```python
#!/usr/bin/env python
# -*- encoding: utf-8 -*-
#
# This file is auto-generated by h2o-3/h2o-bindings/bin/gen_python.py
# Copyright 2016 H2O.ai;  Apache License Version 2.0 (see LICENSE for details)
#
from __future__ import absolute_import, division, print_function, unicode_literals

from h2o.estimators.estimator_base import H2OEstimator
from h2o.exceptions import H2OValueError
from h2o.frame import H2OFrame
from h2o.utils.typechecks import assert_is_type, Enum, numeric


class H2ODeepLearningEstimator(H2OEstimator):
    """
    Deep Learning

    Build a Deep Neural Network model using CPUs
    Builds a feed-forward multilayer artificial neural network on an H2OFrame

    :examples:

    >>> from h2o.estimators.deeplearning import H2ODeepLearningEstimator
    >>> rows = [[1,2,3,4,0], [2,1,2,4,1], [2,1,4,2,1],
    ...         [0,1,2,34,1], [2,3,4,1,0]] * 50
    >>> fr = h2o.H2OFrame(rows)
    >>> fr[4] = fr[4].asfactor()
    >>> model = H2ODeepLearningEstimator()
    >>> model.train(x=range(4), y=4, training_frame=fr)
    >>> model.logloss()
    """
```

```
    algo = "deeplearning"
    supervised_learning = True
    _options_ = {'model_extensions': ['h2o.model.extensions.ScoringHistoryDL',
                                      'h2o.model.extensions.VariableImportance'],
                 'verbose': True}

    def __init__(self,
                 model_id=None,  # type: Optional[Union[None, str, H2OEstimator]]
                 training_frame=None,  # type: Optional[Union[None, str, H2OFrame]]
                 validation_frame=None,  # type: Optional[Union[None, str, H2OFrame]]
                 nfolds=0,  # type: int
                 keep_cross_validation_models=True,  # type: bool
                 keep_cross_validation_predictions=False,  # type: bool
                 keep_cross_validation_fold_assignment=False,  # type: bool
                 fold_assignment="auto",  # type: Literal["auto", "random", "modulo", "stratif
                 fold_column=None,  # type: Optional[str]
                 response_column=None,  # type: Optional[str]
                 ignored_columns=None,  # type: Optional[List[str]]
                 ignore_const_cols=True,  # type: bool
                 score_each_iteration=False,  # type: bool
                 weights_column=None,  # type: Optional[str]
                 offset_column=None,  # type: Optional[str]
                 balance_classes=False,  # type: bool
                 class_sampling_factors=None,  # type: Optional[List[float]]
                 max_after_balance_size=5.0,  # type: float
                 max_confusion_matrix_size=20,  # type: int
                 checkpoint=None,  # type: Optional[Union[None, str, H2OEstimator]]
                 pretrained_autoencoder=None,  # type: Optional[Union[None, str, H2OEstimator]]
                 overwrite_with_best_model=True,  # type: bool
                 use_all_factor_levels=True,  # type: bool
                 standardize=True,  # type: bool
                 activation="rectifier",  # type: Literal["tanh", "tanh_with_dropout", "rectif
                 hidden=[200, 200],  # type: List[int]
                 epochs=10.0,  # type: float
                 train_samples_per_iteration=-2,  # type: int
                 target_ratio_comm_to_comp=0.05,  # type: float
                 seed=-1,  # type: int
                 adaptive_rate=True,  # type: bool
                 rho=0.99,  # type: float
                 epsilon=1e-08,  # type: float
                 rate=0.005,  # type: float
                 rate_annealing=1e-06,  # type: float
                 rate_decay=1.0,  # type: float
                 momentum_start=0.0,  # type: float
                 momentum_ramp=1000000.0,  # type: float
                 momentum_stable=0.0,  # type: float
                 nesterov_accelerated_gradient=True,  # type: bool
                 input_dropout_ratio=0.0,  # type: float
                 hidden_dropout_ratios=None,  # type: Optional[List[float]]
                 l1=0.0,  # type: float
                 l2=0.0,  # type: float
```

```
                max_w2=3.4028235e+38,  # type: float
                initial_weight_distribution="uniform_adaptive",  # type: Literal["uniform_ada
                initial_weight_scale=1.0,  # type: float
                initial_weights=None,  # type: Optional[List[Union[None, str, H2OFrame]]]
                initial_biases=None,  # type: Optional[List[Union[None, str, H2OFrame]]]
                loss="automatic",  # type: Literal["automatic", "cross_entropy", "quadratic",
                distribution="auto",  # type: Literal["auto", "bernoulli", "multinomial", "gau
                quantile_alpha=0.5,  # type: float
                tweedie_power=1.5,  # type: float
                huber_alpha=0.9,  # type: float
                score_interval=5.0,  # type: float
                score_training_samples=10000,  # type: int
                score_validation_samples=0,  # type: int
                score_duty_cycle=0.1,  # type: float
                classification_stop=0.0,  # type: float
                regression_stop=1e-06,  # type: float
                stopping_rounds=5,  # type: int
                stopping_metric="auto",  # type: Literal["auto", "deviance", "logloss", "mse"
                stopping_tolerance=0.0,  # type: float
                max_runtime_secs=0.0,  # type: float
                score_validation_sampling="uniform",  # type: Literal["uniform", "stratified"
                diagnostics=True,  # type: bool
                fast_mode=True,  # type: bool
                force_load_balance=True,  # type: bool
                variable_importances=True,  # type: bool
                replicate_training_data=True,  # type: bool
                single_node_mode=False,  # type: bool
                shuffle_training_data=False,  # type: bool
                missing_values_handling="mean_imputation",  # type: Literal["mean_imputation"
                quiet_mode=False,  # type: bool
                autoencoder=False,  # type: bool
                sparse=False,  # type: bool
                col_major=False,  # type: bool
                average_activation=0.0,  # type: float
                sparsity_beta=0.0,  # type: float
                max_categorical_features=2147483647,  # type: int
                reproducible=False,  # type: bool
                export_weights_and_biases=False,  # type: bool
                mini_batch_size=1,  # type: int
                categorical_encoding="auto",  # type: Literal["auto", "enum", "one_hot_interna
                elastic_averaging=False,  # type: bool
                elastic_averaging_moving_rate=0.9,  # type: float
                elastic_averaging_regularization=0.001,  # type: float
                export_checkpoints_dir=None,  # type: Optional[str]
                auc_type="auto",  # type: Literal["auto", "none", "macro_ovr", "weighted_ovr"
                ):
        """
        :param model_id: Destination id for this model; auto-generated if not specified.
               Defaults to ``None``.
        :type model_id: Union[None, str, H2OEstimator], optional

        :param training_frame: Id of the training data frame.
               Defaults to ``None``.
```

```
137         :type training_frame: Union[None, str, H2OFrame], optional
138         :param validation_frame: Id of the validation data frame.
139                Defaults to ``None``.
140         :type validation_frame: Union[None, str, H2OFrame], optional
141         :param nfolds: Number of folds for K-fold cross-validation (0 to disable or >= 2).
142                Defaults to ``0``.
143         :type nfolds: int
144         :param keep_cross_validation_models: Whether to keep the cross-validation models.
145                Defaults to ``True``.
146         :type keep_cross_validation_models: bool
147         :param keep_cross_validation_predictions: Whether to keep the predictions of the cross-
148                Defaults to ``False``.
149         :type keep_cross_validation_predictions: bool
150         :param keep_cross_validation_fold_assignment: Whether to keep the cross-validation fold
151                Defaults to ``False``.
152         :type keep_cross_validation_fold_assignment: bool
153         :param fold_assignment: Cross-validation fold assignment scheme, if fold_column is not
154                'Stratified' option will stratify the folds based on the response variable, for
155                Defaults to ``"auto"``.
156         :type fold_assignment: Literal["auto", "random", "modulo", "stratified"]
157         :param fold_column: Column with cross-validation fold index assignment per observation
158                Defaults to ``None``.
159         :type fold_column: str, optional
160         :param response_column: Response variable column.
161                Defaults to ``None``.
162         :type response_column: str, optional
163         :param ignored_columns: Names of columns to ignore for training.
164                Defaults to ``None``.
165         :type ignored_columns: List[str], optional
166         :param ignore_const_cols: Ignore constant columns.
167                Defaults to ``True``.
168         :type ignore_const_cols: bool
169         :param score_each_iteration: Whether to score during each iteration of model training.
170                Defaults to ``False``.
171         :type score_each_iteration: bool
172         :param weights_column: Column with observation weights. Giving some observation a weigh
173                to excluding it from the dataset; giving an observation a relative weight of 2 i
174                that row twice. Negative weights are not allowed. Note: Weights are per-row obse
175                not increase the size of the data frame. This is typically the number of times a
176                non-integer values are supported as well. During training, rows with higher weig
177                the larger loss function pre-factor.
178                Defaults to ``None``.
179         :type weights_column: str, optional
180         :param offset_column: Offset column. This will be added to the combination of columns b
181                function.
182                Defaults to ``None``.
183         :type offset_column: str, optional
184         :param balance_classes: Balance training data class counts via over/under-sampling (for
185                Defaults to ``False``.
186         :type balance_classes: bool
187         :param class_sampling_factors: Desired over/under-sampling ratios per class (in lexicog
188                specified, sampling factors will be automatically computed to obtain class balan
```

```
189                Requires balance_classes.
190                Defaults to ``None``.
191            :type class_sampling_factors: List[float], optional
192            :param max_after_balance_size: Maximum relative size of the training data after balanci
193                less than 1.0). Requires balance_classes.
194                Defaults to ``5.0``.
195            :type max_after_balance_size: float
196            :param max_confusion_matrix_size: [Deprecated] Maximum size (# classes) for confusion r
197                the Logs.
198                Defaults to ``20``.
199            :type max_confusion_matrix_size: int
200            :param checkpoint: Model checkpoint to resume training with.
201                Defaults to ``None``.
202            :type checkpoint: Union[None, str, H2OEstimator], optional
203            :param pretrained_autoencoder: Pretrained autoencoder model to initialize this model wi
204                Defaults to ``None``.
205            :type pretrained_autoencoder: Union[None, str, H2OEstimator], optional
206            :param overwrite_with_best_model: If enabled, override the final model with the best mo
207                training.
208                Defaults to ``True``.
209            :type overwrite_with_best_model: bool
210            :param use_all_factor_levels: Use all factor levels of categorical variables. Otherwise
211                is omitted (without loss of accuracy). Useful for variable importances and auto-
212                Defaults to ``True``.
213            :type use_all_factor_levels: bool
214            :param standardize: If enabled, automatically standardize the data. If disabled, the us
215                scaled input data.
216                Defaults to ``True``.
217            :type standardize: bool
218            :param activation: Activation function.
219                Defaults to ``"rectifier"``.
220            :type activation: Literal["tanh", "tanh_with_dropout", "rectifier", "rectifier_with_dro
221                "maxout_with_dropout"]
222            :param hidden: Hidden layer sizes (e.g. [100, 100]).
223                Defaults to ``[200, 200]``.
224            :type hidden: List[int]
225            :param epochs: How many times the dataset should be iterated (streamed), can be fractio
226                Defaults to ``10.0``.
227            :type epochs: float
228            :param train_samples_per_iteration: Number of training samples (globally) per MapReduce
229                values are 0: one epoch, -1: all available data (e.g., replicated training data)
230                Defaults to ``-2``.
231            :type train_samples_per_iteration: int
232            :param target_ratio_comm_to_comp: Target ratio of communication overhead to computation
233                operation and train_samples_per_iteration = -2 (auto-tuning).
234                Defaults to ``0.05``.
235            :type target_ratio_comm_to_comp: float
236            :param seed: Seed for random numbers (affects sampling) - Note: only reproducible when
237                Defaults to ``-1``.
238            :type seed: int
239            :param adaptive_rate: Adaptive learning rate.
240                Defaults to ``True``.
```

```
241         :type adaptive_rate: bool
242         :param rho: Adaptive learning rate time decay factor (similarity to prior updates).
243                Defaults to ``0.99``.
244         :type rho: float
245         :param epsilon: Adaptive learning rate smoothing factor (to avoid divisions by zero and
246                Defaults to ``1e-08``.
247         :type epsilon: float
248         :param rate: Learning rate (higher => less stable, lower => slower convergence).
249                Defaults to ``0.005``.
250         :type rate: float
251         :param rate_annealing: Learning rate annealing: rate / (1 + rate_annealing * samples).
252                Defaults to ``1e-06``.
253         :type rate_annealing: float
254         :param rate_decay: Learning rate decay factor between layers (N-th layer: rate * rate_d
255                Defaults to ``1.0``.
256         :type rate_decay: float
257         :param momentum_start: Initial momentum at the beginning of training (try 0.5).
258                Defaults to ``0.0``.
259         :type momentum_start: float
260         :param momentum_ramp: Number of training samples for which momentum increases.
261                Defaults to ``1000000.0``.
262         :type momentum_ramp: float
263         :param momentum_stable: Final momentum after the ramp is over (try 0.99).
264                Defaults to ``0.0``.
265         :type momentum_stable: float
266         :param nesterov_accelerated_gradient: Use Nesterov accelerated gradient (recommended).
267                Defaults to ``True``.
268         :type nesterov_accelerated_gradient: bool
269         :param input_dropout_ratio: Input layer dropout ratio (can improve generalization, try
270                Defaults to ``0.0``.
271         :type input_dropout_ratio: float
272         :param hidden_dropout_ratios: Hidden layer dropout ratios (can improve generalization),
273                hidden layer, defaults to 0.5.
274                Defaults to ``None``.
275         :type hidden_dropout_ratios: List[float], optional
276         :param l1: L1 regularization (can add stability and improve generalization, causes many
277                Defaults to ``0.0``.
278         :type l1: float
279         :param l2: L2 regularization (can add stability and improve generalization, causes many
280                Defaults to ``0.0``.
281         :type l2: float
282         :param max_w2: Constraint for squared sum of incoming weights per unit (e.g. for Rectif
283                Defaults to ``3.4028235e+38``.
284         :type max_w2: float
285         :param initial_weight_distribution: Initial weight distribution.
286                Defaults to ``"uniform_adaptive"``.
287         :type initial_weight_distribution: Literal["uniform_adaptive", "uniform", "normal"]
288         :param initial_weight_scale: Uniform: -value...value, Normal: stddev.
289                Defaults to ``1.0``.
290         :type initial_weight_scale: float
291         :param initial_weights: A list of H2OFrame ids to initialize the weight matrices of thi
292                Defaults to ``None``.
```

```
293         :type initial_weights: List[Union[None, str, H2OFrame]], optional
294         :param initial_biases: A list of H2OFrame ids to initialize the bias vectors of this mo
295                Defaults to ``None``.
296         :type initial_biases: List[Union[None, str, H2OFrame]], optional
297         :param loss: Loss function.
298                Defaults to ``"automatic"``.
299         :type loss: Literal["automatic", "cross_entropy", "quadratic", "huber", "absolute", "qu
300         :param distribution: Distribution function
301                Defaults to ``"auto"``.
302         :type distribution: Literal["auto", "bernoulli", "multinomial", "gaussian", "poisson",
303                "quantile", "huber"]
304         :param quantile_alpha: Desired quantile for Quantile regression, must be between 0 and
305                Defaults to ``0.5``.
306         :type quantile_alpha: float
307         :param tweedie_power: Tweedie power for Tweedie regression, must be between 1 and 2.
308                Defaults to ``1.5``.
309         :type tweedie_power: float
310         :param huber_alpha: Desired quantile for Huber/M-regression (threshold between quadrati
311                be between 0 and 1).
312                Defaults to ``0.9``.
313         :type huber_alpha: float
314         :param score_interval: Shortest time interval (in seconds) between model scoring.
315                Defaults to ``5.0``.
316         :type score_interval: float
317         :param score_training_samples: Number of training set samples for scoring (0 for all).
318                Defaults to ``10000``.
319         :type score_training_samples: int
320         :param score_validation_samples: Number of validation set samples for scoring (0 for al
321                Defaults to ``0``.
322         :type score_validation_samples: int
323         :param score_duty_cycle: Maximum duty cycle fraction for scoring (lower: more training,
324                Defaults to ``0.1``.
325         :type score_duty_cycle: float
326         :param classification_stop: Stopping criterion for classification error fraction on tra
327                disable).
328                Defaults to ``0.0``.
329         :type classification_stop: float
330         :param regression_stop: Stopping criterion for regression error (MSE) on training data
331                Defaults to ``1e-06``.
332         :type regression_stop: float
333         :param stopping_rounds: Early stopping based on convergence of stopping_metric. Stop if
334                length k of the stopping_metric does not improve for k:=stopping_rounds scoring
335                Defaults to ``5``.
336         :type stopping_rounds: int
337         :param stopping_metric: Metric to use for early stopping (AUTO: logloss for classificat
338                regression and anonomaly_score for Isolation Forest). Note that custom and custo
339                used in GBM and DRF with the Python client.
340                Defaults to ``"auto"``.
341         :type stopping_metric: Literal["auto", "deviance", "logloss", "mse", "rmse", "mae", "rm
342                "misclassification", "mean_per_class_error", "custom", "custom_increasing"]
343         :param stopping_tolerance: Relative tolerance for metric-based stopping criterion (stop
344                is not at least this much)
```

```
345                Defaults to ``0.0``.
346            :type stopping_tolerance: float
347            :param max_runtime_secs: Maximum allowed runtime in seconds for model training. Use 0 t
348                Defaults to ``0.0``.
349            :type max_runtime_secs: float
350            :param score_validation_sampling: Method used to sample validation dataset for scoring
351                Defaults to ``"uniform"``.
352            :type score_validation_sampling: Literal["uniform", "stratified"]
353            :param diagnostics: Enable diagnostics for hidden layers.
354                Defaults to ``True``.
355            :type diagnostics: bool
356            :param fast_mode: Enable fast mode (minor approximation in back-propagation).
357                Defaults to ``True``.
358            :type fast_mode: bool
359            :param force_load_balance: Force extra load balancing to increase training speed for sm
360                cores busy).
361                Defaults to ``True``.
362            :type force_load_balance: bool
363            :param variable_importances: Compute variable importances for input features (Gedeon me
364                large networks.
365                Defaults to ``True``.
366            :type variable_importances: bool
367            :param replicate_training_data: Replicate the entire training dataset onto every node f
368                small datasets.
369                Defaults to ``True``.
370            :type replicate_training_data: bool
371            :param single_node_mode: Run on a single node for fine-tuning of model parameters.
372                Defaults to ``False``.
373            :type single_node_mode: bool
374            :param shuffle_training_data: Enable shuffling of training data (recommended if trainin
375                train_samples_per_iteration is close to #nodes x #rows, of if using balance_clas
376                Defaults to ``False``.
377            :type shuffle_training_data: bool
378            :param missing_values_handling: Handling of missing values. Either MeanImputation or Sk
379                Defaults to ``"mean_imputation"``.
380            :type missing_values_handling: Literal["mean_imputation", "skip"]
381            :param quiet_mode: Enable quiet mode for less output to standard output.
382                Defaults to ``False``.
383            :type quiet_mode: bool
384            :param autoencoder: Auto-Encoder.
385                Defaults to ``False``.
386            :type autoencoder: bool
387            :param sparse: Sparse data handling (more efficient for data with lots of 0 values).
388                Defaults to ``False``.
389            :type sparse: bool
390            :param col_major: #DEPRECATED Use a column major weight matrix for input layer. Can spe
391                propagation, but might slow down backpropagation.
392                Defaults to ``False``.
393            :type col_major: bool
394            :param average_activation: Average activation for sparse auto-encoder. #Experimental
395                Defaults to ``0.0``.
396            :type average_activation: float
```

```
        :param sparsity_beta: Sparsity regularization. #Experimental
               Defaults to ``0.0``.
        :type sparsity_beta: float
        :param max_categorical_features: Max. number of categorical features, enforced via hash
               Defaults to ``2147483647``.
        :type max_categorical_features: int
        :param reproducible: Force reproducibility on small data (will be slow - only uses 1 th
               Defaults to ``False``.
        :type reproducible: bool
        :param export_weights_and_biases: Whether to export Neural Network weights and biases t
               Defaults to ``False``.
        :type export_weights_and_biases: bool
        :param mini_batch_size: Mini-batch size (smaller leads to better fit, larger can speed
               better).
               Defaults to ``1``.
        :type mini_batch_size: int
        :param categorical_encoding: Encoding scheme for categorical features
               Defaults to ``"auto"``.
        :type categorical_encoding: Literal["auto", "enum", "one_hot_internal", "one_hot_explic
               "sort_by_response", "enum_limited"]
        :param elastic_averaging: Elastic averaging between compute nodes can improve distribut
               #Experimental
               Defaults to ``False``.
        :type elastic_averaging: bool
        :param elastic_averaging_moving_rate: Elastic averaging moving rate (only if elastic av
               Defaults to ``0.9``.
        :type elastic_averaging_moving_rate: float
        :param elastic_averaging_regularization: Elastic averaging regularization strength (onl
               enabled).
               Defaults to ``0.001``.
        :type elastic_averaging_regularization: float
        :param export_checkpoints_dir: Automatically export generated models to this directory
               Defaults to ``None``.
        :type export_checkpoints_dir: str, optional
        :param auc_type: Set default multinomial AUC type.
               Defaults to ``"auto"``.
        :type auc_type: Literal["auto", "none", "macro_ovr", "weighted_ovr", "macro_ovo", "weig
        """
        super(H2ODeepLearningEstimator, self).__init__()
        self._parms = {}
        self._id = self._parms['model_id'] = model_id
        self.training_frame = training_frame
        self.validation_frame = validation_frame
        self.nfolds = nfolds
        self.keep_cross_validation_models = keep_cross_validation_models
        self.keep_cross_validation_predictions = keep_cross_validation_predictions
        self.keep_cross_validation_fold_assignment = keep_cross_validation_fold_assignment
        self.fold_assignment = fold_assignment
        self.fold_column = fold_column
        self.response_column = response_column
        self.ignored_columns = ignored_columns
        self.ignore_const_cols = ignore_const_cols
```

```
        self.score_each_iteration = score_each_iteration
        self.weights_column = weights_column
        self.offset_column = offset_column
        self.balance_classes = balance_classes
        self.class_sampling_factors = class_sampling_factors
        self.max_after_balance_size = max_after_balance_size
        self.max_confusion_matrix_size = max_confusion_matrix_size
        self.checkpoint = checkpoint
        self.pretrained_autoencoder = pretrained_autoencoder
        self.overwrite_with_best_model = overwrite_with_best_model
        self.use_all_factor_levels = use_all_factor_levels
        self.standardize = standardize
        self.activation = activation
        self.hidden = hidden
        self.epochs = epochs
        self.train_samples_per_iteration = train_samples_per_iteration
        self.target_ratio_comm_to_comp = target_ratio_comm_to_comp
        self.seed = seed
        self.adaptive_rate = adaptive_rate
        self.rho = rho
        self.epsilon = epsilon
        self.rate = rate
        self.rate_annealing = rate_annealing
        self.rate_decay = rate_decay
        self.momentum_start = momentum_start
        self.momentum_ramp = momentum_ramp
        self.momentum_stable = momentum_stable
        self.nesterov_accelerated_gradient = nesterov_accelerated_gradient
        self.input_dropout_ratio = input_dropout_ratio
        self.hidden_dropout_ratios = hidden_dropout_ratios
        self.l1 = l1
        self.l2 = l2
        self.max_w2 = max_w2
        self.initial_weight_distribution = initial_weight_distribution
        self.initial_weight_scale = initial_weight_scale
        self.initial_weights = initial_weights
        self.initial_biases = initial_biases
        self.loss = loss
        self.distribution = distribution
        self.quantile_alpha = quantile_alpha
        self.tweedie_power = tweedie_power
        self.huber_alpha = huber_alpha
        self.score_interval = score_interval
        self.score_training_samples = score_training_samples
        self.score_validation_samples = score_validation_samples
        self.score_duty_cycle = score_duty_cycle
        self.classification_stop = classification_stop
        self.regression_stop = regression_stop
        self.stopping_rounds = stopping_rounds
        self.stopping_metric = stopping_metric
        self.stopping_tolerance = stopping_tolerance
        self.max_runtime_secs = max_runtime_secs
```

```
        self.score_validation_sampling = score_validation_sampling
        self.diagnostics = diagnostics
        self.fast_mode = fast_mode
        self.force_load_balance = force_load_balance
        self.variable_importances = variable_importances
        self.replicate_training_data = replicate_training_data
        self.single_node_mode = single_node_mode
        self.shuffle_training_data = shuffle_training_data
        self.missing_values_handling = missing_values_handling
        self.quiet_mode = quiet_mode
        self.autoencoder = autoencoder
        self.sparse = sparse
        self.col_major = col_major
        self.average_activation = average_activation
        self.sparsity_beta = sparsity_beta
        self.max_categorical_features = max_categorical_features
        self.reproducible = reproducible
        self.export_weights_and_biases = export_weights_and_biases
        self.mini_batch_size = mini_batch_size
        self.categorical_encoding = categorical_encoding
        self.elastic_averaging = elastic_averaging
        self.elastic_averaging_moving_rate = elastic_averaging_moving_rate
        self.elastic_averaging_regularization = elastic_averaging_regularization
        self.export_checkpoints_dir = export_checkpoints_dir
        self.auc_type = auc_type

    @property
    def training_frame(self):
        """
        Id of the training data frame.

        Type: ``Union[None, str, H2OFrame]``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator()
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
```

```
553         return self._parms.get("training_frame")
554 
555     @training_frame.setter
556     def training_frame(self, training_frame):
557         self._parms["training_frame"] = H2OFrame._validate(training_frame, 'training_frame')
558 
559     @property
560     def validation_frame(self):
561         """
562         Id of the validation data frame.
563 
564         Type: ``Union[None, str, H2OFrame]``.
565 
566         :examples:
567 
568         >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
569         >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
570         >>> predictors = ["displacement","power","weight","acceleration","year"]
571         >>> response = "economy_20mpg"
572         >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
573         >>> cars_dl = H2ODeepLearningEstimator(standardize=True,
574         ...                                    seed=1234)
575         >>> cars_dl.train(x=predictors,
576         ...               y=response,
577         ...               training_frame=train,
578         ...               validation_frame=valid)
579         >>> cars_dl.auc()
580         """
581         return self._parms.get("validation_frame")
582 
583     @validation_frame.setter
584     def validation_frame(self, validation_frame):
585         self._parms["validation_frame"] = H2OFrame._validate(validation_frame, 'validation_fra
586 
587     @property
588     def nfolds(self):
589         """
590         Number of folds for K-fold cross-validation (0 to disable or >= 2).
591 
592         Type: ``int``, defaults to ``0``.
593 
594         :examples:
595 
596         >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
597         >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
598         >>> predictors = ["displacement","power","weight","acceleration","year"]
599         >>> response = "economy_20mpg"
600         >>> cars_dl = H2ODeepLearningEstimator(nfolds=5, seed=1234)
601         >>> cars_dl.train(x=predictors,
602         ...               y=response,
603         ...               training_frame=cars)
604         >>> cars_dl.auc()
```

```
        """
        return self._parms.get("nfolds")

    @nfolds.setter
    def nfolds(self, nfolds):
        assert_is_type(nfolds, None, int)
        self._parms["nfolds"] = nfolds

    @property
    def keep_cross_validation_models(self):
        """
        Whether to keep the cross-validation models.

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> cars_dl = H2ODeepLearningEstimator(keep_cross_validation_models=True,
        ...                                    nfolds=5,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> print(cars_dl.cross_validation_models())
        """
        return self._parms.get("keep_cross_validation_models")

    @keep_cross_validation_models.setter
    def keep_cross_validation_models(self, keep_cross_validation_models):
        assert_is_type(keep_cross_validation_models, None, bool)
        self._parms["keep_cross_validation_models"] = keep_cross_validation_models

    @property
    def keep_cross_validation_predictions(self):
        """
        Whether to keep the predictions of the cross-validation models.

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> cars_dl = H2ODeepLearningEstimator(keep_cross_validation_predictions=True,
        ...                                    nfolds=5,
        ...                                    seed=1234)
```

```
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> print(cars_dl.cross_validation_predictions())
        """
        return self._parms.get("keep_cross_validation_predictions")

    @keep_cross_validation_predictions.setter
    def keep_cross_validation_predictions(self, keep_cross_validation_predictions):
        assert_is_type(keep_cross_validation_predictions, None, bool)
        self._parms["keep_cross_validation_predictions"] = keep_cross_validation_predictions

    @property
    def keep_cross_validation_fold_assignment(self):
        """
        Whether to keep the cross-validation fold assignment.

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> cars_dl = H2ODeepLearningEstimator(keep_cross_validation_fold_assignment=True,
        ...                                    nfolds=5,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> print(cars_dl.cross_validation_fold_assignment())
        """
        return self._parms.get("keep_cross_validation_fold_assignment")

    @keep_cross_validation_fold_assignment.setter
    def keep_cross_validation_fold_assignment(self, keep_cross_validation_fold_assignment):
        assert_is_type(keep_cross_validation_fold_assignment, None, bool)
        self._parms["keep_cross_validation_fold_assignment"] = keep_cross_validation_fold_assig

    @property
    def fold_assignment(self):
        """
        Cross-validation fold assignment scheme, if fold_column is not specified. The 'Stratif
        the folds based on the response variable, for classification problems.

        Type: ``Literal["auto", "random", "modulo", "stratified"]``, defaults to ``"auto"``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
```

```
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(fold_assignment="Random",
        ...                                    nfolds=5,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("fold_assignment")

    @fold_assignment.setter
    def fold_assignment(self, fold_assignment):
        assert_is_type(fold_assignment, None, Enum("auto", "random", "modulo", "stratified"))
        self._parms["fold_assignment"] = fold_assignment

    @property
    def fold_column(self):
        """
        Column with cross-validation fold index assignment per observation.

        Type: ``str``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> fold_numbers = cars.kfold_column(n_folds=5, seed=1234)
        >>> fold_numbers.set_names(["fold_numbers"])
        >>> cars = cars.cbind(fold_numbers)
        >>> print(cars['fold_numbers'])
        >>> cars_dl = H2ODeepLearningEstimator(seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars,
        ...               fold_column="fold_numbers")
        >>> cars_dl.mse()
        """
        return self._parms.get("fold_column")

    @fold_column.setter
    def fold_column(self, fold_column):
        assert_is_type(fold_column, None, str)
        self._parms["fold_column"] = fold_column


    @property
    def response_column(self):
```

```
        """
        Response variable column.

        Type: ``str``.
        """
        return self._parms.get("response_column")

    @response_column.setter
    def response_column(self, response_column):
        assert_is_type(response_column, None, str)
        self._parms["response_column"] = response_column

    @property
    def ignored_columns(self):
        """
        Names of columns to ignore for training.

        Type: ``List[str]``.
        """
        return self._parms.get("ignored_columns")

    @ignored_columns.setter
    def ignored_columns(self, ignored_columns):
        assert_is_type(ignored_columns, None, [str])
        self._parms["ignored_columns"] = ignored_columns

    @property
    def ignore_const_cols(self):
        """
        Ignore constant columns.

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> cars["const_1"] = 6
        >>> cars["const_2"] = 7
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(seed=1234,
        ...                                    ignore_const_cols=True)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.auc()
        """
        return self._parms.get("ignore_const_cols")

```

```
    @ignore_const_cols.setter
    def ignore_const_cols(self, ignore_const_cols):
        assert_is_type(ignore_const_cols, None, bool)
        self._parms["ignore_const_cols"] = ignore_const_cols

    @property
    def score_each_iteration(self):
        """
        Whether to score during each iteration of model training.

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> cars_dl = H2ODeepLearningEstimator(score_each_iteration=True,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> cars_dl.auc()
        """
        return self._parms.get("score_each_iteration")

    @score_each_iteration.setter
    def score_each_iteration(self, score_each_iteration):
        assert_is_type(score_each_iteration, None, bool)
        self._parms["score_each_iteration"] = score_each_iteration

    @property
    def weights_column(self):
        """
        Column with observation weights. Giving some observation a weight of zero is equivalent
        dataset; giving an observation a relative weight of 2 is equivalent to repeating that r
        weights are not allowed. Note: Weights are per-row observation weights and do not incre
        frame. This is typically the number of times a row is repeated, but non-integer values
        During training, rows with higher weights matter more, due to the larger loss function

        Type: ``str``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(seed=1234)
        >>> cars_dl.train(x=predictors,
```

```
865             ...                 y=response,
866             ...                 training_frame=train,
867             ...                 validation_frame=valid)
868             >>> cars_dl.auc()
869             """
870             return self._parms.get("weights_column")
871
872         @weights_column.setter
873         def weights_column(self, weights_column):
874             assert_is_type(weights_column, None, str)
875             self._parms["weights_column"] = weights_column
876
877         @property
878         def offset_column(self):
879             """
880             Offset column. This will be added to the combination of columns before applying the lin
881
882             Type: ``str``.
883
884             :examples:
885
886             >>> boston = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/g
887             >>> predictors = boston.columns[:-1]
888             >>> response = "medv"
889             >>> boston['chas'] = boston['chas'].asfactor()
890             >>> boston["offset"] = boston["medv"].log()
891             >>> train, valid = boston.split_frame(ratios=[.8], seed=1234)
892             >>> boston_dl = H2ODeepLearningEstimator(offset_column="offset",
893             ...                                      seed=1234)
894             >>> boston_dl.train(x=predictors,
895             ...                 y=response,
896             ...                 training_frame=train,
897             ...                 validation_frame=valid)
898             >>> boston_dl.mse()
899             """
900             return self._parms.get("offset_column")
901
902         @offset_column.setter
903         def offset_column(self, offset_column):
904             assert_is_type(offset_column, None, str)
905             self._parms["offset_column"] = offset_column
906
907         @property
908         def balance_classes(self):
909             """
910             Balance training data class counts via over/under-sampling (for imbalanced data).
911
912             Type: ``bool``, defaults to ``False``.
913
914             :examples:
915
916             >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
```

```
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> cov_dl = H2ODeepLearningEstimator(balance_classes=True,
        ...                                   seed=1234)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.mse()
        """
        return self._parms.get("balance_classes")

    @balance_classes.setter
    def balance_classes(self, balance_classes):
        assert_is_type(balance_classes, None, bool)
        self._parms["balance_classes"] = balance_classes

    @property
    def class_sampling_factors(self):
        """
        Desired over/under-sampling ratios per class (in lexicographic order). If not specifie
        be automatically computed to obtain class balance during training. Requires balance_cl

        Type: ``List[float]``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> sample_factors = [1., 0.5, 1., 1., 1., 1., 1.]
        >>> cars_dl = H2ODeepLearningEstimator(balance_classes=True,
        ...                                    class_sampling_factors=sample_factors,
        ...                                    seed=1234)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.mse()
        """
        return self._parms.get("class_sampling_factors")

    @class_sampling_factors.setter
    def class_sampling_factors(self, class_sampling_factors):
        assert_is_type(class_sampling_factors, None, [float])
        self._parms["class_sampling_factors"] = class_sampling_factors


    @property
```

```
    def max_after_balance_size(self):
        """
        Maximum relative size of the training data after balancing class counts (can be less th
        balance_classes.

        Type: ``float``, defaults to ``5.0``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> max = .85
        >>> cov_dl = H2ODeepLearningEstimator(balance_classes=True,
        ...                                   max_after_balance_size=max,
        ...                                   seed=1234)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.logloss()
        """
        return self._parms.get("max_after_balance_size")

    @max_after_balance_size.setter
    def max_after_balance_size(self, max_after_balance_size):
        assert_is_type(max_after_balance_size, None, float)
        self._parms["max_after_balance_size"] = max_after_balance_size

    @property
    def max_confusion_matrix_size(self):
        """
        [Deprecated] Maximum size (# classes) for confusion matrices to be printed in the Logs

        Type: ``int``, defaults to ``20``.
        """
        return self._parms.get("max_confusion_matrix_size")

    @max_confusion_matrix_size.setter
    def max_confusion_matrix_size(self, max_confusion_matrix_size):
        assert_is_type(max_confusion_matrix_size, None, int)
        self._parms["max_confusion_matrix_size"] = max_confusion_matrix_size

    @property
    def checkpoint(self):
        """
        Model checkpoint to resume training with.

        Type: ``Union[None, str, H2OEstimator]``.

```

```
        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(activation="tanh",
        ...                                    autoencoder=True,
        ...                                    seed=1234,
        ...                                    model_id="cars_dl")
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        >>> cars_cont = H2ODeepLearningEstimator(checkpoint=cars_dl,
        ...                                      seed=1234)
        >>> cars_cont.train(x=predictors,
        ...                 y=response,
        ...                 training_frame=train,
        ...                 validation_frame=valid)
        >>> cars_cont.mse()
        """
        return self._parms.get("checkpoint")

    @checkpoint.setter
    def checkpoint(self, checkpoint):
        assert_is_type(checkpoint, None, str, H2OEstimator)
        self._parms["checkpoint"] = checkpoint

    @property
    def pretrained_autoencoder(self):
        """
        Pretrained autoencoder model to initialize this model with.

        Type: ``Union[None, str, H2OEstimator]``.

        :examples:

        >>> from h2o.estimators.deeplearning import H2OAutoEncoderEstimator
        >>> resp = 784
        >>> nfeatures = 20
        >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapto
        >>> train[resp] = train[resp].asfactor()
        >>> test[resp] = test[resp].asfactor()
        >>> sid = train[0].runif(0)
        >>> train_unsupervised = train[sid>=0.5]
        >>> train_unsupervised.pop(resp)
        >>> train_supervised = train[sid<0.5]
        >>> ae_model = H2OAutoEncoderEstimator(activation="Tanh",
```

```
        ...                                     hidden=[nfeatures],
        ...                                     model_id="ae_model",
        ...                                     epochs=1,
        ...                                     ignore_const_cols=False,
        ...                                     reproducible=True,
        ...                                     seed=1234)
        >>> ae_model.train(list(range(resp)), training_frame=train_unsupervised)
        >>> ae_model.mse()
        >>> pretrained_model = H2ODeepLearningEstimator(activation="Tanh",
        ...                                             hidden=[nfeatures],
        ...                                             epochs=1,
        ...                                             reproducible = True,
        ...                                             seed=1234,
        ...                                             ignore_const_cols=False,
        ...                                             pretrained_autoencoder="ae_model")
        >>> pretrained_model.train(list(range(resp)), resp,
        ...                        training_frame=train_supervised,
        ...                        validation_frame=test)
        >>> pretrained_model.mse()
        """
        return self._parms.get("pretrained_autoencoder")

    @pretrained_autoencoder.setter
    def pretrained_autoencoder(self, pretrained_autoencoder):
        assert_is_type(pretrained_autoencoder, None, str, H2OEstimator)
        self._parms["pretrained_autoencoder"] = pretrained_autoencoder

    @property
    def overwrite_with_best_model(self):
        """
        If enabled, override the final model with the best model found during training.

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> boston = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/g
        >>> predictors = boston.columns[:-1]
        >>> response = "medv"
        >>> boston['chas'] = boston['chas'].asfactor()
        >>> boston["offset"] = boston["medv"].log()
        >>> train, valid = boston.split_frame(ratios=[.8], seed=1234)
        >>> boston_dl = H2ODeepLearningEstimator(overwrite_with_best_model=True,
        ...                                      seed=1234)
        >>> boston_dl.train(x=predictors,
        ...                 y=response,
        ...                 training_frame=train,
        ...                 validation_frame=valid)
        >>> boston_dl.mse()
        """
        return self._parms.get("overwrite_with_best_model")

```

```
    @overwrite_with_best_model.setter
    def overwrite_with_best_model(self, overwrite_with_best_model):
        assert_is_type(overwrite_with_best_model, None, bool)
        self._parms["overwrite_with_best_model"] = overwrite_with_best_model

    @property
    def use_all_factor_levels(self):
        """
        Use all factor levels of categorical variables. Otherwise, the first factor level is or
        accuracy). Useful for variable importances and auto-enabled for autoencoder.

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(use_all_factor_levels=True,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.mse()
        """
        return self._parms.get("use_all_factor_levels")

    @use_all_factor_levels.setter
    def use_all_factor_levels(self, use_all_factor_levels):
        assert_is_type(use_all_factor_levels, None, bool)
        self._parms["use_all_factor_levels"] = use_all_factor_levels

    @property
    def standardize(self):
        """
        If enabled, automatically standardize the data. If disabled, the user must provide prop

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
```

```
        >>> response = "economy_20mpg"
        >>> cars_dl = H2ODeepLearningEstimator(standardize=True,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> cars_dl.auc()
        """
        return self._parms.get("standardize")

    @standardize.setter
    def standardize(self, standardize):
        assert_is_type(standardize, None, bool)
        self._parms["standardize"] = standardize

    @property
    def activation(self):
        """
        Activation function.

        Type: ``Literal["tanh", "tanh_with_dropout", "rectifier", "rectifier_with_dropout", "ma
        "maxout_with_dropout"]``, defaults to ``"rectifier"``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> cars_dl = H2ODeepLearningEstimator(activation="tanh")
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("activation")

    @activation.setter
    def activation(self, activation):
        assert_is_type(activation, None, Enum("tanh", "tanh_with_dropout", "rectifier", "recti
        self._parms["activation"] = activation

    @property
    def hidden(self):
        """
        Hidden layer sizes (e.g. [100, 100]).

        Type: ``List[int]``, defaults to ``[200, 200]``.

        :examples:
```

```
        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(hidden=[100,100],
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("hidden")

    @hidden.setter
    def hidden(self, hidden):
        assert_is_type(hidden, None, [int])
        self._parms["hidden"] = hidden

    @property
    def epochs(self):
        """
        How many times the dataset should be iterated (streamed), can be fractional.

        Type: ``float``, defaults to ``10.0``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(epochs=15,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("epochs")

    @epochs.setter
    def epochs(self, epochs):
        assert_is_type(epochs, None, numeric)
        self._parms["epochs"] = epochs

    @property
    def train_samples_per_iteration(self):
        """
```

```
        Number of training samples (globally) per MapReduce iteration. Special values are 0: or
        available data (e.g., replicated training data), -2: automatic.

        Type: ``int``, defaults to ``-2``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(train_samples_per_iteration=-1,
        ...                                        epochs=1,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("train_samples_per_iteration")

    @train_samples_per_iteration.setter
    def train_samples_per_iteration(self, train_samples_per_iteration):
        assert_is_type(train_samples_per_iteration, None, int)
        self._parms["train_samples_per_iteration"] = train_samples_per_iteration

    @property
    def target_ratio_comm_to_comp(self):
        """
        Target ratio of communication overhead to computation. Only for multi-node operation ar
        train_samples_per_iteration = -2 (auto-tuning).

        Type: ``float``, defaults to ``0.05``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
```

```
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(target_ratio_comm_to_comp=0.05,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("target_ratio_comm_to_comp")

    @target_ratio_comm_to_comp.setter
    def target_ratio_comm_to_comp(self, target_ratio_comm_to_comp):
        assert_is_type(target_ratio_comm_to_comp, None, numeric)
        self._parms["target_ratio_comm_to_comp"] = target_ratio_comm_to_comp

    @property
    def seed(self):
        """
        Seed for random numbers (affects sampling) - Note: only reproducible when running sing

        Type: ``int``, defaults to ``-1``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/ju
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.auc()
        """
        return self._parms.get("seed")

    @seed.setter
    def seed(self, seed):
        assert_is_type(seed, None, int)
        self._parms["seed"] = seed

    @property
    def adaptive_rate(self):
        """
        Adaptive learning rate.

        Type: ``bool``, defaults to ``True``.

        :examples:
```

```
1385
1386            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
1387            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
1388            >>> predictors = ["displacement","power","weight","acceleration","year"]
1389            >>> response = "cylinders"
1390            >>> cars_dl = H2ODeepLearningEstimator(adaptive_rate=True)
1391            >>> cars_dl.train(x=predictors,
1392            ...               y=response,
1393            ...               training_frame=train,
1394            ...               validation_frame=valid)
1395            >>> cars_dl.mse()
1396            """
1397            return self._parms.get("adaptive_rate")
1398
1399        @adaptive_rate.setter
1400        def adaptive_rate(self, adaptive_rate):
1401            assert_is_type(adaptive_rate, None, bool)
1402            self._parms["adaptive_rate"] = adaptive_rate
1403
1404        @property
1405        def rho(self):
1406            """
1407            Adaptive learning rate time decay factor (similarity to prior updates).
1408
1409            Type: ``float``, defaults to ``0.99``.
1410
1411            :examples:
1412
1413            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
1414            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
1415            >>> predictors = ["displacement","power","weight","acceleration","year"]
1416            >>> response = "economy_20mpg"
1417            >>> cars_dl = H2ODeepLearningEstimator(rho=0.9,
1418            ...                                    seed=1234)
1419            >>> cars_dl.train(x=predictors,
1420            ...               y=response,
1421            ...               training_frame=cars)
1422            >>> cars_dl.auc()
1423            """
1424            return self._parms.get("rho")
1425
1426        @rho.setter
1427        def rho(self, rho):
1428            assert_is_type(rho, None, numeric)
1429            self._parms["rho"] = rho
1430
1431        @property
1432        def epsilon(self):
1433            """
1434            Adaptive learning rate smoothing factor (to avoid divisions by zero and allow progress)
1435
1436            Type: ``float``, defaults to ``1e-08``.
```

```
1437
1438            :examples:
1439
1440            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
1441            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
1442            >>> predictors = ["displacement","power","weight","acceleration","year"]
1443            >>> response = "cylinders"
1444            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
1445            >>> cars_dl = H2ODeepLearningEstimator(epsilon=1e-6,
1446            ...                                    seed=1234)
1447            >>> cars_dl.train(x=predictors,
1448            ...               y=response,
1449            ...               training_frame=train,
1450            ...               validation_frame=valid)
1451            >>> cars_dl.mse()
1452            """
1453            return self._parms.get("epsilon")
1454
1455        @epsilon.setter
1456        def epsilon(self, epsilon):
1457            assert_is_type(epsilon, None, numeric)
1458            self._parms["epsilon"] = epsilon
1459
1460        @property
1461        def rate(self):
1462            """
1463            Learning rate (higher => less stable, lower => slower convergence).
1464
1465            Type: ``float``, defaults to ``0.005``.
1466
1467            :examples:
1468
1469            >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
1470            >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/laptc
1471            >>> predictors = list(range(0,784))
1472            >>> resp = 784
1473            >>> train[resp] = train[resp].asfactor()
1474            >>> test[resp] = test[resp].asfactor()
1475            >>> nclasses = train[resp].nlevels()[0]
1476            >>> model = H2ODeepLearningEstimator(activation="RectifierWithDropout",
1477            ...                                  adaptive_rate=False,
1478            ...                                  rate=0.01,
1479            ...                                  rate_decay=0.9,
1480            ...                                  rate_annealing=1e-6,
1481            ...                                  momentum_start=0.95,
1482            ...                                  momentum_ramp=1e5,
1483            ...                                  momentum_stable=0.99,
1484            ...                                  nesterov_accelerated_gradient=False,
1485            ...                                  input_dropout_ratio=0.2,
1486            ...                                  train_samples_per_iteration=20000,
1487            ...                                  classification_stop=-1,
1488            ...                                  l1=1e-5)
```

```
1489            >>> model.train (x=predictors,y=resp, training_frame=train, validation_frame=test)
1490            >>> model.model_performance(valid=True)
1491            """
1492            return self._parms.get("rate")
1493
1494        @rate.setter
1495        def rate(self, rate):
1496            assert_is_type(rate, None, numeric)
1497            self._parms["rate"] = rate
1498
1499        @property
1500        def rate_annealing(self):
1501            """
1502            Learning rate annealing: rate / (1 + rate_annealing * samples).
1503
1504            Type: ``float``, defaults to ``1e-06``.
1505
1506            :examples:
1507
1508            >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
1509            >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/laptc
1510            >>> predictors = list(range(0,784))
1511            >>> resp = 784
1512            >>> train[resp] = train[resp].asfactor()
1513            >>> test[resp] = test[resp].asfactor()
1514            >>> nclasses = train[resp].nlevels()[0]
1515            >>> model = H2ODeepLearningEstimator(activation="RectifierWithDropout",
1516            ...                                  adaptive_rate=False,
1517            ...                                  rate=0.01,
1518            ...                                  rate_decay=0.9,
1519            ...                                  rate_annealing=1e-6,
1520            ...                                  momentum_start=0.95,
1521            ...                                  momentum_ramp=1e5,
1522            ...                                  momentum_stable=0.99,
1523            ...                                  nesterov_accelerated_gradient=False,
1524            ...                                  input_dropout_ratio=0.2,
1525            ...                                  train_samples_per_iteration=20000,
1526            ...                                  classification_stop=-1,
1527            ...                                  l1=1e-5)
1528            >>> model.train (x=predictors,
1529            ...              y=resp,
1530            ...              training_frame=train,
1531            ...              validation_frame=test)
1532            >>> model.mse()
1533            """
1534            return self._parms.get("rate_annealing")
1535
1536        @rate_annealing.setter
1537        def rate_annealing(self, rate_annealing):
1538            assert_is_type(rate_annealing, None, numeric)
1539            self._parms["rate_annealing"] = rate_annealing
1540
```

```
    @property
    def rate_decay(self):
        """
        Learning rate decay factor between layers (N-th layer: rate * rate_decay ^ (n - 1).

        Type: ``float``, defaults to ``1.0``.

        :examples:

        >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/laptc
        >>> predictors = list(range(0,784))
        >>> resp = 784
        >>> train[resp] = train[resp].asfactor()
        >>> test[resp] = test[resp].asfactor()
        >>> nclasses = train[resp].nlevels()[0]
        >>> model = H2ODeepLearningEstimator(activation="RectifierWithDropout",
        ...                                  adaptive_rate=False,
        ...                                  rate=0.01,
        ...                                  rate_decay=0.9,
        ...                                  rate_annealing=1e-6,
        ...                                  momentum_start=0.95,
        ...                                  momentum_ramp=1e5,
        ...                                  momentum_stable=0.99,
        ...                                  nesterov_accelerated_gradient=False,
        ...                                  input_dropout_ratio=0.2,
        ...                                  train_samples_per_iteration=20000,
        ...                                  classification_stop=-1,
        ...                                  l1=1e-5)
        >>> model.train (x=predictors,
        ...              y=resp,
        ...              training_frame=train,
        ...              validation_frame=test)
        >>> model.model_performance()
        """
        return self._parms.get("rate_decay")

    @rate_decay.setter
    def rate_decay(self, rate_decay):
        assert_is_type(rate_decay, None, numeric)
        self._parms["rate_decay"] = rate_decay

    @property
    def momentum_start(self):
        """
        Initial momentum at the beginning of training (try 0.5).

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
```

```
        >>> predictors = ["Year","Month","DayofMonth","DayOfWeek","CRSDepTime",
        ...               "CRSArrTime","UniqueCarrier","FlightNum"]
        >>> response_col = "IsDepDelayed"
        >>> airlines_dl = H2ODeepLearningEstimator(hidden=[200,200],
        ...                                        activation="Rectifier",
        ...                                        input_dropout_ratio=0.0,
        ...                                        momentum_start=0.9,
        ...                                        momentum_stable=0.99,
        ...                                        momentum_ramp=1e7,
        ...                                        epochs=100,
        ...                                        stopping_rounds=4,
        ...                                        train_samples_per_iteration=30000,
        ...                                        mini_batch_size=32,
        ...                                        score_duty_cycle=0.25,
        ...                                        score_interval=1)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response_col,
        ...                   training_frame=airlines)
        >>> airlines_dl.mse()
        """
        return self._parms.get("momentum_start")

    @momentum_start.setter
    def momentum_start(self, momentum_start):
        assert_is_type(momentum_start, None, numeric)
        self._parms["momentum_start"] = momentum_start

    @property
    def momentum_ramp(self):
        """
        Number of training samples for which momentum increases.

        Type: ``float``, defaults to ``1000000.0``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> predictors = ["Year","Month","DayofMonth","DayOfWeek","CRSDepTime",
        ...               "CRSArrTime","UniqueCarrier","FlightNum"]
        >>> response_col = "IsDepDelayed"
        >>> airlines_dl = H2ODeepLearningEstimator(hidden=[200,200],
        ...                                        activation="Rectifier",
        ...                                        input_dropout_ratio=0.0,
        ...                                        momentum_start=0.9,
        ...                                        momentum_stable=0.99,
        ...                                        momentum_ramp=1e7,
        ...                                        epochs=100,
        ...                                        stopping_rounds=4,
        ...                                        train_samples_per_iteration=30000,
        ...                                        mini_batch_size=32,
        ...                                        score_duty_cycle=0.25,
        ...                                        score_interval=1)
```

```
        >>> airlines_dl.train(x=predictors,
        ...                   y=response_col,
        ...                   training_frame=airlines)
        >>> airlines_dl.mse()
        """
        return self._parms.get("momentum_ramp")

    @momentum_ramp.setter
    def momentum_ramp(self, momentum_ramp):
        assert_is_type(momentum_ramp, None, numeric)
        self._parms["momentum_ramp"] = momentum_ramp

    @property
    def momentum_stable(self):
        """
        Final momentum after the ramp is over (try 0.99).

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> predictors = ["Year","Month","DayofMonth","DayOfWeek","CRSDepTime",
        ...               "CRSArrTime","UniqueCarrier","FlightNum"]
        >>> response_col = "IsDepDelayed"
        >>> airlines_dl = H2ODeepLearningEstimator(hidden=[200,200],
        ...                                        activation="Rectifier",
        ...                                        input_dropout_ratio=0.0,
        ...                                        momentum_start=0.9,
        ...                                        momentum_stable=0.99,
        ...                                        momentum_ramp=1e7,
        ...                                        epochs=100,
        ...                                        stopping_rounds=4,
        ...                                        train_samples_per_iteration=30000,
        ...                                        mini_batch_size=32,
        ...                                        score_duty_cycle=0.25,
        ...                                        score_interval=1)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response_col,
        ...                   training_frame=airlines)
        >>> airlines_dl.mse()
        """
        return self._parms.get("momentum_stable")

    @momentum_stable.setter
    def momentum_stable(self, momentum_stable):
        assert_is_type(momentum_stable, None, numeric)
        self._parms["momentum_stable"] = momentum_stable

    @property
    def nesterov_accelerated_gradient(self):
        """
```

```
        Use Nesterov accelerated gradient (recommended).

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/laptc
        >>> predictors = list(range(0,784))
        >>> resp = 784
        >>> train[resp] = train[resp].asfactor()
        >>> test[resp] = test[resp].asfactor()
        >>> nclasses = train[resp].nlevels()[0]
        >>> model = H2ODeepLearningEstimator(activation="RectifierWithDropout",
        ...                                  adaptive_rate=False,
        ...                                  rate=0.01,
        ...                                  rate_decay=0.9,
        ...                                  rate_annealing=1e-6,
        ...                                  momentum_start=0.95,
        ...                                  momentum_ramp=1e5,
        ...                                  momentum_stable=0.99,
        ...                                  nesterov_accelerated_gradient=False,
        ...                                  input_dropout_ratio=0.2,
        ...                                  train_samples_per_iteration=20000,
        ...                                  classification_stop=-1,
        ...                                  l1=1e-5)
        >>> model.train (x=predictors,
        ...              y=resp,
        ...              training_frame=train,
        ...              validation_frame=test)
        >>> model.model_performance()
        """
        return self._parms.get("nesterov_accelerated_gradient")

    @nesterov_accelerated_gradient.setter
    def nesterov_accelerated_gradient(self, nesterov_accelerated_gradient):
        assert_is_type(nesterov_accelerated_gradient, None, bool)
        self._parms["nesterov_accelerated_gradient"] = nesterov_accelerated_gradient

    @property
    def input_dropout_ratio(self):
        """
        Input layer dropout ratio (can improve generalization, try 0.1 or 0.2).

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
```

```
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(input_dropout_ratio=0.2,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.auc()
        """
        return self._parms.get("input_dropout_ratio")

    @input_dropout_ratio.setter
    def input_dropout_ratio(self, input_dropout_ratio):
        assert_is_type(input_dropout_ratio, None, numeric)
        self._parms["input_dropout_ratio"] = input_dropout_ratio

    @property
    def hidden_dropout_ratios(self):
        """
        Hidden layer dropout ratios (can improve generalization), specify one value per hidden

        Type: ``List[float]``.

        :examples:

        >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> valid = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> features = list(range(0,784))
        >>> target = 784
        >>> train[target] = train[target].asfactor()
        >>> valid[target] = valid[target].asfactor()
        >>> model = H2ODeepLearningEstimator(epochs=20,
        ...                                  hidden=[200,200],
        ...                                  hidden_dropout_ratios=[0.5,0.5],
        ...                                  seed=1234,
        ...                                  activation='tanhwithdropout')
        >>> model.train(x=features,
        ...             y=target,
        ...             training_frame=train,
        ...             validation_frame=valid)
        >>> model.mse()
        """
        return self._parms.get("hidden_dropout_ratios")

    @hidden_dropout_ratios.setter
    def hidden_dropout_ratios(self, hidden_dropout_ratios):
        assert_is_type(hidden_dropout_ratios, None, [numeric])
        self._parms["hidden_dropout_ratios"] = hidden_dropout_ratios

    @property
    def l1(self):
        """
```

```
        L1 regularization (can add stability and improve generalization, causes many weights to

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> hh_imbalanced = H2ODeepLearningEstimator(l1=1e-5,
        ...                                          activation="Rectifier",
        ...                                          loss="CrossEntropy",
        ...                                          hidden=[200,200],
        ...                                          epochs=1,
        ...                                          balance_classes=False,
        ...                                          reproducible=True,
        ...                                          seed=1234)
        >>> hh_imbalanced.train(x=list(range(54)),y=54, training_frame=covtype)
        >>> hh_imbalanced.mse()
        """
        return self._parms.get("l1")

    @l1.setter
    def l1(self, l1):
        assert_is_type(l1, None, numeric)
        self._parms["l1"] = l1

    @property
    def l2(self):
        """
        L2 regularization (can add stability and improve generalization, causes many weights to

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> hh_imbalanced = H2ODeepLearningEstimator(l2=1e-5,
        ...                                          activation="Rectifier",
        ...                                          loss="CrossEntropy",
        ...                                          hidden=[200,200],
        ...                                          epochs=1,
        ...                                          balance_classes=False,
        ...                                          reproducible=True,
        ...                                          seed=1234)
        >>> hh_imbalanced.train(x=list(range(54)),y=54, training_frame=covtype)
        >>> hh_imbalanced.mse()
        """
        return self._parms.get("l2")

    @l2.setter
    def l2(self, l2):
```

```
        assert_is_type(l2, None, numeric)
        self._parms["l2"] = l2

    @property
    def max_w2(self):
        """
        Constraint for squared sum of incoming weights per unit (e.g. for Rectifier).

        Type: ``float``, defaults to ``3.4028235e+38``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> cov_dl = H2ODeepLearningEstimator(activation="RectifierWithDropout",
        ...                                   hidden=[10,10],
        ...                                   epochs=10,
        ...                                   input_dropout_ratio=0.2,
        ...                                   l1=1e-5,
        ...                                   max_w2=10.5,
        ...                                   stopping_rounds=0)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.mse()
        """
        return self._parms.get("max_w2")

    @max_w2.setter
    def max_w2(self, max_w2):
        assert_is_type(max_w2, None, float)
        self._parms["max_w2"] = max_w2

    @property
    def initial_weight_distribution(self):
        """
        Initial weight distribution.

        Type: ``Literal["uniform_adaptive", "uniform", "normal"]``, defaults to ``"uniform_adap

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(initial_weight_distribution="Uniform",
```

```
        ...                                     seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.auc()
        """
        return self._parms.get("initial_weight_distribution")

    @initial_weight_distribution.setter
    def initial_weight_distribution(self, initial_weight_distribution):
        assert_is_type(initial_weight_distribution, None, Enum("uniform_adaptive", "uniform", "
        self._parms["initial_weight_distribution"] = initial_weight_distribution

    @property
    def initial_weight_scale(self):
        """
        Uniform: -value...value, Normal: stddev.

        Type: ``float``, defaults to ``1.0``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(initial_weight_scale=1.5,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.auc()
        """
        return self._parms.get("initial_weight_scale")

    @initial_weight_scale.setter
    def initial_weight_scale(self, initial_weight_scale):
        assert_is_type(initial_weight_scale, None, numeric)
        self._parms["initial_weight_scale"] = initial_weight_scale

    @property
    def initial_weights(self):
        """
        A list of H2OFrame ids to initialize the weight matrices of this model with.

        Type: ``List[Union[None, str, H2OFrame]]``.

        :examples:
```

```
        >>> iris = h2o.import_file("http://h2o-public-test-data.s3.amazonaws.com/smalldata/iris
        >>> dl1 = H2ODeepLearningEstimator(hidden=[10,10],
        ...                                export_weights_and_biases=True)
        >>> dl1.train(x=list(range(4)), y=4, training_frame=iris)
        >>> p1 = dl1.model_performance(iris).logloss()
        >>> ll1 = dl1.predict(iris)
        >>> print(p1)
        >>> w1 = dl1.weights(0)
        >>> w2 = dl1.weights(1)
        >>> w3 = dl1.weights(2)
        >>> b1 = dl1.biases(0)
        >>> b2 = dl1.biases(1)
        >>> b3 = dl1.biases(2)
        >>> dl2 = H2ODeepLearningEstimator(hidden=[10,10],
        ...                                initial_weights=[w1, w2, w3],
        ...                                initial_biases=[b1, b2, b3],
        ...                                epochs=0)
        >>> dl2.train(x=list(range(4)), y=4, training_frame=iris)
        >>> dl2.initial_weights
        """
        return self._parms.get("initial_weights")

    @initial_weights.setter
    def initial_weights(self, initial_weights):
        assert_is_type(initial_weights, None, [None, str, H2OFrame])
        self._parms["initial_weights"] = initial_weights

    @property
    def initial_biases(self):
        """
        A list of H2OFrame ids to initialize the bias vectors of this model with.

        Type: ``List[Union[None, str, H2OFrame]]``.

        :examples:

        >>> iris = h2o.import_file("http://h2o-public-test-data.s3.amazonaws.com/smalldata/iris
        >>> dl1 = H2ODeepLearningEstimator(hidden=[10,10],
        ...                                export_weights_and_biases=True)
        >>> dl1.train(x=list(range(4)), y=4, training_frame=iris)
        >>> p1 = dl1.model_performance(iris).logloss()
        >>> ll1 = dl1.predict(iris)
        >>> print(p1)
        >>> w1 = dl1.weights(0)
        >>> w2 = dl1.weights(1)
        >>> w3 = dl1.weights(2)
        >>> b1 = dl1.biases(0)
        >>> b2 = dl1.biases(1)
        >>> b3 = dl1.biases(2)
        >>> dl2 = H2ODeepLearningEstimator(hidden=[10,10],
        ...                                initial_weights=[w1, w2, w3],
        ...                                initial_biases=[b1, b2, b3],
```

```
2009            ...                                 epochs=0)
2010            >>> dl2.train(x=list(range(4)), y=4, training_frame=iris)
2011            >>> dl2.initial_biases
2012            """
2013            return self._parms.get("initial_biases")
2014
2015        @initial_biases.setter
2016        def initial_biases(self, initial_biases):
2017            assert_is_type(initial_biases, None, [None, str, H2OFrame])
2018            self._parms["initial_biases"] = initial_biases
2019
2020        @property
2021        def loss(self):
2022            """
2023            Loss function.
2024
2025            Type: ``Literal["automatic", "cross_entropy", "quadratic", "huber", "absolute", "quanti
2026            ``"automatic"``.
2027
2028            :examples:
2029
2030            >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2031            >>> covtype[54] = covtype[54].asfactor()
2032            >>> hh_imbalanced = H2ODeepLearningEstimator(l1=1e-5,
2033            ...                                          activation="Rectifier",
2034            ...                                          loss="CrossEntropy",
2035            ...                                          hidden=[200,200],
2036            ...                                          epochs=1,
2037            ...                                          balance_classes=False,
2038            ...                                          reproducible=True,
2039            ...                                          seed=1234)
2040            >>> hh_imbalanced.train(x=list(range(54)),y=54, training_frame=covtype)
2041            >>> hh_imbalanced.mse()
2042            """
2043            return self._parms.get("loss")
2044
2045        @loss.setter
2046        def loss(self, loss):
2047            assert_is_type(loss, None, Enum("automatic", "cross_entropy", "quadratic", "huber", "al
2048            self._parms["loss"] = loss
2049
2050        @property
2051        def distribution(self):
2052            """
2053            Distribution function
2054
2055            Type: ``Literal["auto", "bernoulli", "multinomial", "gaussian", "poisson", "gamma", "tw
2056            "quantile", "huber"]``, defaults to ``"auto"``.
2057
2058            :examples:
2059
2060            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jun
```

```
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(distribution="poisson",
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("distribution")

    @distribution.setter
    def distribution(self, distribution):
        assert_is_type(distribution, None, Enum("auto", "bernoulli", "multinomial", "gaussian"
        self._parms["distribution"] = distribution

    @property
    def quantile_alpha(self):
        """
        Desired quantile for Quantile regression, must be between 0 and 1.

        Type: ``float``, defaults to ``0.5``.

        :examples:

        >>> boston = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/g
        >>> predictors = boston.columns[:-1]
        >>> response = "medv"
        >>> boston['chas'] = boston['chas'].asfactor()
        >>> train, valid = boston.split_frame(ratios=[.8], seed=1234)
        >>> boston_dl = H2ODeepLearningEstimator(distribution="quantile",
        ...                                      quantile_alpha=.8,
        ...                                      seed=1234)
        >>> boston_dl.train(x=predictors,
        ...                 y=response,
        ...                 training_frame=train,
        ...                 validation_frame=valid)
        >>> boston_dl.mse()
        """
        return self._parms.get("quantile_alpha")

    @quantile_alpha.setter
    def quantile_alpha(self, quantile_alpha):
        assert_is_type(quantile_alpha, None, numeric)
        self._parms["quantile_alpha"] = quantile_alpha

    @property
    def tweedie_power(self):
        """
```

```
2113            Tweedie power for Tweedie regression, must be between 1 and 2.
2114
2115            Type: ``float``, defaults to ``1.5``.
2116
2117            :examples:
2118
2119            >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2120            >>> airlines["Year"]= airlines["Year"].asfactor()
2121            >>> airlines["Month"]= airlines["Month"].asfactor()
2122            >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
2123            >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
2124            >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
2125            >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
2126            ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
2127            >>> response = "IsDepDelayed"
2128            >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
2129            >>> airlines_dl = H2ODeepLearningEstimator(tweedie_power=1.5,
2130            ...                                        seed=1234)
2131            >>> airlines_dl.train(x=predictors,
2132            ...                   y=response,
2133            ...                   training_frame=train,
2134            ...                   validation_frame=valid)
2135            >>> airlines_dl.auc()
2136            """
2137            return self._parms.get("tweedie_power")
2138
2139        @tweedie_power.setter
2140        def tweedie_power(self, tweedie_power):
2141            assert_is_type(tweedie_power, None, numeric)
2142            self._parms["tweedie_power"] = tweedie_power
2143
2144        @property
2145        def huber_alpha(self):
2146            """
2147            Desired quantile for Huber/M-regression (threshold between quadratic and linear loss, r
2148
2149            Type: ``float``, defaults to ``0.9``.
2150
2151            :examples:
2152
2153            >>> insurance = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldat
2154            >>> predictors = insurance.columns[0:4]
2155            >>> response = 'Claims'
2156            >>> insurance['Group'] = insurance['Group'].asfactor()
2157            >>> insurance['Age'] = insurance['Age'].asfactor()
2158            >>> train, valid = insurance.split_frame(ratios=[.8], seed=1234)
2159            >>> insurance_dl = H2ODeepLearningEstimator(distribution="huber",
2160            ...                                         huber_alpha=0.9,
2161            ...                                         seed=1234)
2162            >>> insurance_dl.train(x=predictors,
2163            ...                    y=response,
2164            ...                    training_frame=train,
```

```
2165            ...                     validation_frame=valid)
2166            >>> insurance_dl.mse()
2167            """
2168            return self._parms.get("huber_alpha")
2169
2170        @huber_alpha.setter
2171        def huber_alpha(self, huber_alpha):
2172            assert_is_type(huber_alpha, None, numeric)
2173            self._parms["huber_alpha"] = huber_alpha
2174
2175        @property
2176        def score_interval(self):
2177            """
2178            Shortest time interval (in seconds) between model scoring.
2179
2180            Type: ``float``, defaults to ``5.0``.
2181
2182            :examples:
2183
2184            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2185            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2186            >>> predictors = ["displacement","power","weight","acceleration","year"]
2187            >>> response = "economy_20mpg"
2188            >>> cars_dl = H2ODeepLearningEstimator(score_interval=3,
2189            ...                                    seed=1234)
2190            >>> cars_dl.train(x=predictors,
2191            ...               y=response,
2192            ...               training_frame=cars)
2193            >>> cars_dl.auc()
2194            """
2195            return self._parms.get("score_interval")
2196
2197        @score_interval.setter
2198        def score_interval(self, score_interval):
2199            assert_is_type(score_interval, None, numeric)
2200            self._parms["score_interval"] = score_interval
2201
2202        @property
2203        def score_training_samples(self):
2204            """
2205            Number of training set samples for scoring (0 for all).
2206
2207            Type: ``int``, defaults to ``10000``.
2208
2209            :examples:
2210
2211            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2212            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2213            >>> predictors = ["displacement","power","weight","acceleration","year"]
2214            >>> response = "economy_20mpg"
2215            >>> cars_dl = H2ODeepLearningEstimator(score_training_samples=10000,
2216            ...                                    seed=1234)
```

```
2217            >>> cars_dl.train(x=predictors,
2218            ...               y=response,
2219            ...               training_frame=cars)
2220            >>> cars_dl.auc()
2221            """
2222            return self._parms.get("score_training_samples")
2223
2224        @score_training_samples.setter
2225        def score_training_samples(self, score_training_samples):
2226            assert_is_type(score_training_samples, None, int)
2227            self._parms["score_training_samples"] = score_training_samples
2228
2229        @property
2230        def score_validation_samples(self):
2231            """
2232            Number of validation set samples for scoring (0 for all).
2233
2234            Type: ``int``, defaults to ``0``.
2235
2236            :examples:
2237
2238            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2239            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2240            >>> predictors = ["displacement","power","weight","acceleration","year"]
2241            >>> response = "economy_20mpg"
2242            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
2243            >>> cars_dl = H2ODeepLearningEstimator(score_validation_samples=3,
2244            ...                                    seed=1234)
2245            >>> cars_dl.train(x=predictors,
2246            ...               y=response,
2247            ...               training_frame=train,
2248            ...               validation_frame=valid)
2249            >>> cars_dl.auc()
2250            """
2251            return self._parms.get("score_validation_samples")
2252
2253        @score_validation_samples.setter
2254        def score_validation_samples(self, score_validation_samples):
2255            assert_is_type(score_validation_samples, None, int)
2256            self._parms["score_validation_samples"] = score_validation_samples
2257
2258        @property
2259        def score_duty_cycle(self):
2260            """
2261            Maximum duty cycle fraction for scoring (lower: more training, higher: more scoring).
2262
2263            Type: ``float``, defaults to ``0.1``.
2264
2265            :examples:
2266
2267            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2268            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
```

```
2269            >>> predictors = ["displacement","power","weight","acceleration","year"]
2270            >>> response = "economy_20mpg"
2271            >>> cars_dl = H2ODeepLearningEstimator(score_duty_cycle=0.2,
2272            ...                                    seed=1234)
2273            >>> cars_dl.train(x=predictors,
2274            ...               y=response,
2275            ...               training_frame=cars)
2276            >>> cars_dl.auc()
2277            """
2278            return self._parms.get("score_duty_cycle")
2279
2280        @score_duty_cycle.setter
2281        def score_duty_cycle(self, score_duty_cycle):
2282            assert_is_type(score_duty_cycle, None, numeric)
2283            self._parms["score_duty_cycle"] = score_duty_cycle
2284
2285        @property
2286        def classification_stop(self):
2287            """
2288            Stopping criterion for classification error fraction on training data (-1 to disable).
2289
2290            Type: ``float``, defaults to ``0.0``.
2291
2292            :examples:
2293
2294            >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2295            >>> covtype[54] = covtype[54].asfactor()
2296            >>> predictors = covtype.columns[0:54]
2297            >>> response = 'C55'
2298            >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
2299            >>> cars_dl = H2ODeepLearningEstimator(classification_stop=1.5,
2300            ...                                    seed=1234)
2301            >>> cov_dl.train(x=predictors,
2302            ...              y=response,
2303            ...              training_frame=train,
2304            ...              validation_frame=valid)
2305            >>> cov_dl.mse()
2306            """
2307            return self._parms.get("classification_stop")
2308
2309        @classification_stop.setter
2310        def classification_stop(self, classification_stop):
2311            assert_is_type(classification_stop, None, numeric)
2312            self._parms["classification_stop"] = classification_stop
2313
2314        @property
2315        def regression_stop(self):
2316            """
2317            Stopping criterion for regression error (MSE) on training data (-1 to disable).
2318
2319            Type: ``float``, defaults to ``1e-06``.
2320
```

```
        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...              "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(regression_stop=1e-6,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("regression_stop")

    @regression_stop.setter
    def regression_stop(self, regression_stop):
        assert_is_type(regression_stop, None, numeric)
        self._parms["regression_stop"] = regression_stop

    @property
    def stopping_rounds(self):
        """
        Early stopping based on convergence of stopping_metric. Stop if simple moving average o
        stopping_metric does not improve for k:=stopping_rounds scoring events (0 to disable)

        Type: ``int``, defaults to ``5``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...              "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(stopping_metric="auc",
        ...                                        stopping_rounds=3,
        ...                                        stopping_tolerance=1e-2,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
```

```
        ...                    y=response,
        ...                    training_frame=train,
        ...                    validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("stopping_rounds")

    @stopping_rounds.setter
    def stopping_rounds(self, stopping_rounds):
        assert_is_type(stopping_rounds, None, int)
        self._parms["stopping_rounds"] = stopping_rounds

    @property
    def stopping_metric(self):
        """
        Metric to use for early stopping (AUTO: logloss for classification, deviance for regres
        for Isolation Forest). Note that custom and custom_increasing can only be used in GBM a
        client.

        Type: ``Literal["auto", "deviance", "logloss", "mse", "rmse", "mae", "rmsle", "auc", "a
        "misclassification", "mean_per_class_error", "custom", "custom_increasing"]``, defaults

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(stopping_metric="auc",
        ...                                        stopping_rounds=3,
        ...                                        stopping_tolerance=1e-2,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("stopping_metric")

    @stopping_metric.setter
    def stopping_metric(self, stopping_metric):
        assert_is_type(stopping_metric, None, Enum("auto", "deviance", "logloss", "mse", "rmse
        self._parms["stopping_metric"] = stopping_metric

    @property
```

```
    def stopping_tolerance(self):
        """
        Relative tolerance for metric-based stopping criterion (stop if relative improvement is

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(stopping_metric="auc",
        ...                                        stopping_rounds=3,
        ...                                        stopping_tolerance=1e-2,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.auc()
        """
        return self._parms.get("stopping_tolerance")

    @stopping_tolerance.setter
    def stopping_tolerance(self, stopping_tolerance):
        assert_is_type(stopping_tolerance, None, numeric)
        self._parms["stopping_tolerance"] = stopping_tolerance

    @property
    def max_runtime_secs(self):
        """
        Maximum allowed runtime in seconds for model training. Use 0 to disable.

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(max_runtime_secs=10,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
```

```
2477            ...              y=response,
2478            ...              training_frame=train,
2479            ...              validation_frame=valid)
2480            >>> cars_dl.auc()
2481            """
2482            return self._parms.get("max_runtime_secs")
2483
2484        @max_runtime_secs.setter
2485        def max_runtime_secs(self, max_runtime_secs):
2486            assert_is_type(max_runtime_secs, None, numeric)
2487            self._parms["max_runtime_secs"] = max_runtime_secs
2488
2489        @property
2490        def score_validation_sampling(self):
2491            """
2492            Method used to sample validation dataset for scoring.
2493
2494            Type: ``Literal["uniform", "stratified"]``, defaults to ``"uniform"``.
2495
2496            :examples:
2497
2498            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2499            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2500            >>> predictors = ["displacement","power","weight","acceleration","year"]
2501            >>> response = "economy_20mpg"
2502            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
2503            >>> cars_dl = H2ODeepLearningEstimator(score_validation_sampling="uniform",
2504            ...                                    seed=1234)
2505            >>> cars_dl.train(x=predictors,
2506            ...               y=response,
2507            ...               training_frame=train,
2508            ...               validation_frame=valid)
2509            >>> cars_dl.auc()
2510            """
2511            return self._parms.get("score_validation_sampling")
2512
2513        @score_validation_sampling.setter
2514        def score_validation_sampling(self, score_validation_sampling):
2515            assert_is_type(score_validation_sampling, None, Enum("uniform", "stratified"))
2516            self._parms["score_validation_sampling"] = score_validation_sampling
2517
2518        @property
2519        def diagnostics(self):
2520            """
2521            Enable diagnostics for hidden layers.
2522
2523            Type: ``bool``, defaults to ``True``.
2524
2525            :examples:
2526
2527            >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2528            >>> covtype[54] = covtype[54].asfactor()
```

```
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(diagnostics=True,
        ...                                    seed=1234)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.mse()
        """
        return self._parms.get("diagnostics")

    @diagnostics.setter
    def diagnostics(self, diagnostics):
        assert_is_type(diagnostics, None, bool)
        self._parms["diagnostics"] = diagnostics

    @property
    def fast_mode(self):
        """
        Enable fast mode (minor approximation in back-propagation).

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(fast_mode=False,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("fast_mode")

    @fast_mode.setter
    def fast_mode(self, fast_mode):
        assert_is_type(fast_mode, None, bool)
        self._parms["fast_mode"] = fast_mode

    @property
    def force_load_balance(self):
        """
        Force extra load balancing to increase training speed for small datasets (to keep all c

```

```
        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(force_load_balance=False,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("force_load_balance")

    @force_load_balance.setter
    def force_load_balance(self, force_load_balance):
        assert_is_type(force_load_balance, None, bool)
        self._parms["force_load_balance"] = force_load_balance

    @property
    def variable_importances(self):
        """
        Compute variable importances for input features (Gedeon method) - can be slow for large

        Type: ``bool``, defaults to ``True``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> airlines_dl = H2ODeepLearningEstimator(variable_importances=True,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.mse()
        """
        return self._parms.get("variable_importances")
```

```
2633
2634        @variable_importances.setter
2635        def variable_importances(self, variable_importances):
2636            assert_is_type(variable_importances, None, bool)
2637            self._parms["variable_importances"] = variable_importances
2638
2639        @property
2640        def replicate_training_data(self):
2641            """
2642            Replicate the entire training dataset onto every node for faster training on small data
2643
2644            Type: ``bool``, defaults to ``True``.
2645
2646            :examples:
2647
2648            >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2649            >>> airlines["Year"]= airlines["Year"].asfactor()
2650            >>> airlines["Month"]= airlines["Month"].asfactor()
2651            >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
2652            >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
2653            >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
2654            >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
2655            ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
2656            >>> response = "IsDepDelayed"
2657            >>> airlines_dl = H2ODeepLearningEstimator(replicate_training_data=False)
2658            >>> airlines_dl.train(x=predictors,
2659            ...                   y=response,
2660            ...                   training_frame=airlines)
2661            >>> airlines_dl.auc()
2662            """
2663            return self._parms.get("replicate_training_data")
2664
2665        @replicate_training_data.setter
2666        def replicate_training_data(self, replicate_training_data):
2667            assert_is_type(replicate_training_data, None, bool)
2668            self._parms["replicate_training_data"] = replicate_training_data
2669
2670        @property
2671        def single_node_mode(self):
2672            """
2673            Run on a single node for fine-tuning of model parameters.
2674
2675            Type: ``bool``, defaults to ``False``.
2676
2677            :examples:
2678
2679            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2680            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2681            >>> predictors = ["displacement","power","weight","acceleration","year"]
2682            >>> response = "economy_20mpg"
2683            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
2684            >>> cars_dl = H2ODeepLearningEstimator(single_node_mode=True,
```

```
        ...                                       seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> cars_dl.auc()
        """
        return self._parms.get("single_node_mode")

    @single_node_mode.setter
    def single_node_mode(self, single_node_mode):
        assert_is_type(single_node_mode, None, bool)
        self._parms["single_node_mode"] = single_node_mode

    @property
    def shuffle_training_data(self):
        """
        Enable shuffling of training data (recommended if training data is replicated and trai
        close to #nodes x #rows, of if using balance_classes).

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/ju
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "economy_20mpg"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(shuffle_training_data=True,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=cars)
        >>> cars_dl.auc()
        """
        return self._parms.get("shuffle_training_data")

    @shuffle_training_data.setter
    def shuffle_training_data(self, shuffle_training_data):
        assert_is_type(shuffle_training_data, None, bool)
        self._parms["shuffle_training_data"] = shuffle_training_data

    @property
    def missing_values_handling(self):
        """
        Handling of missing values. Either MeanImputation or Skip.

        Type: ``Literal["mean_imputation", "skip"]``, defaults to ``"mean_imputation"``.

        :examples:

        >>> boston = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/g
```

```
        >>> predictors = boston.columns[:-1]
        >>> response = "medv"
        >>> boston['chas'] = boston['chas'].asfactor()
        >>> boston.insert_missing_values()
        >>> train, valid = boston.split_frame(ratios=[.8])
        >>> boston_dl = H2ODeepLearningEstimator(missing_values_handling="skip")
        >>> boston_dl.train(x=predictors,
        ...                 y=response,
        ...                 training_frame=train,
        ...                 validation_frame=valid)
        >>> boston_dl.mse()
        """
        return self._parms.get("missing_values_handling")

    @missing_values_handling.setter
    def missing_values_handling(self, missing_values_handling):
        assert_is_type(missing_values_handling, None, Enum("mean_imputation", "skip"))
        self._parms["missing_values_handling"] = missing_values_handling

    @property
    def quiet_mode(self):
        """
        Enable quiet mode for less output to standard output.

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> titanic = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> titanic['survived'] = titanic['survived'].asfactor()
        >>> predictors = titanic.columns
        >>> del predictors[1:3]
        >>> response = 'survived'
        >>> train, valid = titanic.split_frame(ratios=[.8], seed=1234)
        >>> titanic_dl = H2ODeepLearningEstimator(quiet_mode=True,
        ...                                       seed=1234)
        >>> titanic_dl.train(x=predictors,
        ...                  y=response,
        ...                  training_frame=train,
        ...                  validation_frame=valid)
        >>> titanic_dl.mse()
        """
        return self._parms.get("quiet_mode")

    @quiet_mode.setter
    def quiet_mode(self, quiet_mode):
        assert_is_type(quiet_mode, None, bool)
        self._parms["quiet_mode"] = quiet_mode

    @property
    def autoencoder(self):
        """
```

```
2789            Auto-Encoder.
2790
2791            Type: ``bool``, defaults to ``False``.
2792
2793            :examples:
2794
2795            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2796            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2797            >>> predictors = ["displacement","power","weight","acceleration","year"]
2798            >>> response = "cylinders"
2799            >>> cars_dl = H2ODeepLearningEstimator(autoencoder=True)
2800            >>> cars_dl.train(x=predictors,
2801            ...               y=response,
2802            ...               training_frame=train,
2803            ...               validation_frame=valid)
2804            >>> cars_dl.mse()
2805            """
2806            return self._parms.get("autoencoder")
2807
2808        @autoencoder.setter
2809        def autoencoder(self, autoencoder):
2810            assert_is_type(autoencoder, None, bool)
2811            self._parms["autoencoder"] = autoencoder
2812
2813        @property
2814        def sparse(self):
2815            """
2816            Sparse data handling (more efficient for data with lots of 0 values).
2817
2818            Type: ``bool``, defaults to ``False``.
2819
2820            :examples:
2821
2822            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2823            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2824            >>> predictors = ["displacement","power","weight","acceleration","year"]
2825            >>> response = "economy_20mpg"
2826            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
2827            >>> cars_dl = H2ODeepLearningEstimator(sparse=True,
2828            ...                                    seed=1234)
2829            >>> cars_dl.train(x=predictors,
2830            ...               y=response,
2831            ...               training_frame=cars)
2832            >>> cars_dl.auc()
2833            """
2834            return self._parms.get("sparse")
2835
2836        @sparse.setter
2837        def sparse(self, sparse):
2838            assert_is_type(sparse, None, bool)
2839            self._parms["sparse"] = sparse
2840
```

```
    @property
    def col_major(self):
        """
        #DEPRECATED Use a column major weight matrix for input layer. Can speed up forward prop
        down backpropagation.

        Type: ``bool``, defaults to ``False``.
        """
        return self._parms.get("col_major")

    @col_major.setter
    def col_major(self, col_major):
        assert_is_type(col_major, None, bool)
        self._parms["col_major"] = col_major

    @property
    def average_activation(self):
        """
        Average activation for sparse auto-encoder. #Experimental

        Type: ``float``, defaults to ``0.0``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> cars_dl = H2ODeepLearningEstimator(average_activation=1.5,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("average_activation")

    @average_activation.setter
    def average_activation(self, average_activation):
        assert_is_type(average_activation, None, numeric)
        self._parms["average_activation"] = average_activation

    @property
    def sparsity_beta(self):
        """
        Sparsity regularization. #Experimental

        Type: ``float``, defaults to ``0.0``.

        :examples:
```

```
        >>> from h2o.estimators import H2OAutoEncoderEstimator
        >>> resp = 784
        >>> nfeatures = 20
        >>> train = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapt
        >>> test = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/bigdata/lapto
        >>> train[resp] = train[resp].asfactor()
        >>> test[resp] = test[resp].asfactor()
        >>> sid = train[0].runif(0)
        >>> train_unsupervised = train[sid>=0.5]
        >>> train_unsupervised.pop(resp)
        >>> ae_model = H2OAutoEncoderEstimator(activation="Tanh",
        ...                                    hidden=[nfeatures],
        ...                                    epochs=1,
        ...                                    ignore_const_cols=False,
        ...                                    reproducible=True,
        ...                                    sparsity_beta=0.5,
        ...                                    seed=1234)
        >>> ae_model.train(list(range(resp)),
        ...                training_frame=train_unsupervised)
        >>> ae_model.mse()
        """
        return self._parms.get("sparsity_beta")

    @sparsity_beta.setter
    def sparsity_beta(self, sparsity_beta):
        assert_is_type(sparsity_beta, None, numeric)
        self._parms["sparsity_beta"] = sparsity_beta

    @property
    def max_categorical_features(self):
        """
        Max. number of categorical features, enforced via hashing. #Experimental

        Type: ``int``, defaults to ``2147483647``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> cov_dl = H2ODeepLearningEstimator(balance_classes=True,
        ...                                   max_categorical_features=2147483647,
        ...                                   seed=1234)
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.logloss()
        """
        return self._parms.get("max_categorical_features")
```

```
2945
2946        @max_categorical_features.setter
2947        def max_categorical_features(self, max_categorical_features):
2948            assert_is_type(max_categorical_features, None, int)
2949            self._parms["max_categorical_features"] = max_categorical_features
2950
2951        @property
2952        def reproducible(self):
2953            """
2954            Force reproducibility on small data (will be slow - only uses 1 thread).
2955
2956            Type: ``bool``, defaults to ``False``.
2957
2958            :examples:
2959
2960            >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
2961            >>> airlines["Year"]= airlines["Year"].asfactor()
2962            >>> airlines["Month"]= airlines["Month"].asfactor()
2963            >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
2964            >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
2965            >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
2966            >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
2967            ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
2968            >>> response = "IsDepDelayed"
2969            >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
2970            >>> airlines_dl = H2ODeepLearningEstimator(reproducible=True)
2971            >>> airlines_dl.train(x=predictors,
2972            ...                   y=response,
2973            ...                   training_frame=train,
2974            ...                   validation_frame=valid)
2975            >>> airlines_dl.auc()
2976            """
2977            return self._parms.get("reproducible")
2978
2979        @reproducible.setter
2980        def reproducible(self, reproducible):
2981            assert_is_type(reproducible, None, bool)
2982            self._parms["reproducible"] = reproducible
2983
2984        @property
2985        def export_weights_and_biases(self):
2986            """
2987            Whether to export Neural Network weights and biases to H2O Frames.
2988
2989            Type: ``bool``, defaults to ``False``.
2990
2991            :examples:
2992
2993            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
2994            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
2995            >>> predictors = ["displacement","power","weight","acceleration","year"]
2996            >>> response = "cylinders"
```

```
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(export_weights_and_biases=True,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("export_weights_and_biases")

    @export_weights_and_biases.setter
    def export_weights_and_biases(self, export_weights_and_biases):
        assert_is_type(export_weights_and_biases, None, bool)
        self._parms["export_weights_and_biases"] = export_weights_and_biases

    @property
    def mini_batch_size(self):
        """
        Mini-batch size (smaller leads to better fit, larger can speed up and generalize better

        Type: ``int``, defaults to ``1``.

        :examples:

        >>> covtype = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> covtype[54] = covtype[54].asfactor()
        >>> predictors = covtype.columns[0:54]
        >>> response = 'C55'
        >>> train, valid = covtype.split_frame(ratios=[.8], seed=1234)
        >>> cov_dl = H2ODeepLearningEstimator(activation="RectifierWithDropout",
        ...                                   hidden=[10,10],
        ...                                   epochs=10,
        ...                                   input_dropout_ratio=0.2,
        ...                                   l1=1e-5,
        ...                                   max_w2=10.5,
        ...                                   stopping_rounds=0)
        ...                                   mini_batch_size=35
        >>> cov_dl.train(x=predictors,
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cov_dl.mse()
        """
        return self._parms.get("mini_batch_size")

    @mini_batch_size.setter
    def mini_batch_size(self, mini_batch_size):
        assert_is_type(mini_batch_size, None, int)
        self._parms["mini_batch_size"] = mini_batch_size

    @property
```

```
    def categorical_encoding(self):
        """
        Encoding scheme for categorical features

        Type: ``Literal["auto", "enum", "one_hot_internal", "one_hot_explicit", "binary", "eige
        "sort_by_response", "enum_limited"]``, defaults to ``"auto"``.

        :examples:

        >>> airlines= h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/
        >>> airlines["Year"]= airlines["Year"].asfactor()
        >>> airlines["Month"]= airlines["Month"].asfactor()
        >>> airlines["DayOfWeek"] = airlines["DayOfWeek"].asfactor()
        >>> airlines["Cancelled"] = airlines["Cancelled"].asfactor()
        >>> airlines['FlightNum'] = airlines['FlightNum'].asfactor()
        >>> predictors = ["Origin", "Dest", "Year", "UniqueCarrier",
        ...               "DayOfWeek", "Month", "Distance", "FlightNum"]
        >>> response = "IsDepDelayed"
        >>> train, valid= airlines.split_frame(ratios=[.8], seed=1234)
        >>> encoding = "one_hot_internal"
        >>> airlines_dl = H2ODeepLearningEstimator(categorical_encoding=encoding,
        ...                                        seed=1234)
        >>> airlines_dl.train(x=predictors,
        ...                   y=response,
        ...                   training_frame=train,
        ...                   validation_frame=valid)
        >>> airlines_dl.mse()
        """
        return self._parms.get("categorical_encoding")

    @categorical_encoding.setter
    def categorical_encoding(self, categorical_encoding):
        assert_is_type(categorical_encoding, None, Enum("auto", "enum", "one_hot_internal", "o
        self._parms["categorical_encoding"] = categorical_encoding

    @property
    def elastic_averaging(self):
        """
        Elastic averaging between compute nodes can improve distributed model convergence. #Exp

        Type: ``bool``, defaults to ``False``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(elastic_averaging=True,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
```

```
        ...              y=response,
        ...              training_frame=train,
        ...              validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("elastic_averaging")

    @elastic_averaging.setter
    def elastic_averaging(self, elastic_averaging):
        assert_is_type(elastic_averaging, None, bool)
        self._parms["elastic_averaging"] = elastic_averaging

    @property
    def elastic_averaging_moving_rate(self):
        """
        Elastic averaging moving rate (only if elastic averaging is enabled).

        Type: ``float``, defaults to ``0.9``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
        >>> predictors = ["displacement","power","weight","acceleration","year"]
        >>> response = "cylinders"
        >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
        >>> cars_dl = H2ODeepLearningEstimator(elastic_averaging_moving_rate=.8,
        ...                                    seed=1234)
        >>> cars_dl.train(x=predictors,
        ...               y=response,
        ...               training_frame=train,
        ...               validation_frame=valid)
        >>> cars_dl.mse()
        """
        return self._parms.get("elastic_averaging_moving_rate")

    @elastic_averaging_moving_rate.setter
    def elastic_averaging_moving_rate(self, elastic_averaging_moving_rate):
        assert_is_type(elastic_averaging_moving_rate, None, numeric)
        self._parms["elastic_averaging_moving_rate"] = elastic_averaging_moving_rate

    @property
    def elastic_averaging_regularization(self):
        """
        Elastic averaging regularization strength (only if elastic averaging is enabled).

        Type: ``float``, defaults to ``0.001``.

        :examples:

        >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
        >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
```

```
3153            >>> predictors = ["displacement","power","weight","acceleration","year"]
3154            >>> response = "cylinders"
3155            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
3156            >>> cars_dl = H2ODeepLearningEstimator(elastic_averaging_regularization=.008,
3157            ...                                    seed=1234)
3158            >>> cars_dl.train(x=predictors,
3159            ...               y=response,
3160            ...               training_frame=train,
3161            ...               validation_frame=valid)
3162            >>> cars_dl.mse()
3163            """
3164            return self._parms.get("elastic_averaging_regularization")
3165
3166        @elastic_averaging_regularization.setter
3167        def elastic_averaging_regularization(self, elastic_averaging_regularization):
3168            assert_is_type(elastic_averaging_regularization, None, numeric)
3169            self._parms["elastic_averaging_regularization"] = elastic_averaging_regularization
3170
3171        @property
3172        def export_checkpoints_dir(self):
3173            """
3174            Automatically export generated models to this directory.
3175
3176            Type: ``str``.
3177
3178            :examples:
3179
3180            >>> import tempfile
3181            >>> from os import listdir
3182            >>> cars = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-data/smalldata/jur
3183            >>> cars["economy_20mpg"] = cars["economy_20mpg"].asfactor()
3184            >>> predictors = ["displacement","power","weight","acceleration","year"]
3185            >>> response = "cylinders"
3186            >>> train, valid = cars.split_frame(ratios=[.8], seed=1234)
3187            >>> checkpoints_dir = tempfile.mkdtemp()
3188            >>> cars_dl = H2ODeepLearningEstimator(export_checkpoints_dir=checkpoints_dir,
3189            ...                                    seed=1234)
3190            >>> cars_dl.train(x=predictors,
3191            ...               y=response,
3192            ...               training_frame=train,
3193            ...               validation_frame=valid)
3194            >>> len(listdir(checkpoints_dir))
3195            """
3196            return self._parms.get("export_checkpoints_dir")
3197
3198        @export_checkpoints_dir.setter
3199        def export_checkpoints_dir(self, export_checkpoints_dir):
3200            assert_is_type(export_checkpoints_dir, None, str)
3201            self._parms["export_checkpoints_dir"] = export_checkpoints_dir
3202
3203        @property
3204        def auc_type(self):
```

```
3205            """
3206            Set default multinomial AUC type.
3207
3208            Type: ``Literal["auto", "none", "macro_ovr", "weighted_ovr", "macro_ovo", "weighted_ovo
3209            ``"auto"``.
3210            """
3211            return self._parms.get("auc_type")
3212
3213        @auc_type.setter
3214        def auc_type(self, auc_type):
3215            assert_is_type(auc_type, None, Enum("auto", "none", "macro_ovr", "weighted_ovr", "macro
3216            self._parms["auc_type"] = auc_type
3217
3218
3219
3220    class H2OAutoEncoderEstimator(H2ODeepLearningEstimator):
3221        """
3222        :examples:
3223
3224        >>> import h2o as ml
3225        >>> from h2o.estimators.deeplearning import H2OAutoEncoderEstimator
3226        >>> ml.init()
3227        >>> rows = [[1,2,3,4,0]*50, [2,1,2,4,1]*50, [2,1,4,2,1]*50, [0,1,2,34,1]*50, [2,3,4,1,0]*50
3228        >>> fr = ml.H2OFrame(rows)
3229        >>> fr[4] = fr[4].asfactor()
3230        >>> model = H2OAutoEncoderEstimator()
3231        >>> model.train(x=range(4), training_frame=fr)
3232        """
3233
3234        supervised_learning = False
3235
3236        def __init__(self, **kwargs):
3237            super(H2OAutoEncoderEstimator, self).__init__(**kwargs)
3238            self._parms['autoencoder'] = True
```