal VAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the MVAE loss.

Return type
losses (dict)

`calc_nll`(*x*, *K=1000*, *batch_size_K=100*)

Calculate negative log-likelihood used to evaluate model performance.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Negative log-likelihood.

Return type
nll (torch.Tensor)

class `multiviewae.models.mmVAE`(*cfg=None*, *input_dim=None*, *z_dim=None*)

Mixture-of-Experts Multimodal Variational Autoencoder (MMVAE).

Code is based on: https://github.com/iffsid/mmvae

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.K (int): Number of samples to take from encoding distribution.
  - model.DREG_loss (bool): Whether to use DReG estimator when using large K value.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Shi, Y., Siddharth, N., Paige, B., & Torr, P.H. (2019). Variational Mixture-of-Experts Autoencoders for Multi-Modal Deep Generative Models. ArXiv, abs/1911.03393.

`encode`(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of encoding distributions.

Return type
(list)

encode_subset(*x*, *subset*)

Forward pass through encoder networks for a subset of modalities. :param x: list of input data of type torch.Tensor. :type x: list :param subset: list of modalities to encode. :type subset: list

Returns
list of encoding distributions.

Return type
(list)

decode(*qz_xs*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions. The outer list has a n_view element indicating latent dimensions index. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*qz_xs*, *subset*)

Forward pass through decoder networks for a subset of modalities. Each latent is passed through its own decoder.

Parameters

- **qz_xs** (*list*) – list of encoding distributions.
- **subset** (*list*) – list of modalities to decode.

Returns
A nested list of decoding distributions. The outer list is a single element list, the inner list is a subset element list of decoding distributions.

Return type
(list)

forward(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding (qz_xs) and decoding (px_zs) distributions.

Return type
fwd_rtn (dict)

loss_function(*x*, *fwd_rtn*)

Wrapper function for mmVAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing mmVAE loss.

Return type
losses (dict)

`moe_iwae`(*x*, *qz_xs*, *px_zs*)

Calculate Mixture-of-Experts importance weighted autoencoder (IWAE) loss used for the mmVAE model.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
the output tensor.

Return type
(torch.Tensor)

`log_mean_exp`(*value*, *dim=0*, *keepdim=False*)

Returns the log of the mean of the exponentials along the given dimension (dim).

Parameters

- **value** (*torch.Tensor*) – the input tensor.
- **dim** (*int, optional*) – the dimension along which to take the mean.
- **keepdim** (*bool, optional*) – whether the output tensor has dim retained or not.

Returns
the output tensor.

Return type
(torch.Tensor)

class `multiviewae.models.`**`mvtCAE`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multi-View Total Correlation Auto-Encoder (MVTCAE).

Code is based on: https://github.com/gr8joo/MVTCAE

NOTE: This implementation currently only caters for a PoE posterior distribution. MoE and MoPoE posteriors will be included in further work.

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.alpha (int, float): Log likelihood, Conditional VIB and VIB weighting term.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar(int, float): Initial value for log variance of decoder.

  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Hwang, HyeongJoo and Kim, Geon-Hyeong and Hong, Seunghoon and Kim, Kee-Eung. Multi-View Representation Learning via Total Correlation Objective. 2021. NeurIPS

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
qz_xs (list): list containing separate encoding distributions. qz_x (list): Single element list containing PoE joint encoding distribution.

Return type
Returns the separate and/or joint encoding distributions depending on whether the model is in the training stage

encode_subset(*x*, *subset*)

Forward pass through encoder networks for a subset of modalities.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **subset** (*list*) – list of modalities to encode.

Returns
qz_xs (list): list containing separate encoding distributions. qz_x (list): Single element list containing PoE joint encoding distribution.

Return type
Returns either the joint or separate encoding distributions depending on whether the model is in the training stage

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
qz_x (*list*) – list of joint encoding distribution.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*qz_x*, *subset*)

Forward pass of joint latent dimensions through decoder networks for a subset of modalities.

forward(*x*)

Apply encode and decode methods to input data to generate the joint latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

loss_function(*x*, *fwd_rtn*)

Calculate MVTCAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the MVTCAE loss.

Return type
losses (dict)

calc_kl_cvib(*qz_x*, *qz_xs*)

Calculate KL-divergence between PoE joint encoding distribution and the encoding distribution for each view.

Parameters
qz_xs (*list*) – list of encoding distributions of each view.

Returns
KL-divergence loss.

Return type
kl (torch.Tensor)

calc_kl_groupwise(*qz_x*)

Calculate KL-divergence between the PoE joint encoding distribution and the prior distribution.

Parameters
qz_xs (*list*) – list of encoding distributions of each view.

Returns
KL-divergence loss.

Return type
kl (torch.Tensor)

calc_ll(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

class multiviewae.models.**DVCCA**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Deep Variational Canonical Correlation Analysis (DVCCA).

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.private (bool): Whether to include private view-specific latent dimensions.
  - model.sparse (bool): Whether to enforce sparsity of the encoding distribution.
  - model.threshold (float): Dropout threshold applied to the latent dimensions. Default is 0.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar(int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Wang, Weiran & Lee, Honglak & Livescu, Karen. (2016). Deep Variational Canonical Correlation Analysis.

configure_optimizers()

Configure optimizers for encoder, private encoder, and decoder network parameters.

Returns
list of Adam optimizers for encoders and decoders.

Return type
optimizers (list)

encode(*x*)

Forward pass through encoder network. For DVCCA-private a forward pass is performed through each private encoder and the output latent is concatenated with the shared latent.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
qz_x (list): list containing the shared encoding distribution. qz_xs (list): list of encoding distributions for shared and private latents of DVCCA-private. qh_xs (list): list of encoding distributions for private latents of DVCCA-private.

Return type
Returns a combination of the following depending on the training stage and model type

`decode`(*qz_x*)

Forward pass through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared or shared and private latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

`forward`(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions. For DVCCA, the shared encoding distribution is passed to the decode method. For DVCCA-private, the joint distribution of the shared and private latents for each view is passed to the decode method.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing list of decoding distributions (px_zs), shared encoding distribution (qz_x), and (for DVCCA-private) private encoding distributions (qh_xs).

Return type
fwd_rtn (dict)

`loss_function`(*x*, *fwd_rtn*)

Calculate DVCCA loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing list of decoding distributions (px_zs), shared encoding distribution (qz_x), and (for DVCCA-private) private encoding distributions (qh_xs).

Returns
dictionary containing each element of the DVCCA loss.

Return type
losses (dict)

`calc_kl`(*qz_x*, *qh_xs*)

Wrapper function for calculating KL-divergence loss.

Parameters

- **qz_x** (*list*) – Single element list containing shared encoding distribution.
- **qh_xs** (*list*) – list of private encoding distributions for DVCCA-private.

Returns
KL-divergence loss across all views.

Return type
(torch.Tensor)

calc_kl_(*dist*)

Calculate KL-divergence.

Parameters
dist – Distribution object.

Returns
Kl-divergence.

Return type
(torch.Tensor)

calc_ll(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

class multiviewae.models.**MoPoEVAE**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Mixture-of-Product-of-Experts Variational Autoencoder.

Code is based on: https://github.com/thomassutter/MoPoE

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Sutter, Thomas & Daunhawer, Imant & Vogt, Julia. (2021). Generalized Multimodal ELBO.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list containing the MoE joint encoding distribution. If training, the model also returns the encoding distribution for each subset.

Return type
(list)

encode_subset(*x*, *subset*)

Forward pass through encoder networks for a subset of modalities. :param x: list of input data of type torch.Tensor. :type x: list :param subset: list of modalities to encode. :type subset: list

Returns
list containing the MoE joint encoding distribution.

Return type
(list)

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

forward(*x*)

Apply encode and decode methods to input data to generate the joint and subset latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

loss_function(*x*, *fwd_rtn*)

Calculate MoPoE VAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the MoPoE VAE loss.

Return type
losses (dict)

`calc_kl_moe`(*qz_xs*)

Calculate KL-divergence between the each PoE subset posterior and the prior distribution.

Parameters
qz_xs (*list*) – list of encoding distributions.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`set_subsets`(*n_views=None*)

Create combinations of subsets of views.

Returns
list of unique combinations of n_views.

Return type
subset_list (list)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

class `multiviewae.models.`**`mmJSD`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Multimodal Jensen-Shannon divergence (mmJSD) model with Product-of-Experts dynamic prior.

Code is based on: https://github.com/thomassutter/mmjsd

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.private (bool): Whether to include private modality-specific latent dimensions.
  - model.beta (int, float): KL divergence weighting term.
  - model.alpha (int, float): JSD divergence weighting term.
  - model.s_dim (int): Number of private latent dimensions.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.

- decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
- decoder.default.init_logvar(int, float): Initial value for log variance of decoder.
- decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Sutter, Thomas & Daunhawer, Imant & Vogt, Julia. (2021). Multimodal Generative Learning Utilizing Jensen-Shannon-Divergence. Advances in Neural Information Processing Systems. 33.

encode(*x*)

Forward pass through encoder network. If self.private=True, the first two dimensions of each latent are used for the modality-specific part and the remaining dimensions for the joint content.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns

qz_xs (list): Single element list containing the MoE encoding distribution for self.private=False. qzs_xs (list): list containing each encoding distribution for self.private=False. qz_x (list): Single element list containing the PoE encoding distribution for self.private=False.

qc_x (list): Single element list containing the PoE shared encoding distribution. qscs_xs (list): list containing combined shared and private latents. qs_xs (list): list of encoding distributions for private latents. qcs_xs (list): list containing encoding distributions for shared latent dimensions for each view.

Return type
Returns a combination of the following depending on the training stage and model type

encode_subset(*x*, *subset*)

Forward pass through encoder networks for a subset of modalities. :param x: list of input data of type torch.Tensor. :type x: list :param subset: list of modalities to encode. :type subset: list

Returns
list containing the PoE joint encoding distribution.

Return type
(list)

decode(*qz_x*)

Forward pass of latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element, the inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*qz_x*, *subset*)

Forward pass of latent dimensions through decoder networks for a subset of modalities.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **subset** (*list*) – list of modalities to decode.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element, the inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

forward(*x*)

Apply encode and decode methods to input data to generate the joint and modality specific latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

calc_kl(*qz_xs*)

Calculate KL-divergence loss.

Parameters
qz_xs (*list*) – list of encoding distributions.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

calc_ll(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

calc_jsd(*qcs_xs*, *qc_x*)

Calculate Jensen-Shannon Divergence loss.

Parameters

- **qcs_xs** (*list*) – list of encoding distributions of each view for shared latent dimensions.
- **qc_x** (*list*) – Dynamic prior given by PoE of shared encoding distributions.

Returns
Jensen-Shannon Divergence loss.

Return type
jsd (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate mmJSD loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the mmJSD loss.

Return type
losses (dict)

class `multiviewae.models.`**`weighted_mVAE`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Generalised Product-of-Experts Variational Autoencoder (gPoE-MVAE).

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.beta (int, float): KL divergence weighting term.
  - model.private (bool): Whether to include private view-specific latent dimensions.
  - model.sparse (bool): Whether to enforce sparsity of the encoding distribution.
  - model.threshold (float): Dropout threshold applied to the latent dimensions. Default is 0.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Cao, Y., & Fleet, D. (2014). Generalized Product of Experts for Automatic and Principled Fusion of Gaussian Process Predictions. arXiv. Lawry Aguila, A., Chapman, J., Altmann, A. (2023). Multi-modal Variational Autoencoders for normative modelling across multiple imaging modalities. arXiv

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns

qz_x (list): Single element list containing the PoE encoding distribution for self.private=False.

qc_x (list): Single element list containing the PoE shared encoding distribution. qs_xs (list): list of encoding distributions for private latents. qscs_xs (list): list containing combined shared and private latents. qcs_xs (list): list containing encoding distributions for shared latent dimensions for each view.

Return type
Returns a combination of the following depending on the training stage and model type

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

forward(*x*)

Apply encode and decode methods to input data to generate the joint latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

calc_kl(*qz_x*)

Calculate KL-divergence loss.

Parameters
qz_x (*list*) – Single element list containing joint encoding distribution.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_kl_separate`(*qc_xs*)

Calculate KL-divergence loss.

Parameters
qc_xs (*list*) – list of encoding distributions for private/shared latent dimensions for each view.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll`(*x*, *px_zs*)

Calculate log-likelihood loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate Multimodal VAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the MVAE loss.

Return type
losses (dict)

class `multiviewae.models.DMVAE`(*cfg=None*, *input_dim=None*, *z_dim=None*)

Disentangled multi-modal variational autoencoder (DMVAE)

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model._lambda (list, optional): Log likelihood weighting term for each modality.
  - model.s_dim (int): Number of private latent dimensions.
  - model.beta (int, float): KL divergence weighting term.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.

- decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
- decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
- decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

M. Lee and V. Pavlovic, "Private-Shared Disentangled Multimodal VAE for Learning of Latent Representations," 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Nashville, TN, USA, 2021, pp. 1692-1700, doi: 10.1109/CVPRW53098.2021.00185.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list containing the PoE shared encoding distribution. qcs_xs (list): list containing encoding distributions for shared latent dimensions for each view. qs_xs (list): list of encoding distributions for private latents. qscs_xs (list): nested list containing combined PoE shared and private latents. qscss_xs (list): nested list containing combined shared latents from each modality and private latents for same and cross view reconstruction.

Return type
qc_x (list)

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_separate(*qz_xs*)

Forward pass through decoder networks. Each shared latent is passed through all of the decoders with the private latents from the same view.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a n_view element list with position in the list indicating the decoder index. The inner list is a n_view element list with the position in the list indicating latent dimensions index. NOTE: This is the reverse to other models.

Return type
(list)

`forward`(*x*)

Apply encode and decode methods to input data to generate the latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

`calc_kl_joint_latent`(*qz_x*, *qs_xs*)

Calculate KL-divergence loss for the first terms in Equation 3.

Parameters

- **qz_x** (*list*) – Single element list containing joint encoding distribution.
- **qs_xs** (*list*) – list of encoding distributions for private latent dimensions for each view.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_kl_separate_latent`(*qcs_xs*, *qs_xs*)

Calculate KL-divergence loss for the second terms in Equation 3.

Parameters

- **qcs_x** (*list*) – list of the shared encoding distributions calculated from each view.
- **qs_xs** (*list*) – list of encoding distributions for private latent dimensions for each view.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll_joint`(*x*, *px_zs*)

Calculate log-likelihood loss from the joint encoding distribution for each modality.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`calc_ll_separate`(*x*, *pxs_zs*)

Calculate cross-modal and self-reconstrution log-likelihood loss from the shared encoding distribution for each modality and private latents.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **pxs_zs** (*list*) – nested list of decoding distributons. NOTE: The ordering of decoding distribution is the reverse compared to other models.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`loss_function`(*x*, *fwd_rtn*)

Calculate DMVAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the DMVAE loss.

Return type
losses (dict)

`configure_optimizers`()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
```

(continues on next page)

(continued from previous page)

```
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]
```

(continues on next page)

(continued from previous page)

```
# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

class `multiviewae.models.`**`weighted_DMVAE`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Variant of Disentangled multi-modal variational autoencoder (DMVAE) using weighted Product-of-Experts for joint encoding distribution.

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model._lambda (list, optional): Log likelihood weighting term for each modality.
  - model.s_dim (int): Number of private latent dimensions.
  - model.beta (int, float): KL divergence weighting term.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

M. Lee and V. Pavlovic, "Private-Shared Disentangled Multimodal VAE for Learning of Latent Representations," 2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Nashville, TN, USA, 2021, pp. 1692-1700, doi: 10.1109/CVPRW53098.2021.00185.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
Single element list containing the PoE shared encoding distribution. qcs_xs (list): list containing encoding distributions for shared latent dimensions for each view. qs_xs (list): list of encoding distributions for private latents. qscs_xs (list): nested list containing combined PoE shared and private latents. qscss_xs (list): nested list containing combined shared latents from each modality and private latents for same and cross view reconstruction.

Return type
qc_x (list)

decode(*qz_x*)

Forward pass of joint latent dimensions through decoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a single element indicating the shared latent dimensions. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_separate(*qz_xs*)

Forward pass through decoder networks. Each shared latent is passed through all of the decoders with the private latents from the same view.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
A nested list of decoding distributions, px_zs. The outer list has a n_view element list with position in the list indicating the decoder index. The inner list is a n_view element list with the position in the list indicating latent dimensions index. NOTE: This is the reverse to other models.

Return type
(list)

forward(*x*)

Apply encode and decode methods to input data to generate the latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding and decoding distributions.

Return type
fwd_rtn (dict)

`calc_kl_joint_latent`(*qz_x*, *qs_xs*)

Calculate KL-divergence loss for the first terms in Equation 3.

Parameters

- **qz_x** (*list*) – Single element list containing joint encoding distribution.
- **qs_xs** (*list*) – list of encoding distributions for private latent dimensions for each view.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_kl_separate_latent`(*qcs_xs*, *qs_xs*)

Calculate KL-divergence loss for the second terms in Equation 3.

Parameters

- **qcs_x** (*list*) – list of the shared encoding distributions calculated from each view.
- **qs_xs** (*list*) – list of encoding distributions for private latent dimensions for each view.

Returns
KL-divergence loss.

Return type
(torch.Tensor)

`calc_ll_joint`(*x*, *px_zs*)

Calculate log-likelihood loss from the joint encoding distribution for each modality.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **px_zs** (*list*) – list of decoding distributions.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

`calc_ll_separate`(*x*, *pxs_zs*)

Calculate cross-modal and self-reconstrution log-likelihood loss from the shared encoding distribution for each modality and private latents.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **pxs_zs** (*list*) – nested list of decoding distributons. NOTE: The ordering of decoding distribution is the reverse compared to other models.

Returns
Log-likelihood loss.

Return type
ll (torch.Tensor)

loss_function(*x*, *fwd_rtn*)

Calculate DMVAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing each element of the DMVAE loss.

Return type
losses (dict)

configure_optimizers()

Choose what optimizers and learning-rate schedulers to use in your optimization. Normally you'd need one. But in the case of GANs or similar you might have multiple.

Returns

Any of these 6 options.

- **Single optimizer**.
- **List or Tuple** of optimizers.
- **Two lists** - The first list has multiple optimizers, and the second has multiple LR schedulers (or multiple `lr_scheduler_config`).
- **Dictionary**, with an `"optimizer"` key, and (optionally) a `"lr_scheduler"` key whose value is a single LR scheduler or `lr_scheduler_config`.
- **Tuple of dictionaries** as described above, with an optional `"frequency"` key.
- **None** - Fit will run without any optimizer.

The `lr_scheduler_config` is a dictionary which contains the scheduler and its associated configuration. The default configuration is shown below.

```
lr_scheduler_config = {
    # REQUIRED: The scheduler instance
    "scheduler": lr_scheduler,
    # The unit of the scheduler's step size, could also be 'step'.
    # 'epoch' updates the scheduler on epoch end whereas 'step'
    # updates it after a optimizer update.
    "interval": "epoch",
    # How many epochs/steps should pass between calls to
    # `scheduler.step()`. 1 corresponds to updating the learning
    # rate after every epoch/step.
    "frequency": 1,
    # Metric to to monitor for schedulers like `ReduceLROnPlateau`
    "monitor": "val_loss",
    # If set to `True`, will enforce that the value specified 'monitor'
    # is available when the scheduler is updated, thus stopping
    # training if not found. If set to `False`, it will only produce a warning
    "strict": True,
    # If using the `LearningRateMonitor` callback to monitor the
    # learning rate progress, this keyword can be used to specify
    # a custom logged name
```

(continues on next page)

(continued from previous page)

```
    "name": None,
}
```

When there are schedulers in which the `.step()` method is conditioned on a value, such as the `torch.optim.lr_scheduler.ReduceLROnPlateau` scheduler, Lightning requires that the `lr_scheduler_config` contains the keyword `"monitor"` set to the metric name that the scheduler should be conditioned on.

```
# The ReduceLROnPlateau scheduler requires a monitor
def configure_optimizers(self):
    optimizer = Adam(...)
    return {
        "optimizer": optimizer,
        "lr_scheduler": {
            "scheduler": ReduceLROnPlateau(optimizer, ...),
            "monitor": "metric_to_track",
            "frequency": "indicates how often the metric is updated"
            # If "monitor" references validation metrics, then "frequency"␣
↪should be set to a
            # multiple of "trainer.check_val_every_n_epoch".
        },
    }


# In the case of two optimizers, only one using the ReduceLROnPlateau scheduler
def configure_optimizers(self):
    optimizer1 = Adam(...)
    optimizer2 = SGD(...)
    scheduler1 = ReduceLROnPlateau(optimizer1, ...)
    scheduler2 = LambdaLR(optimizer2, ...)
    return (
        {
            "optimizer": optimizer1,
            "lr_scheduler": {
                "scheduler": scheduler1,
                "monitor": "metric_to_track",
            },
        },
        {"optimizer": optimizer2, "lr_scheduler": scheduler2},
    )
```

Metrics can be made available to monitor by simply logging it using `self.log('metric_to_track', metric_val)` in your `LightningModule`.

Note: The `frequency` value specified in a dict along with the `optimizer` key is an int corresponding to the number of sequential batches optimized with the specific optimizer. It should be given to none or to all of the optimizers. There is a difference between passing multiple optimizers in a list, and passing multiple optimizers in dictionaries with a frequency of 1:

- In the former case, all optimizers will operate on the given batch in each optimization step.
- In the latter, only one optimizer will operate on the given batch at every step.

This is different from the `frequency` value specified in the `lr_scheduler_config` mentioned above.

```
def configure_optimizers(self):
    optimizer_one = torch.optim.SGD(self.model.parameters(), lr=0.01)
    optimizer_two = torch.optim.SGD(self.model.parameters(), lr=0.01)
    return [
        {"optimizer": optimizer_one, "frequency": 5},
        {"optimizer": optimizer_two, "frequency": 10},
    ]
```

In this example, the first optimizer will be used for the first 5 steps, the second optimizer for the next 10 steps and that cycle will continue. If an LR scheduler is specified for an optimizer using the `lr_scheduler` key in the above dict, the scheduler will only be updated when its optimizer is being used.

Examples:

```
# most cases. no learning rate scheduler
def configure_optimizers(self):
    return Adam(self.parameters(), lr=1e-3)

# multiple optimizer case (e.g.: GAN)
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    return gen_opt, dis_opt

# example with learning rate schedulers
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    dis_sch = CosineAnnealing(dis_opt, T_max=10)
    return [gen_opt, dis_opt], [dis_sch]

# example with step-based learning rate schedulers
# each optimizer has its own scheduler
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    gen_sch = {
        'scheduler': ExponentialLR(gen_opt, 0.99),
        'interval': 'step'  # called after each training step
    }
    dis_sch = CosineAnnealing(dis_opt, T_max=10) # called every epoch
    return [gen_opt, dis_opt], [gen_sch, dis_sch]

# example with optimizer frequencies
# see training procedure in `Improved Training of Wasserstein GANs`, Algorithm 1
# https://arxiv.org/abs/1704.00028
def configure_optimizers(self):
    gen_opt = Adam(self.model_gen.parameters(), lr=0.01)
    dis_opt = Adam(self.model_dis.parameters(), lr=0.02)
    n_critic = 5
    return (
        {'optimizer': dis_opt, 'frequency': n_critic},
```

(continues on next page)

(continued from previous page)

```
        {'optimizer': gen_opt, 'frequency': 1}
    )
```

Note: Some things to know:

- Lightning calls `.backward()` and `.step()` on each optimizer and learning rate scheduler as needed.
- If you use 16-bit precision (`precision=16`), Lightning will automatically handle the optimizers.
- If you use multiple optimizers, `training_step()` will have an additional `optimizer_idx` parameter.
- If you use `torch.optim.LBFGS`, Lightning handles the closure function automatically for you.
- If you use multiple optimizers, gradients will be calculated only for the parameters of current optimizer at each training step.
- If you need to control how often those optimizers step or override the default `.step()` schedule, override the `optimizer_step()` hook.

class `multiviewae.models.`**`mmVAEPlus`**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Mixture-of-Experts Multimodal Variational Autoencoder (MMVAE).

Code is based on: https://github.com/iffsid/mmvae

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model.K (int): Number of samples to take from encoding distribution.
  - model.DREG_loss (bool): Whether to use DReG estimator when using large K value.
  - encoder.default._target_ (multiviewae.architectures.mlp.VariationalEncoder): Type of encoder class to use.
  - encoder.default.enc_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Encoding distribution.
  - decoder.default._target_ (multiviewae.architectures.mlp.VariationalDecoder): Type of decoder class to use.
  - decoder.default.init_logvar (int, float): Initial value for log variance of decoder.
  - decoder.default.dec_dist._target_ (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal): Decoding distribution.
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Shi, Y., Siddharth, N., Paige, B., & Torr, P.H. (2019). Variational Mixture-of-Experts Autoencoders for Multi-Modal Deep Generative Models. ArXiv, abs/1911.03393.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of encoding distributions for shared (qu_xs) and private (qw_xs) latent dimensions during training, otherwise return samples from encoding distributions.

Return type
(list)

encode_subset(*x*, *subset*)

Forward pass through encoder networks for a subset of modalities. For modalities not in subset, shared latents are sampled from a random modality and private latents from the shared prior.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **subset** (*list*) – list of modalities to encode.

Returns
list of samples from encoding distributions.

Return type
(list)

decode(*zss*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters

- **zss** (*list*) – list of latent samples if not training or list containing qw_xs (list of private encoding distributions) and
- **qu_xs** (*list of shared encoding distributions*) –

Returns
A nested list of decoding distributions. The outer list has a n_view element indicating latent dimensions index. The inner list is a n_view element list with the position in the list indicating the decoder index.

Return type
(list)

decode_subset(*zss*, *subset*)

Forward pass through decoder networks for a subset of modalities. Each latent is passed through its own decoder.

Parameters

- **zss** (*list*) – list of latent samples for each modality.
- **subset** (*list*) – list of modalities to decode.

Returns
A list of decoding distributions for each modality in subset.

Return type
(list)

`forward`(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing encoding (qw_xs and qu_xs) and decoding (px_zs) distributions.

Return type
fwd_rtn (dict)

`loss_function`(*x*, *fwd_rtn*)

Wrapper function for mmVAE loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
dictionary containing mmVAE loss.

Return type
losses (dict)

`moe_iwae`(*x*, *qw_xs*, *qu_xs*, *px_zs*)

Calculate Mixture-of-Experts importance weighted autoencoder (IWAE) loss used for the mmVAE model.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing encoding and decoding distributions.

Returns
the output tensor.

Return type
(torch.Tensor)

`log_mean_exp`(*value*, *dim=0*, *keepdim=False*)

Returns the log of the mean of the exponentials along the given dimension (dim).

Parameters

- **value** (*torch.Tensor*) – the input tensor.
- **dim** (*int, optional*) – the dimension along which to take the mean.
- **keepdim** (*bool, optional*) – whether the output tensor has dim retained or not.

Returns
the output tensor.

Return type
(torch.Tensor)

class multiviewae.models.**DCCAE**(*cfg=None*, *input_dim=None*, *z_dim=None*)

Deep Canonically Correlated Autoencoder (DCCAE). CCA implementation adapted from: https://github.com/jameschapman19/cca_zoo

Parameters

- **cfg** (*str*) – Path to configuration file. Model specific parameters in addition to default parameters:
  - model._lambda (int, float): Reconstruction weighting term
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

References

Wang, Weiran & Arora, Raman & Livescu, Karen & Bilmes, Jeff. (2016). On Deep Multi-View Representation Learning: Objectives and Optimization.

encode(*x*)

Forward pass through encoder networks.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
list of latent dimensions for each view of type torch.Tensor.

Return type
z (list)

decode(*z*)

Forward pass through decoder networks. Each latent is passed through all of the decoders.

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
list of data reconstructions.

Return type
x_recon (list)

cca(*zs*)

CCA loss calculation. Adapted from: https://github.com/jameschapman19/cca_zoo/blob/main/cca_zoo/deep/_discriminative/_dmcca.py

Parameters
z (*list*) – list of latent dimensions for each view of type torch.Tensor.

Returns
CCA loss.

Return type
cca_loss (torch.Tensor)

forward(*x*)

Apply encode and decode methods to input data to generate latent dimensions and data reconstructions.

Parameters
x (*list*) – list of input data of type torch.Tensor.

Returns
dictionary containing list of data reconstructions (x_recon) and latent dimensions (z).

Return type
fwd_rtn (dict)

`loss_function`(*x*, *fwd_rtn*)

Calculate reconstruction loss.

Parameters

- **x** (*list*) – list of input data of type torch.Tensor.
- **fwd_rtn** (*dict*) – dictionary containing list of data reconstructions (x_recon) and latent dimensions (z).

Returns
dictionary containing reconstruction loss.

Return type
losses (dict)

1.8 Architectures

`class multiviewae.architectures.Encoder`(*input_dim*, *z_dim*, *non_linear*, *bias*, *enc_dist*, ***kwargs*)

Configurable convolutional encoder.

Parameters

- **input_dim** (*list*) – Dimensionality of the input data.
- **num_filters** (*list[int]*) – Number of filters for each convolutional layer.
- **input_shape** (*list[int]*) – Input shape to first conv layer.
- **kernel_size** (*list[int]*) – kernel_size
- **stride** (*list[int]*) –
- **padding** (*list[int]*) –
- **padding_mode** (*str*) –

Initialize internal Module state, shared by both nn.Module and ScriptModule.

`forward`(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

`class multiviewae.architectures.VariationalEncoder`(*input_dim*, *z_dim*, *non_linear*, *bias*, *sparse*, *log_alpha*, *enc_dist*, ***kwargs*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class multiviewae.architectures.**ConditionalVariationalEncoder**(*input_dim*, *z_dim*, *hidden_layer_dim*, *non_linear*, *bias*, *sparse*, *log_alpha*, *enc_dist*, *num_cat*, *one_hot*, *multiple_latents=False*, *u_dim=None*, *w_dim=None*)

MLP Variational Conditional Encoder

Parameters

- **y** (*list*) –
- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.
- **hidden_layer_dim** (*list*) – Number of nodes per hidden layer.
- **non_linear** (*bool*) – Whether to include a ReLU() function between layers.
- **bias** (*bool*) – Whether to include a bias term in hidden layers.
- **sparse** (*bool*) – Whether to enforce sparsity of the encoding distribution.
- **log_alpha** (*float*) – Log of the dropout parameter.
- **enc_dist** (`multiviewae.base.distributions.Normal`, `multiviewae.base.distributions.MultivariateNormal`) – Encoder distribution.
- **num_cat** (*int*) – Number of categories of the labels.
- **one_hot** (*bool*) – Whether to one-hot encode the labels.
- **multiple_latents** (*bool, optional*) – Whether the model using a separate linear layers for shared and private latent spaces.
- **u_dim** (*int, optional*) – Dimensionality of the shared latent space.
- **w_dim** (*int, optional*) – Dimensionality of the private latent space.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class multiviewae.architectures.**Decoder**(*input_dim*, *z_dim*, *non_linear*, *bias*, *dec_dist*, *init_logvar=None*, ***kwargs*)

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*z*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class multiviewae.architectures.**VariationalDecoder**(*input_dim*, *z_dim*, *hidden_layer_dim*, *bias*, *non_linear*, *init_logvar*, *dec_dist*)

MLP Variational Decoder

Parameters

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.
- **hidden_layer_dim** (*list*) – Number of nodes per hidden layer. The layer order is reversed e.g. [100, 50, 5] becomes [5, 50, 100].
- **non_linear** (*bool*) – Whether to include a ReLU() function between layers.
- **bias** (*bool*) – Whether to include a bias term in hidden layers.
- **init_logvar** (*int, float*) – Initial value for log variance of decoder.
- **dec_dist** (`multiviewae.base.distributions.Normal`, `multiviewae.base.distributions.MultivariateNormal`) – Decoder distribution.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*z*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

---

class multiviewae.architectures.**ConditionalVariationalDecoder**(*input_dim*, *z_dim*, *hidden_layer_dim*, *bias*, *non_linear*, *init_logvar*, *dec_dist*, *num_cat*, *one_hot*)

MLP Conditinal Variational Decoder

Parameters

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of latent dimensions.

- **hidden_layer_dim** (*list*) – Number of nodes per hidden layer. The layer order is reversed e.g. [100, 50, 5] becomes [5, 50, 100].
- **non_linear** (*bool*) – Whether to include a ReLU() function between layers.
- **bias** (*bool*) – Whether to include a bias term in hidden layers.
- **init_logvar** (*int, float*) – Initial value for log variance of decoder.
- **dec_dist** (multiviewae.base.distributions.Normal, multiviewae.base.distributions.MultivariateNormal) – Decoder distribution.
- **num_cat** (*int*) – Number of categories of the labels.
- **one_hot** (*bool*) – Whether to one-hot encode the labels.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*z*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

class multiviewae.architectures.**Discriminator**(*input_dim*, *output_dim*, *hidden_layer_dim*, *non_linear*, *bias*, *dropout_threshold*, *is_wasserstein*)

MLP Discriminator

Parameters

- **input_dim** (*list*) – Dimensionality of the input data.
- **z_dim** (*int*) – Number of output dimensions.
- **hidden_layer_dim** (*list*) – Number of nodes per hidden layer.
- **non_linear** (*bool*) – Whether to include a ReLU() function between layers.
- **bias** (*bool*) – Whether to include a bias term in hidden layers.
- **dropout_threshold** (*float*) – Dropout threshold of layers.
- **is_wasserstein** (*bool*) – Whether model employs a wasserstein loss.

Initialize internal Module state, shared by both nn.Module and ScriptModule.

forward(*x*)

Define the computation performed at every call.

Should be overridden by all subclasses.

Note: Although the recipe for forward pass needs to be defined within this function, one should call the `Module` instance afterwards instead of this since the former takes care of running the registered hooks while the latter silently ignores them.

1.9 Parameters and Configurables

1.9.1 General and Default

model

- save_model : False
- seed_everything : True
- seed : 42
- z_dim : 5
- learning_rate : 0.001
- sparse : False
- threshold : 0
- return_mean : True

datamodule

- _target_ : multiviewae.base.dataloaders.MultiviewDataModule
- batch_size : None
- is_validate : True
- train_size : 0.9
- dataset : {'_target_': 'multiviewae.base.datasets.MVDataset'}

encoder

- default
  - _target_ : multiviewae.architectures.mlp.Encoder
  - hidden_layer_dim : []
  - bias : True
  - non_linear : False
  - enc_dist
    - _target_ : multiviewae.base.distributions.Default

decoder

- default
  - _target_ : multiviewae.architectures.mlp.Decoder
  - hidden_layer_dim : []
  - bias : True
  - non_linear : False
  - dec_dist
    - _target_ : multiviewae.base.distributions.Default

prior

- _target_ : multiviewae.base.distributions.Normal
- loc : 0.0
- scale : 1.0

trainer

- _target_ : pytorch_lightning.Trainer
- accelerator : auto
- max_epochs : 10
- deterministic : False
- log_every_n_steps : 2

optimizer

- _target_ : torch.optim.Adam

callbacks

- model_checkpoint
  - _target_ : pytorch_lightning.callbacks.ModelCheckpoint
  - monitor : val_loss
  - mode : min
  - save_last : True
  - dirpath : ${out_dir}
- early_stopping
  - _target_ : pytorch_lightning.callbacks.EarlyStopping
  - monitor : val_loss
  - mode : min

  - patience : 299
  - min_delta : 0.001
  - verbose : True

logger

- _target_ : pytorch_lightning.loggers.tensorboard.TensorBoardLogger
- save_dir : ${out_dir}/logs

1.9.2 Model-specific

AE

DCCAE

- model
  - _lambda : 1
- optimizer
  - _target_ : torch.optim.LBFGS

DMVAE

- model
  - private : True
  - s_dim : 3
  - beta : 1
- encoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalEncoder
    * non_linear : False
    * multiple_latents : False
    * enc_dist
      · _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalDecoder
    * non_linear : False
    * init_logvar : -3
    * dec_dist
      · _target_ : multiviewae.base.distributions.Normal

DVCCA

- model
  - beta : 1
  - private : False
  - sparse : True
  - threshold : 0
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalDecoder
    - non_linear : False
    - init_logvar : -3
    - dec_dist
      - _target_ : multiviewae.base.distributions.Normal

JMVAE

- model
  - alpha : 1
  - warmup : 0
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalDecoder
    - non_linear : False
    - init_logvar : -3
    - dec_dist

        - _target_ : multiviewae.base.distributions.Normal

mAAE

- discriminator
  - _target_ : multiviewae.architectures.mlp.Discriminator
  - hidden_layer_dim : []
  - bias : True
  - non_linear : False
  - dropout_threshold : 0

mcVAE

- model
  - beta : 1
  - sparse : False
  - threshold : 0
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalDecoder
    - non_linear : False
    - init_logvar : -3
    - dec_dist
      - _target_ : multiviewae.base.distributions.Normal

me_mVAE

- model
  - beta : 1
  - join_type : PoE
  - warmup : 0
  - sparse : False
  - use_prior : False

  - weight_kld : False
  - weight_ll : False
- encoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalEncoder
    * non_linear : False
    * enc_dist
      · _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalDecoder
    * non_linear : False
    * init_logvar : -3
    * dec_dist
      · _target_ : multiviewae.base.distributions.Normal

mmJSD

- model
  - private : True
  - s_dim : 2
  - alpha : 1
  - beta : 1
  - weight_ll : False
- encoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalEncoder
    * non_linear : False
    * enc_dist
      · _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    * _target_ : multiviewae.architectures.mlp.VariationalDecoder
    * non_linear : False
    * init_logvar : -3
    * dec_dist
      · _target_ : multiviewae.base.distributions.Normal

mmVAE

- model
  - K : 1
  - beta : 1
  - DREG_loss : False
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalDecoder
    - non_linear : False
    - init_logvar : -3
    - dec_dist
      - _target_ : multiviewae.base.distributions.Normal

mmVAEPlus

- model
  - K : 1
  - beta : 1
  - u_dim : 3
  - w_dim : 2
  - z_dim : 5
  - learn_private_prior : True
  - learn_shared_prior : False
  - multiple_latents : True
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder

    - default
        * _target_ : multiviewae.architectures.mlp.VariationalDecoder
        * non_linear : False
        * init_logvar : -3
        * dec_dist
            · _target_ : multiviewae.base.distributions.Normal

MoPoEVAE

- model
    - beta : 1
    - weight_ll : False
- encoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalEncoder
        * non_linear : False
        * enc_dist
            · _target_ : multiviewae.base.distributions.Normal
- decoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalDecoder
        * non_linear : False
        * init_logvar : -3
        * dec_dist
            · _target_ : multiviewae.base.distributions.Normal

mVAE

- model
    - beta : 1
    - join_type : PoE
    - sparse : False
    - threshold : 0.2
    - warmup : 10
    - use_prior : True
    - weight_ll : False
- encoder
    - default

        * _target_ : multiviewae.architectures.mlp.VariationalEncoder
        * non_linear : False
        * enc_dist
            · _target_ : multiviewae.base.distributions.Normal
- decoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalDecoder
        * non_linear : False
        * init_logvar : -3
        * dec_dist
            · _target_ : multiviewae.base.distributions.Normal

mvtCAE

- model
    - beta : 1
    - alpha : 0.5
- encoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalEncoder
        * non_linear : False
        * enc_dist
            · _target_ : multiviewae.base.distributions.Normal
- decoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalDecoder
        * non_linear : False
        * init_logvar : -3
        * dec_dist
            · _target_ : multiviewae.base.distributions.Normal

mWAE

- discriminator
  - _target_ : multiviewae.architectures.mlp.Discriminator
  - hidden_layer_dim : []
  - bias : True
  - non_linear : False
  - dropout_threshold : 0

weighted_DMVAE

- model
  - private : True
  - s_dim : 3
  - beta : 1
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder
    - non_linear : False
    - enc_dist
      - _target_ : multiviewae.base.distributions.Normal
- decoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalDecoder
    - non_linear : False
    - init_logvar : -3
    - dec_dist
      - _target_ : multiviewae.base.distributions.Normal

weighted_mVAE

- model
  - beta : 1
  - _lambda : 1
  - private : False
- encoder
  - default
    - _target_ : multiviewae.architectures.mlp.VariationalEncoder

        * non_linear : False
        * enc_dist
            · _target_ : multiviewae.base.distributions.Normal
- decoder
    - default
        * _target_ : multiviewae.architectures.mlp.VariationalDecoder
        * non_linear : False
        * init_logvar : -3
        * dec_dist
            · _target_ : multiviewae.base.distributions.Normal

1.10 Datamodules

class multiviewae.base.dataloaders.**MultiviewDataModule**(**args: Any, **kwargs: Any*)

LightningDataModule for multi-view data.

Parameters

- **n_views** (*int*) – Number of views in the data.
- **batch_size** (*int*) – Batch size.
- **is_validate** (*bool*) – Whether to use a validation set.
- **train_size** (*float*) – Proportion of batch to use for training between 0 and 1. Remainder of batch is used for validation.
- **data** (*list*) – Input data. list of torch.Tensors.
- **labels** (*np.array*) – Dataset labels.

setup(*stage*)

Called at the beginning of fit (train + validate), validate, test, and predict. This is a good hook when you need to build models dynamically or adjust something about them. This hook is called on every process when using DDP.

Parameters

stage – either `'fit'`, `'validate'`, `'test'`, or `'predict'`

Example:

```
class LitModel(...):
    def __init__(self):
        self.l1 = None

    def prepare_data(self):
        download_data()
        tokenize()

        # don't do this
        self.something = else
```

(continues on next page)

(continued from previous page)

```
    def setup(stage):
        data = Load_data(...)
        self.l1 = nn.Linear(28, data.num_classes)
```

train_dataloader()

Implement one or more PyTorch DataLoaders for training.

Returns
A collection of `torch.utils.data.DataLoader` specifying training samples. In the case of multiple dataloaders, please see this page.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

For data processing use the following pattern:

- download in `prepare_data()`
- process and split in *setup()*

However, the above are only necessary for distributed processing.

> **Warning:** do not assign state in prepare_data

- `fit()`
- ...
- `prepare_data()`
- *setup()*
- *train_dataloader()*

Note: Lightning adds the correct sampler for distributed and arbitrary hardware. There is no need to set it yourself.

Example:

```
# single dataloader
def train_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=True, transform=transform,
                    download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=True
    )
    return loader

# multiple dataloaders, return as list
def train_dataloader(self):
```

(continues on next page)

(continued from previous page)

```
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a list of tensors: [batch_mnist, batch_cifar]
    return [mnist_loader, cifar_loader]

# multiple dataloader, return as dict
def train_dataloader(self):
    mnist = MNIST(...)
    cifar = CIFAR(...)
    mnist_loader = torch.utils.data.DataLoader(
        dataset=mnist, batch_size=self.batch_size, shuffle=True
    )
    cifar_loader = torch.utils.data.DataLoader(
        dataset=cifar, batch_size=self.batch_size, shuffle=True
    )
    # each batch will be a dict of tensors: {'mnist': batch_mnist, 'cifar': batch_
↪cifar}
    return {'mnist': mnist_loader, 'cifar': cifar_loader}
```

val_dataloader()

Implement one or multiple PyTorch DataLoaders for validation.

The dataloader you return will not be reloaded unless you set **:paramref:`~pytorch_lightning.trainer.Trainer.reload_dataloaders_every_n_epochs`** to a positive integer.

It's recommended that all data downloads and preparation happen in `prepare_data()`.

- `fit()`
- ...
- `prepare_data()`
- *train_dataloader()*
- *val_dataloader()*
- `test_dataloader()`

Note: Lightning adds the correct sampler for distributed and arbitrary hardware There is no need to set it yourself.

Returns
A `torch.utils.data.DataLoader` or a sequence of them specifying validation samples.

Examples:

```
def val_dataloader(self):
    transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize((0.5,), (1.0,))])
    dataset = MNIST(root='/path/to/mnist/', train=False,
                    transform=transform, download=True)
    loader = torch.utils.data.DataLoader(
        dataset=dataset,
        batch_size=self.batch_size,
        shuffle=False
    )

    return loader

# can also return multiple dataloaders
def val_dataloader(self):
    return [loader_a, loader_b, ..., loader_n]
```

Note: If you don't need a validation dataset and a `validation_step()`, you don't need to implement this method.

Note: In the case where you return multiple validation dataloaders, the `validation_step()` will have an argument `dataloader_idx` which matches the order here.

class multiviewae.base.dataloaders.**IndexDataModule**(**args: Any, **kwargs: Any*)

LightningDataModule for multi-view data.

Parameters

- **n_views** (*int*) – Number of views in the data.
- **batch_size** (*int*) – Batch size.
- **is_validate** (*bool*) – Whether to use a validation set.
- **train_size** (*float*) – Proportion of batch to use for training between 0 and 1. Remainder of batch is used for validation.
- **data** (*list*) – Input data. list of identifiers to load data from.
- **labels** (*np.array*) – Dataset labels.

1.11 Datasets

Class for loading data into Pytorch float tensor

From: https://gitlab.com/acasamitjana/latentmodels_ad

class multiviewae.base.datasets.**MVDataset**(*data, n_views, labels=None, return_index=False, transform=None*)

PyTorch Dataset for storing and accessing multi-view data.

Parameters

- **data** (*list*) – Input data. list of torch.Tensors.

- **labels** (*np.array*) – Dataset labels.
- **return_index** (*bool*) – Whether to return batch index labels.
- **transform** (*torchvision.transforms*) – Torchvision transformation to apply to the data. Default is None.

class multiviewae.base.datasets.**IndexMVDataset**(*data*, *n_views*, *labels=None*, *filename=''*, *data_dir=''*)

1.12 Distributions

class multiviewae.base.distributions.**Default**(***kwargs*)

Artificial distribution designed for data with unspecified distribution. Used so that log_likelihood and _sample methods can be called by model class. :param x: List of input data. :type x: list

log_likelihood(*x*)

calculates the mean squared error between input data and reconstruction.

Parameters
x (*torch.Tensor*) – data reconstruction.

Returns
Negative mean squared error.

Return type
torch.Tensor

class multiviewae.base.distributions.**Categorical**(***kwargs*)

Artificial distribution designed for categorical data. Used so that log_likelihood and _sample methods can be called by model class. :param x: List of input data. :type x: list

log_likelihood(*x*, *eps=1e-06*)

calculates the k-class cross entropy between input data and reconstruction. :param x: data reconstruction. :type x: torch.Tensor

Returns
Negative k-class cross entropy.

Return type
torch.Tensor

class multiviewae.base.distributions.**Normal**(***kwargs*)

Univariate normal distribution. Inherits from torch.distributions.Normal.

Parameters

- **loc** (*int, torch.Tensor*) – Mean of distribution.
- **scale** (*int, torch.Tensor*) – Standard deviation of distribution.

property variance

Returns the variance of the distribution.

sparse_kl_divergence()

Implementation from: https://github.com/senya-ashukha/variational-dropout-sparsifies-dnn/blob/master/KL%20approximation.ipynb

class multiviewae.base.distributions.**MultivariateNormal**(***kwargs*)

Multivariate normal distribution with diagonal covariance matrix. Inherits from torch.distributions.multivariate_normal.MultivariateNormal.

Parameters

- **loc** (*list, torch.Tensor*) – Mean of distribution.
- **scale** (*int, torch.Tensor*) – Standard deviation of distribution.

property variance

Returns the variance of the distribution.

class multiviewae.base.distributions.**Bernoulli**(***kwargs*)

Bernoulli distribution. Inherits from torch.distributions.Bernoulli. :param x: List of input data. :type x: list

rsample()

Generates a sample_shape shaped reparameterized sample or sample_shape shaped batch of reparameterized samples if the distribution parameters are batched.

class multiviewae.base.distributions.**ApproxBernoulli**(***kwargs*)

Artificial distribution designed for (approximately) Bernoulli distributed data. The data isn't restricted to bernoulli distribution, this class is designed as a wrapper for the log_likelihood() method which is required for the multiview models.

Parameters

x (*list*) – List of input data.

class multiviewae.base.distributions.**Laplace**(***kwargs*)

Laplace distribution. Inherits from torch.distributions.Laplace.

Parameters

- **loc** (*list, torch.Tensor*) – Mean of distribution.
- **scale** (*int, torch.Tensor*) – Standard deviation of distribution.

1.13 How to contribute

Contributions to the `multi-view-AE` library are welcome and greatly appreciated! Here are some ways you can contribute:

1.13.1 Reporting a bug

Please report bugs by submitting an issue at https://github.com/alawryaguila/multi-view-AE/issues

If you are reporting a bug, please include the following information:

- A quick summary and/or background.
- Your operating system name and version.
- Details about your local setup that might be helpful in troubleshooting e.g. python version, library versions
- Detailed steps to reproduce the bug.
- What you expected to happen.
- What actually happens.

1.13.2 Proposing a new feature

The best way to propose a new feature is by submitting an issue https://github.com/alawryaguila/multi-view-AE/issues

To propose a feature please include:

- Describe in detail how the new feature would work.
- Explain the use case of the new feature.
- Please keep the scope as narrow and specific as possible, to make it easier to implement.

1.13.3 Writing documentation

The latest documentation for the `multi-view-AE` library is available at https://multi-view-ae.readthedocs.io/en/latest/

If any documentation is unclear or requires correction, please submit an issue.

1.13.4 Submitting changes

To submit your code when fixing bugs, documentation, or implementing new features, please follow the steps below.

1. Fork the `multi-view-AE` repository on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/multi-view-AE.git
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature

Make your desired changes on your local branch.
```

4. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

5. Submit a pull request through GitHub.

1.14 Support

If you are having any issues with the code or the `multi-view-AE` library in general, please don't hesitate to reach out at: ana.aguila.18@ucl.ac.uk

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search

PYTHON MODULE INDEX

m

INDEX

E

F

G

I

J

L

M

N

R

S

T

V

W