Package 'autotab'

May 7, 2026

Title Variational Autoencoders for Heterogeneous Tabular Data

Version 1.0

Description Build and train a variational autoencoder (VAE) for mixed-type tabular data (continuous, binary, categorical).
Models are implemented using 'TensorFlow' and 'Keras' via the 'reticulate' interface, enabling reproducible VAE training for heterogeneous tabular datasets.

License MIT + file LICENSE

URL https://github.com/SarahMilligan-hub/AutoTab

BugReports https://github.com/SarahMilligan-hub/AutoTab/issues

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1)

Imports keras, magrittr, R6, reticulate, tensorflow

Suggests caret

SystemRequirements Python (>= 3.8); TensorFlow (>= 2.10); Keras; TensorFlow Addons

NeedsCompilation no

Author Sarah Milligan [aut, cre]

Maintainer Sarah Milligan <slm1999@bu.edu>

Repository CRAN

Date/Publication 2026-03-25 19:10:02 UTC

Contents

`decoder_model` *Builds the decoder graph for an AutoTab VAE*

Description

Reconstructs the **decoder** computational graph used during training. This is used internally by `VAE_train()` and externally when you want to load the trained decoder weights and generate new samples by sampling the latent space.

Usage

```
decoder_model(
  decoder_input,
  decoder_info,
  latent_dim,
  feat_dist,
  lip_dec,
  pi_dec,
  max_std = 10,
  min_val = 0.001,
  temperature = 0.5
)
```

Arguments

`decoder_input`	Ignored; pass `NULL`. No input is needed when building the compitational graph.
`decoder_info`	List defining the decoder architecture, e.g. `list(list("dense", 80, "relu"), list("dropout", 0.1), list("dense", 100, "relu"))`. Each `dense` entry is `list("dense", units, activation)`. Each `dropout` entry is `list("dropout", rate)`. Optional elements: `[[4]]` L2 flag (0/1), `[[5]]` L2 value, `[[6]]` BN flag (FALSE/TRUE), `[[7]]` BN momentum, `[[8]]` BN scale/center (TRUE/FALSE).
`latent_dim`	Integer. Latent dimension used during training.
`feat_dist`	Data frame with columns `column_name`, `distribution`, `num_params` (created by `extracting_distribution()` and set via `set_feat_dist()`).
`lip_dec`	0/1 (logical). Use spectral normalization on dense hidden layers.

pi_dec	Integer. Power-iteration count for spectral normalization.
max_std	Numeric. Upper bound for Gaussian SD heads (default 10.0).
min_val	Numeric. Lower bound (epsilon) for Gaussian SD heads (default 1e-3).
temperature	Numeric. Gumbel–Softmax temperature for categorical heads (default 0.5).

Details

The final output layer of an AutoTab decoder slices outputs by feature distribution in feat_dist: Gaussian heads output mean/SD (with min_val/max_std constraints), Bernoulli heads output logits passed through sigmoid to extract probabilities, and Categorical heads use Gumbel–Softmax with the given temperature.

If lip_dec = 1, dense hidden layers are wrapped with #' spectral normalization using pi_dec power iterations.

Value

A compiled **Keras model** representing the decoder computational graph. You can load trained decoder weights with Decoder_weights() + set_weights(), then call predict(decoder, Z) where Z is an n x latent_dim matrix (typically a sample from your latent space).

See Also

VAE_train(), Decoder_weights(), encoder_latent(), Latent_sample(), extracting_distribution()

Examples

```
if (reticulate::py_module_available("tensorflow") &&
    exists("training") &&
    exists("feat_dist")) {

  # Assume you already have feat_dist set via set_feat_dist(feat_dist)
  decoder_info <- list(
    list("dense", 80, "relu"),
    list("dense", 100, "relu")
  )

  # Rebuild and apply decoder
  weights_decoder <- Decoder_weights(
    encoder_layers = 2,
    trained_model  = training$trained_model,
    lip_enc        = 0,
    pi_enc         = 0,
    prior_learn    = "fixed",
    BNenc_layers   = 0,
    learn_BN       = 0
  )

  decoder <- decoder_model(
    decoder_input = NULL,
    decoder_info  = decoder_info,
```

```
    latent_dim    = 5,
    feat_dist     = feat_dist,
    lip_dec       = 0,
    pi_dec        = 0
  )

  decoder %>% keras::set_weights(weights_decoder)
}
```

`Decoder_weights` *Extract decoder-only weights from a trained Keras model*

Description

Pulls just the **decoder** weights from `keras::get_weights(trained_model)`, skipping encoder parameters and (if used) the final trainable tensors from a learnable mixture-of-Gaussians (MoG) prior (means, log_vars, and weight logits).

Usage

```
Decoder_weights(
  encoder_layers,
  trained_model,
  lip_enc,
  pi_enc,
  prior_learn,
  BNenc_layers,
  learn_BN
)
```

Arguments

`encoder_layers`	Integer. Number of encoder layers (used to compute split index).
`trained_model`	Keras model. Typically `training$trained_model`.
`lip_enc`	Integer (0/1). Whether spectral normalization was used in the encoder.
`pi_enc`	Integer. Power iterations used in encoder spectral normalization.
`prior_learn`	Character. `"fixed"` for fixed prior; any other value implies learnable MoG.
`BNenc_layers`	Integer. Number of encoder BN layers (affects split index).
`learn_BN`	Integer (0/1). Whether BN layers learned scale and center.

Details

- When `prior_learn != "fixed"`, the final **three** tensors are assumed to belong to the learnable MoG prior (`mog_means`, `mog_log_vars`, `mog_weights_logit`) and are excluded.
- The split index math mirrors `Encoder_weights()` and assumes the standard AutoTab graph wiring.
- All model weights can always be accessed directly using `keras::get_weights(trained_model)`. This function is provided as a convenience tool within AutoTab to streamline decoder reconstruction but is not the only method available.

Value

A `list()` of decoder weight tensors in order, suitable for `set_weights()`.

See Also

`decoder_model()`, `Encoder_weights()`, `VAE_train()`

Examples

```
decoder_info <- list(
  list("dense", 80, "relu"),
  list("dense", 100, "relu")
)

if (reticulate::py_module_available("tensorflow") &&
    exists("training")) {
weights_decoder <- Decoder_weights(
  encoder_layers = 2,
  trained_model  = training$trained_model,  #where training = VAE_train(...)
  lip_enc        = 0,
  pi_enc         = 0,
  prior_learn    = "fixed",
  BNenc_layers   = 0,
  learn_BN       = 0
)
}
```

---

encoder_decoder_information

Specifying Encoder and Decoder Architectures for `VAE_train()`

Description

Specifying Encoder and Decoder Architectures for `VAE_train()`

Encoder and Decoder configuration

The arguments `encoder_info` and `decoder_info` define the architecture of the encoder and decoder networks used in `VAE_train()`. Each is a list in which every element describes one layer in sequence.

AutoTab currently supports two layer types: `"dense"` and `"dropout"`.

Dense layers

When `input1 = "dense"`, the layer specification takes the form:

- `input2`: *Numeric.* Number of units (nodes).
- `input3`: *Character.* Activation function (any TensorFlow/Keras activation name).
- `input4`: *Integer (0/1).* L2 regularization flag. Default: `0`.
- `input5`: *Numeric.* L2 regularization strength (`lambda`). Default: `1e-4`.
- `input6`: *Logical.* Apply batch normalization. Default: `FALSE`.
- `input7`: *Numeric.* Batch normalization momentum. Default: `0.99`.
- `input8`: *Logical.* Whether batch normalization scale and center parameters are trainable. Default: `TRUE`.

Dropout layers

When `input1 = "dropout"`, the layer specification is:

- `input2`: *Numeric.* Dropout rate.

Together, these lists fully specify the encoder and decoder architectures used during VAE training.

See Also

`VAE_train()`

---

encoder_latent *Rebuild the encoder graph to export z_mean and z_log_var*

Description

Constructs the encoder computation graph (matching your original `encoder_info`) so that weights extracted by `Encoder_weights()` can be applied and the encoder to produce `z_mean` and `z_log_var`.

Usage

```
encoder_latent(
  encoder_input,
  encoder_info,
  latent_dim,
  Lip_en,
  power_iterations
)
```

Arguments

`encoder_input`	Data frame or matrix of the **preprocessed** variables (used for shape only).
`encoder_info`	List defining encoder architecture.
`latent_dim`	Integer. Latent dimension.
`Lip_en`	Integer (0/1). Whether spectral normalization was used in the encoder.
`power_iterations`	Integer. Power iterations for spectral normalization (if used).

Details

- Spectral normalization is sourced from TensorFlow Addons via `get_tfa()`.
- `encoder_input` provides shape; the data are not consumed at build time.
- Apply weights with `set_weights()` using the output of `Encoder_weights()`.

Value

A Keras model whose outputs are `list(z_mean, z_log_var)`.

See Also

`Encoder_weights()`, `Latent_sample()`, `Decoder_weights()`

Examples

```
encoder_info <- list(
  list("dense", 100, "relu"),
  list("dense",  80, "relu")
)

if (reticulate::py_module_available("tensorflow") &&
    exists("training")) {
weights_encoder <- Encoder_weights(
  encoder_layers = 2,
  trained_model  = training$trained_model,  #where training = VAE_train(...)
  lip_enc        = 0,
  pi_enc         = 0,
  BNenc_layers   = 0,
  learn_BN       = 0
)

latent_encoder <- encoder_latent(
  encoder_input    = data,
  encoder_info     = encoder_info,
  latent_dim       = 5,
  Lip_en           = 0,
  power_iterations = 0
)
latent_encoder %>% keras::set_weights(weights_encoder)
}
```

Encoder_weights *Extract encoder-only weights from a trained Keras model*

Description

Pulls just the **encoder** weights from `keras::get_weights(trained_model)`, skipping any parameters introduced by batch normalization (BN) or spectral normalization (SN). The split index is computed from the number of encoder layers and whether BN/SN were used.

Usage

```
Encoder_weights(
  encoder_layers,
  trained_model,
  lip_enc,
  pi_enc,
  BNenc_layers,
  learn_BN
)
```

Arguments

`encoder_layers`	Integer. Number of encoder layers (used to compute split index).
`trained_model`	Keras model. Typically `training$trained_model` from `VAE_train()`.
`lip_enc`	Integer (0/1). Whether spectral normalization was used in the encoder.
`pi_enc`	Integer. Power iteration count if spectral normalization was used.
`BNenc_layers`	Integer. Number of encoder layers that had batch normalization.
`learn_BN`	Integer (0/1). Whether BN layers learned scale and center.

Details

- The index arithmetic assumes AutoTab's standard Dense/BN/SN layout. If you substantially change layer ordering or introduce new per-layer parameters, re-check the split index.
- All model weights can always be accessed directly using `keras::get_weights(trained_model)`. This function is provided as a convenience tool within AutoTab to streamline encoder reconstruction but is not the only method available.

Value

A `list()` of encoder weight tensors in order, suitable for `set_weights()`.

See Also

`encoder_latent()`, `Decoder_weights()`, `VAE_train()`, `Latent_sample()`

Examples

```
encoder_info <- list(
  list("dense", 100, "relu"),
  list("dense",  80, "relu")
)


if (reticulate::py_module_available("tensorflow") &&
    exists("training")) {
weights_encoder <- Encoder_weights(
  encoder_layers = 2,
  trained_model  = training$trained_model, #where training = VAE_train(...)
  lip_enc        = 0,
  pi_enc         = 0,
  BNenc_layers   = 0,
  learn_BN       = 0
)
}
```

`extracting_distribution` *Build the* `feat_dist` *data frame for AutoTab*

Description

Creates one row per original variable with columns:

- `column_name`: variable name
- `distribution`: one of `"gaussian"`, `"bernoulli"`, or `"categorical"`
- `num_params`: number of decoder outputs the VAE should produce for that variable

Usage

```
extracting_distribution(data)
```

Arguments

`data`	Data frame of the **original (not preprocessed)** variables.

Details

A variable is classified as:

- **bernoulli** if it has exactly 2 unique values (any type)
- **categorical** if it is a character/factor with more than 2 unique values
- **gaussian** otherwise (e.g., numeric with >2 distinct values)

AutoTab is not built to handle missing data. A message will prompt the user if the data has NA values.

In AutoTab, the decoder outputs **distribution-specific parameters** for each variable, not reconstructed values directly. Therefore:

- **Continuous (Gaussian)** variables output **two parameters** per feature: the mean (μ) and the standard deviation (σ).
- **Binary (Bernoulli)** variables output **one parameter**: the probability (p) of observing a 1.
- **Categorical** variables output **one parameter per category level**: the probabilities corresponding to each possible class.

As a result, the **decoder output matrix** will typically have **more columns** than the original training data.

For example, if your original dataset has:

```
1 continuous variable   →  2 decoder parameters
1 binary variable       →  1 decoder parameter
1 categorical variable with 3 levels → 3 decoder parameters
```

The total number of decoder outputs will be **2 + 1 + 3 = 6**, even though the input data has only 3 original variables.

AutoTab keeps track of this mapping internally through the `feat_dist` object, ensuring that the reconstruction loss and sampling functions correctly handle each distributional head.

Value

A data frame with columns `column_name`, `distribution`, and `num_params`. Note: refer to `feat_reorder()`.

See Also

`feat_reorder()`, `set_feat_dist()`

Examples

```
data_example <- data.frame(
  cont = rnorm(5),
  bin  = c(0,1,0,1,1),
  cat  = factor(c("A","B","C","A","C"))
)

feat_dist <- extracting_distribution(data_example)
print(feat_dist)
# column_name distribution num_params
# 1        cont      gaussian          2
# 2         bin     bernoulli          1
# 3         cat    categorical          3

# The decoder will therefore output 6 total columns (2+1+3)
```

feat_reorder *Reorder* feat_dist *rows to match preprocessed data*

Description

Ensures row order in feat_dist matches the **column prefix order** in the preprocessed (dummy-coded) training data. This assumes dummy columns are named as <original_name>_<level> and therefore start with the original variable name.

Usage

```
feat_reorder(feat_dist, data)
```

Arguments

feat_dist	Data frame from extracting_distribution().
data	Data frame of the **original (preprocessed)** variables.

Value

The input feat_dist, reordered to align with data.

See Also

extracting_distribution(), set_feat_dist()

Examples

```
# Small toy dataset
data_example <- data.frame(
  cont = rnorm(5),
  bin  = c(0, 1, 0, 1, 1),
  cat  = factor(c("A", "B", "C", "A", "C"))
)

# Extract feature distributions in original column order
feat_dist <- extracting_distribution(data_example)

# Suppose preprocessing (e.g., dummy coding) reordered the columns
data_reordered <- data_example[, c("cat", "cont", "bin")]

# Reorder feat_dist rows to match the preprocessed data columns
feat_dist_reordered <- feat_reorder(feat_dist, data_reordered)
feat_dist_reordered
```

get_feat_dist *Get the stored feature distribution*

Description

Retrieves the `feat_dist` object previously stored by `set_feat_dist()`. Throws an error if it has not been set.

Usage

```
get_feat_dist()
```

Value

A data.frame containing feature distribution metadata.

Latent_sample *Sample from the latent space*

Description

Draws a stochastic sample from the latent space of a trained VAE given the mean (`z_mean`) and log-variance (`z_log_var`) outputs of the encoder. This operation implements the **reparameterization trick**:

$$z = \mu + \sigma \odot \epsilon$$

where $\epsilon \sim \mathcal{N}(0, I)$.

Usage

```
Latent_sample(z_mean, z_log_var)
```

Arguments

`z_mean`	TensorFlow tensor or R matrix. The mean values of the latent space.
`z_log_var`	TensorFlow tensor or R matrix. The log-variances of the latent space.

Details

The function is used internally within `VAE_train()` but can also be called directly to sample latent points and decode synthetic output. Typically, `z_mean` and `z_log_var` are obtained via `encoder_latent()` and the corresponding weights extracted using `Encoder_weights()`.

- The log-variance (`z_log_var`) is clamped between -10 and 10 to prevent numerical overflow or vanishing variance during training.
- The standard deviation is lower-bounded by `1e-3` for stability.

This function returns a TensorFlow tensor representing the sampled latent points. Use `as.matrix()` or `as.data.frame()` to convert to an R matrix or data frame before passing to `decoder_model()` or other R functions.

Value

A TensorFlow tensor of latent samples with the same shape as `z_mean`.

See Also

`VAE_train()`, `encoder_latent()`, `Encoder_weights()`, `decoder_model()`

Examples

```
# Suppose encoder_latent() returns z_mean and z_log_var
z_mean    <- matrix(rnorm(10), ncol = 5)
z_log_var <- matrix(rnorm(10), ncol = 5)


if (reticulate::py_module_available("tensorflow")) {
  # Sample from latent space
  z_sample <- Latent_sample(z_mean, z_log_var)

  # Convert to R matrix for decoder prediction
  z_mat <- as.matrix(z_sample)

  # Suppose the computational graph was rebuilt using `decoder_model()`
  # and assigned to an object named `decoder`:
  # decoder_output <- predict(decoder, z_mat)
}
```

---

min_max_scale *Min–max scale continuous variables*

Description

Scales numeric vectors to the [0, 1] range using the formula:

$$(x - \min(x))/(\max(x) - \min(x))$$

Usage

```
min_max_scale(x)
```

Arguments

`x`	Numeric vector. Continuous variable(s) to scale.

Details

This is the recommended preprocessing step for continuous variables prior to VAE training with AutoTab, ensuring all inputs are on comparable scales to binary and categorical features.

- The transformation is **performed column-wise** when applied to data frames.

Value

Numeric vector of the same length as `x`, scaled to [0, 1].

See Also

`extracting_distribution()`, `set_feat_dist()`, `VAE_train()`

Examples

```
x <- c(10, 20, 30)
min_max_scale(x)

# Apply to multiple columns
data <- data.frame(age = c(20, 40, 60), income = c(3000, 5000, 7000))
Continuous_MinMaxScaled = as.data.frame(lapply(data, min_max_scale))
```

---

mog_prior *Mixture-of-Gaussians (MoG) prior in AutoTab*

Description

AutoTab allows the encoder prior to be either a single Gaussian (`prior = "single_gaussian"`) or a mixture of Gaussians (`prior = "mixture_gaussian"`). When using a MoG prior, the user may optionally specify the component means, variances, and mixture weights. The user may also indicate if the means, variances, and mixture weights can be learned or not using learnable_mog with a logical TRUE/FALSE.

Details

If `prior = "single_gaussian"`, the prior is a standard Normal in the latent space and the MoG-related arguments (`K`, `mog_means`, `mog_log_vars`, `mog_weights`, `learnable_mog`) are ignored.

When `prior = "mixture_gaussian"`:

- If `learnable_mog = FALSE`, then `mog_means`, `mog_log_vars`, and `mog_weights` **must** be supplied and are treated as fixed.
- If `learnable_mog = TRUE`, any of `mog_means`, `mog_log_vars`, or `mog_weights` that are provided are used as initial values and are updated during training. If they are omitted, AutoTab initializes them internally (e.g., Normal or zero-centered initializations).

Prior options in `VAE_train()`

- `prior`: character, one of `"single_gaussian"` or `"mixture_gaussian"`.
- `K`: integer, number of mixture components when `prior = "mixture_gaussian"`.
- `learnable_mog`: logical; if `TRUE`, the MoG parameters (means, log-variances, and mixture weights) are learned during training.
- `mog_means`: optional numeric matrix of size `K x latent_dim`, giving the initial means for each mixture component in the latent space.
- `mog_log_vars`: optional numeric matrix of size `K x latent_dim`, giving initial log-variances for each component.
- `mog_weights`: optional numeric vector of length `K`, giving initial mixture weights that should sum to 1.

Shape of `mog_means`

For a latent dimension `latent_dim` and `K` mixture components, `mog_means` must be a numeric matrix with:

- `nrow(mog_means) == K`
- `ncol(mog_means) == latent_dim`

Each row corresponds to the mean vector of one mixture component in the latent space.

See Also

`VAE_train()`

Examples

```
# Examples of a Mixture-of-Gaussians (MoG) prior in AutoTab

# These examples illustrate:
# 1) learnable_mog = FALSE with fixed MoG parameters
# 2) learnable_mog = TRUE with preset means/variances/weights
# 3) learnable_mog = TRUE with all MoG parameters learned

# Required packages for the full example:
# - AutoTab (this package)
# - keras
# - caret (for dummyVars)


if (requireNamespace("caret", quietly = TRUE) &&
    reticulate::py_module_available("tensorflow")) {

  # --------------------------------
  # Data simulation and preparation
  # --------------------------------
  set.seed(123)
  age        <- rnorm(100, mean = 45, sd = 12)
```

```
income     <- rnorm(100, mean = 60000, sd = 15000)
bmi        <- rnorm(100, mean = 25, sd = 4)
smoker     <- rbinom(100, 1, 0.25)
exercise   <- rbinom(100, 1, 0.6)
diabetic   <- rbinom(100, 1, 0.15)
education  <- sample(
  c("HighSchool", "College", "Graduate"),
  100, replace = TRUE,
  prob = c(0.4, 0.4, 0.2)
)
marital    <- sample(
  c("Single", "Married", "Divorced"),
  100, replace = TRUE
)
occupation <- sample(
  c("Clerical", "Technical", "Professional", "Other"),
  100, replace = TRUE
)

data_final <- data.frame(
  age, income, bmi,
  smoker, exercise, diabetic,
  education, marital, occupation
)

# One-hot encode categorical variables
encoded_data  <- caret::dummyVars(~ education + marital + occupation,
                                  data = data_final)
one_hot_coded <- as.data.frame(predict(encoded_data, newdata = data_final))

data_cont <- subset(data_final, select = c(age, income, bmi))
Continuous_MinMaxScaled <- as.data.frame(
  lapply(data_cont, min_max_scale)  # min_max_scale is an AutoTab function
)
data_bin <- subset(data_final, select = c(smoker, exercise, diabetic))

# Bind all data together
data <- cbind(Continuous_MinMaxScaled, data_bin, one_hot_coded)

# Step 1: Extract and set feature distributions
feat_dist   <- feat_reorder(extracting_distribution(data_final), data)
rownames(feat_dist) <- NULL
set_feat_dist(feat_dist)

# Step 2: Define encoder / decoder architectures and MoG parameters
encoder_info <- list(
  list("dense", 25, "relu"),
  list("dense", 50, "relu")
)

decoder_info <- list(
  list("dense", 50, "relu"),
  list("dense", 25, "relu")
```

```
)

mog_means <- matrix(
  c(rep(-5, 5), rep(0, 5), rep(5, 5)),
  nrow = 3, byrow = TRUE
)
mog_log_vars <- matrix(log(0.5), nrow = 3, ncol = 5)
mog_weights  <- c(0.3, 0.4, 0.3)

# ------------------------------------------------------------
# Example 1: learnable_mog = FALSE (fixed MoG)
# ------------------------------------------------------------
reset_seeds(1234)

training <- VAE_train(
  data         = data,
  encoder_info = encoder_info,
  decoder_info = decoder_info,
  Lip_en       = 0,
  pi_enc       = 0,
  lip_dec      = 0,
  pi_dec       = 0,
  latent_dim   = 5,
  epoch        = 200,
  beta         = 0.01,
  kl_warm      = TRUE,
  beta_epoch   = 20,
  temperature  = 0.5,
  batchsize    = 16,
  wait         = 20,
  lr           = 0.001,
  K            = 3,
  mog_means    = mog_means,
  mog_log_vars = mog_log_vars,
  mog_weights  = mog_weights,
  prior        = "mixture_gaussian",
  learnable_mog = FALSE
)


# ------------------------------------------------------------------
# Example 2: learnable_mog = TRUE with preset MoG params
# ------------------------------------------------------------------
reset_seeds(1234)

training <- VAE_train(
  data         = data,
  encoder_info = encoder_info,
  decoder_info = decoder_info,
  Lip_en       = 0,
  pi_enc       = 0,
  lip_dec      = 0,
  pi_dec       = 0,
```

```
  latent_dim   = 5,
  epoch        = 200,
  beta         = 0.01,
  kl_warm      = TRUE,
  beta_epoch   = 20,
  temperature  = 0.5,
  batchsize    = 16,
  wait         = 20,
  lr           = 0.001,
  K            = 3,
  mog_means    = mog_means,
  mog_log_vars = mog_log_vars,
  mog_weights  = mog_weights,
  prior        = "mixture_gaussian",
  learnable_mog = TRUE
)


# ------------------------------------------------------------------------
# Example 3: learnable_mog = TRUE with all MoG params learned
#           (mog_means, mog_log_vars, mog_weights = NULL)
# ------------------------------------------------------------------------
reset_seeds(1234)

training <- VAE_train(
  data         = data,
  encoder_info = encoder_info,
  decoder_info = decoder_info,
  Lip_en       = 0,
  pi_enc       = 0,
  lip_dec      = 0,
  pi_dec       = 0,
  latent_dim   = 5,
  epoch        = 200,
  beta         = 0.01,
  kl_warm      = TRUE,
  beta_epoch   = 20,
  temperature  = 0.5,
  batchsize    = 16,
  wait         = 20,
  lr           = 0.001,
  K            = 3,
  mog_means    = NULL,
  mog_log_vars = NULL,
  mog_weights  = NULL,
  prior        = "mixture_gaussian",
  learnable_mog = TRUE
)


}
```

`reset_seeds` *Reset all random seeds across R, TensorFlow, and Python*

Description

Ensures reproducibility by synchronizing random seeds across:

- R's random number generator (`set.seed()`),
- TensorFlow's random state (`tf$random$set_seed()`),
- Python's built-in `random` module.

Usage

```
reset_seeds(spec_seed)
```

Arguments

`spec_seed`	Integer. The seed value to apply across R, TensorFlow, and Python.

Details

This also clears the current Keras/TensorFlow graph and session before reseeding, preventing residual state from prior model builds.

- This function is **not** called automatically within AutoTab. Use it before training runs for reproducibility.
- Equivalent results still require identical environments (same TensorFlow, CUDA/cuDNN, and library versions).

Value

No return value but will print a confirmation message.

See Also

`VAE_train()`, `set_feat_dist()`

Examples

```
if (reticulate::py_module_available("tensorflow")) {
reset_seeds(1234)
}
```

set_feat_dist *Set the feature distribution for AutoTab*

Description

This function stores the output of `extracting_distribution()` / `feat_reorder()` inside the package, so subsequent functions (e.g., `VAE_train()`) can access it safely without relying on the global environment.

Usage

```
set_feat_dist(feat_dist)
```

Arguments

`feat_dist`	A data.frame returned by `extracting_distribution()` or `feat_reorder()`.

VAE_train *Train an AutoTab VAE on mixed-type tabular data*

Description

Runs the full AutoTab training loop (encoder + decoder + latent space), with optional Beta-annealing (linear or cyclical), optional Gumbel-softmax temperature warming for categorical outputs, and options for the prior.

Usage

```
VAE_train(
  data,
  encoder_info,
  decoder_info,
  Lip_en,
  pi_enc = 1,
  lip_dec,
  pi_dec = 1,
  latent_dim,
  epoch,
  beta,
  kl_warm = FALSE,
  kl_cyclical = FALSE,
  n_cycles,
  ratio,
  beta_epoch = 15,
  temperature,
```

```
  temp_warm = FALSE,
  temp_epoch,
  batchsize,
  wait,
  min_delta = 0.001,
  lr,
  max_std = 10,
  min_val = 0.001,
  weighted = 0,
  recon_weights,
  seperate = 0,
  prior = "single_gaussian",
  K = 3,
  learnable_mog = FALSE,
  mog_means = NULL,
  mog_log_vars = NULL,
  mog_weights = NULL
)
```

Arguments

`data`	Matrix/data.frame. **Preprocessed** training data (columns match the order in `feat_dist`).
`encoder_info`, `decoder_info`	Lists describing layer stacks. Each element is e.g. `list("dense", units, "activation", L2_flag, L2_lambda, BN_flag, BN_momentum, BN_learn)` or `list("dropout", rate)`.
`Lip_en`, `lip_dec`	Integer (0/1). Use spectral normalization (Lipschitz) in encoder/decoder.
`pi_enc`, `pi_dec`	Integer. Power-iteration counts for spectral normalization.
`latent_dim`	Integer. Latent dimensionality.
`epoch`	Integer. Max training epochs.
`beta`	Numeric. Beta-VAE weight on the KL term in the ELBO.
`kl_warm`	Logical. Enable Beta-annealing.
`kl_cyclical`	Logical. Enable **cyclical** Beta-annealing (requires `kl_warm = TRUE`).
`n_cycles`	Integer. Number of cycles when `kl_cyclical = TRUE`.
`ratio`	Numeric from range 0 to 1. Fraction of each cycle used for warm-up (rise from 0→Beta).
`beta_epoch`	Integer. Warm-up length (epochs) for **linear** Beta-annealing; when `kl_cyclical = TRUE`, the cycle length is `(beta_epoch / n_cycles)`.
`temperature`	Numeric. Gumbel-softmax temperature (used for categorical heads).
`temp_warm`	Logical. Enable temperature warm-up.
`temp_epoch`	Integer. Warm-up length (epochs) for temperature when `temp_warm = TRUE`.
`batchsize`	Integer. Mini-batch size.
`wait`	Integer. Early-stopping patience (epochs) on validation reconstruction loss.

min_delta	Numeric. Minimum improvement to reset patience (early stopping).
lr	Numeric. Learning rate (Adam).
max_std, min_val	Numerics. Decoder constraints for Gaussian heads (max SD; minimum variance surrogate).
weighted	Integer (0/1). If 1, weight reconstruction terms by type.
recon_weights	Numeric length-3. Weights for (continuous, binary, categorical); **required** when weighted = 1.
seperate	Integer (0/1). If 1, logs per-group reconstruction losses as metrics (cont_loss, bin_loss, cat_loss) in addition to total recon_loss.
prior	Character. "single_gaussian" or "mixture_gaussian".
K	Integer. Number of mixture components when prior = "mixture_gaussian".
learnable_mog	Logical. If TRUE, MoG prior parameters are trainable.
mog_means, mog_log_vars, mog_weights	Optional initial values for the MoG prior (ignored unless prior = "mixture_gaussian"; when learnable_mog = FALSE they must be provided).

Details

Prerequisite: call set_feat_dist() once before training to register the per-feature distributions and parameter counts (see extracting_distribution() and feat_reorder()).

Metrics exposed during training: loss, recon_loss, kl_loss, and, when seperate = 1, cont_loss, bin_loss, cat_loss, and, beta, temperature when annealed.

Early stopping: monitored on val_recon_loss with patience = wait.

Reproducibility: set seeds via your own workflow or the helper reset_seeds().

Expected Warning: When running AutoTab the user will receive the following warning from tensorflow: "WARNING:tensorflow:The following Variables were used in a Lambda layer's call (tf.math.multiply_3), but are not present in its tracked objects: <tf.Variable 'beta:0' shape=() dtype=float32>. This is a strong indication that the Lambda layer should be rewritten as a subclassed Layer."

This is merely a warning and should not effect the computation of AutoTab. This occurs because tensorflow does not see beta, (the weight on the regularization part of the ELBO) until after the first iteration of training and the first computation of the loss is initiated. Therefore it is not an internally tracked object. However, it is being tracked and updated outside of the model graph which can be seen in the KL loss plots and in the training printout in the R console.

Value

A list with:

- trained_model — the compiled Keras model (encoder→decoder) with KL and recon losses added.
- loss_history — numeric vector of per-epoch total loss (as tracked during training).

See Also

set_feat_dist(), extracting_distribution(), feat_reorder(), Encoder_weights(), encoder_latent(), Decoder_weights(), Latent_sample()

Index