Deep Learning (Neural Networks)

Introduction

H2O's Deep Learning is based on a multi-layer feedforward artificial neural network that is trained with stochastic gradient descent using back-propagation. The network can contain a large number of hidden layers consisting of neurons with tanh, rectifier, and maxout activation functions. Advanced features such as adaptive learning rate, rate annealing, momentum training, dropout, L1 or L2 regularization, checkpointing, and grid search enable high predictive accuracy. Each compute node trains a copy of the global model parameters on its local data with multi-threading (asynchronously) and contributes periodically to the global model via model averaging across the network.

A feedforward artificial neural network (ANN) model, also known as deep neural network (DNN) or multi-layer perceptron (MLP), is the most common type of Deep Neural Network and the only type that is supported natively in H2O-3. Several other types of DNNs are popular as well, such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). MLPs work well on transactional (tabular) data; however if you have image data, then CNNs are a great choice. If you have sequential data (e.g. text, audio, time-series), then RNNs are a good choice.

Quick Start and Additional Resources

- Deep Learning Booklet
- Deep Learning in H2O Tutorial (R): [GitHub]
- H2O + TensorFlow on AWS GPU Tutorial (Python Notebook) [Blog] [Github]
- Deep learning in H2O with Arno Candel (Overview) [Youtube]
- Top 10 tips and tricks [Youtube]
- NYC Tour Deep Learning Panel: Tensorflow, Mxnet, Caffe [Youtube]

Defining a Deep Learning Model

H2O Deep Learning models have many input parameters, many of which are only accessible via the expert mode. For most cases, use the default values. Please read the following instructions before building extensive Deep Learning models. The application of grid search and successive continuation of winning models via checkpoint restart is highly recommended, as model performance can vary greatly.

- model_id: (Optional) Specify a custom name for the model to use as a reference. By default, H2O automatically generates a destination key.
- training_frame: (Required) Specify the dataset used to build the model. **NOTE**: In Flow, if you click the **Build a model** button from the `Parse` cell, the training frame is entered automatically.
- validation_frame: (Optional) Specify the dataset used to evaluate the accuracy of the model.
- nfolds: Specify the number of folds for cross-validation. This option defaults to 0 (no cross-validation).

    Note: Cross-validation is not supported when **autoencoder** is enabled.

- keep_cross_validation_models: Specify whether to keep the cross-validated models. Keeping cross-validation models may consume significantly more memory in the H2O cluster. This option defaults to true.
- keep_cross_validation_predictions: Enable this option to keep the cross-validation predictions. This option defaults to false.
- keep_cross_validation_fold_assignment: Enable this option to preserve the cross-validation fold assignment. This option defaults to false.
- y: Specify the column to use as the dependent variable. The data can be numeric or categorical.
- x: Specify a vector containing the names or indices of the predictor variables to use when building the model. If `x` is missing, then all columns except `y` are used.
- fold_assignment: (Applicable only if a value for **nfolds** is specified and **fold_column** is not specified) Specify the cross-validation fold assignment scheme. The available options are AUTO (which is Random), Random, Modulo, or Stratified (which will stratify the folds based on the response variable for classification problems). This option defaults to AUTO.
- fold_column: Specify the column that contains the cross-validation fold index assignment per observation.
- ignored_columns: (Optional, Python and Flow only) Specify the column or columns to be excluded from the model. In Flow, click the checkbox next to a column name to add it to the list of columns excluded from the model. To add all columns, click the **All** button. To remove a column from the list of ignored columns, click the X next to the column name. To remove all columns from the list of ignored columns, click the **None** button. To search for a specific column, type the column name in the **Search** field above the column list. To only show columns with a specific percentage of missing values, specify the percentage in the **Only show columns with more than 0% missing values** field. To change the selections for the hidden columns, use the **Select Visible** or **Deselect Visible** buttons.
- ignore_const_cols: Specify whether to ignore constant training columns, since no information can be gained from them. This option is enabled by default.
- score_each_iteration: (Optional) Specify whether to score during each iteration of the model training. This option defaults to false.
- weights_column: Specify a column to use for the observation weights, which are used for bias correction. The specified `weights_column` must be included in the specified `training_frame`.

Python only: To use a weights column when passing an H2OFrame to `x` instead of a list of column names, the specified `training_frame` must contain the specified `weights_column`.

Note: Weights are per-row observation weights. This is typically the number of times a row is repeated, but non-integer values are supported as well. During training, rows with higher weights matter more, due to the larger loss function pre-factor.

- offset_column: (Applicable for regression only) Specify a column to use as the offset.

  Note: Offsets are per-row "bias values" that are used during model training. For Gaussian distributions, they can be seen as simple corrections to the response (y) column. Instead of learning to predict the response (y-row), the model learns to predict the (row) offset of the response column. For other distributions, the offset corrections are applied in the linearized space before applying the inverse link function to get the actual response values.

- balance_classes: (Applicable for classification only) Specify whether to oversample the minority classes to balance the class distribution. This option is not enabled by default and can increase the data frame size. This option is only applicable for classification. Majority classes can be undersampled to satisfy the `max_after_balance_size` parameter. This option defaults to false.
- class_sampling_factors: (Applicable only for classification and when `balance_classes` is enabled) Specify the per-class (in lexicographical order) over/under-sampling ratios. By default, these ratios are automatically computed during training to obtain the class balance.
- max_after_balance_size: Specify the maximum relative size of the training data after balancing class counts (**balance_classes** must be enabled). The value can be less than 1.0. This option defaults to 5.0.
- **max_confusion_matrix_size**: This option is deprecated and will be removed in a future release.
- checkpoint: Enter a model key associated with a previously-trained Deep Learning model. Use this option to build a new model as a continuation of a previously-generated model.

  Note: Cross-validation is not supported during checkpoint restarts.

- **pretrained_ autoencoder** : Specify a pretrained **autoencoder** model to initialize this model with.
- **overwrite_with_best_model**: Specify whether to overwrite the final model with the best model found during training, based on the option specified for **stopping_metric**. This option is enabled by default.
- use_all_factor_levels: Specify whether to use all factor levels in the possible set of predictors; if you enable this option, sufficient regularization is required. By default, the first factor level is skipped. For Deep Learning models, this option is useful for determining variable importances and is automatically enabled if the **autoencoder** is selected. This option is true by default.

- standardize: If enabled, automatically standardize the data (mean 0, variance 1). If disabled, the user must provide properly scaled input data. This option defaults to true.
- **activation**: Specify the activation function (Tanh, Tanh with dropout, Rectifier, Rectifier with dropout, Maxout, Maxout with dropout). This option defaults to Rectifier.

  Note: **Maxout** is not supported when **autoencoder** is enabled.

- **hidden**: Specify the hidden layer sizes (e.g., 100,100). The value must be positive. This option defaults to (200,200).
- **epochs**: Specify the number of times to iterate (stream) the dataset. The value can be a fraction. This option defaults to 10.
- **train_samples_per_iteration**: Specify the number of global training samples per MapReduce iteration. To specify one epoch, enter 0. To specify all available data (e.g., replicated training data), enter -1. To use the automatic (default) values, enter -2.
- **target_ratio_comm_to_comp**: Specify the target ratio of communication overhead to computation. This option is only enabled for multi-node operation and if **train_samples_per_iteration** equals -2 (auto-tuning). This option defaults to 0.05.
- **seed**: Specify the random number generator (RNG) seed for algorithm components dependent on randomization. The seed is consistent for each H2O instance so that you can create models with the same starting conditions in alternative configurations. This option defaults to -1 (time-based random number).
- **adaptive_rate**: Specify whether to enable the adaptive learning rate (ADADELTA). This option is enabled by default.
- **rho**: (Applicable only if **adaptive_rate** is enabled) Specify the adaptive learning rate time decay factor. This option defaults to 0.99.
- **epsilon**:(Applicable only if **adaptive_rate** is enabled) Specify the adaptive learning rate time smoothing factor to avoid dividing by zero. This option defaults to 1e-08.
- rate: (Applicable only if **adaptive_rate** is disabled) Specify the learning rate. Higher values result in a less stable model, while lower values lead to slower convergence. This option defaults to 0.005.
- rate_annealing: (Applicable only if **adaptive_rate** is disabled) Specify the rate annealing value. The rate annealing is calculated as rate / (1 + rate_annealing * samples). This option defaults to 1e-06.
- rate_decay: (Applicable only if **adaptive_rate** is disabled) Specify the rate decay factor between layers. The rate decay is calculated as (N-th layer: rate * rate_decay ^ (n - 1)). This options defaults to 1.
- **momentum_start**: (Applicable only if **adaptive_rate** is disabled) Specify the initial momentum at the beginning of training; we suggest 0.5. This option defaults to 0.
- **momentum_ramp**: (Applicable only if **adaptive_rate** is disabled) Specify the number of training samples for which the momentum increases. This option defaults to 1000000.
- **momentum_stable**: (Applicable only if **adaptive_rate** is disabled) Specify the final momentum after the ramp is over; we suggest 0.99. This option defaults to 0.
- **nesterov_accelerated_gradient**: (Applicable only if **adaptive_rate** is disabled) Enables the Nesterov Accelerated Gradient. This option defaults to true.

- **input_dropout_ratio**: Specify the input layer dropout ratio to improve generalization. Suggested values are 0.1 or 0.2. This option defaults to 0.
- **hidden_dropout_ratios**: (Applicable only if the activation type is **TanhWithDropout**, **RectifierWithDropout**, or **MaxoutWithDropout**) Specify the hidden layer dropout ratio to improve generalization. Specify one value per hidden layer. The range is >= 0 to <1, and the default is 0.5.
- **l1**: Specify the L1 regularization to add stability and improve generalization; sets the value of many weights to 0 (default).
- **l2**: Specify the L2 regularization to add stability and improve generalization; sets the value of many weights to smaller values. Defaults to 0.
- **max_w2**: Specify the constraint for the squared sum of the incoming weights per unit (e.g., for Rectifier). Defaults to 3.4028235e+38.
- **initial_weight_distribution**: Specify the initial weight distribution (Uniform Adaptive, Uniform, or Normal). This option defaults to Uniform Adaptive.
- **initial_weight_scale**: (Applicable only if **initial_weight_distribution** is **Uniform** or **Normal**) Specify the scale of the distribution function. For **Uniform**, the values are drawn uniformly. For **Normal**, the values are drawn from a Normal distribution with a standard deviation. This option defaults to 1.
- **initial_weights**: Specify a list of H2OFrame IDs to initialize the weight matrices of this model with.
- **initial_biases**: Specify a list of H2OFrame IDs to initialize the bias vectors of this model with.
- **loss**: Specify the loss function. The options are Automatic, CrossEntropy, Quadratic, Huber, or Absolute and the default value is Automatic. This option defaults to Automatic.
    - Use **Absolute**, **Quadratic**, or **Huber** for regression
    - Use **Absolute**, **Quadratic**, **Huber**, or **CrossEntropy** for classification

- distribution: Specify the distribution (i.e., the loss function). The options are AUTO, bernoulli, multinomial, gaussian, poisson, gamma, laplace, quantile, huber, or tweedie. This option defaults to AUTO.

    - If the distribution is `bernoulli`, the the response column must be 2-class categorical
    - If the distribution is `multinomial`, the response column must be categorical.
    - If the distribution is `poisson`, the response column must be numeric.
    - If the distribution is `laplace`, the response column must be numeric.
    - If the distribution is `tweedie`, the response column must be numeric.
    - If the distribution is `gaussian`, the response column must be numeric.
    - If the distribution is `huber`, the response column must be numeric.
    - If the distribution is `gamma`, the response column must be numeric.
    - If the distribution is `quantile`, the response column must be numeric.

- quantile_alpha: (Only applicable if `distribution="quantile"`.) Specify the quantile to be used for Quantile Regression. This option defaults to 0.5.

- tweedie_power: (Only applicable if `distribution="tweedie"`) Specify the Tweedie power. The range is from 1 to 2, and the default is 1.5.
    - For a normal distribution, enter `0`.
    - For Poisson distribution, enter `1`.
    - For a gamma distribution, enter `2`.
    - For a compound Poisson-gamma distribution, enter a value greater than 1 but less than 2.

  For more information, refer to Tweedie distribution.
- huber_alpha: Specify the desired quantile for Huber/M-regression (the threshold between quadratic and linear loss). This value must be between 0 and 1, and the default is 0.9.
- **score_interval**: Specify the shortest time interval (in seconds) to wait between model scoring. This option defaults to 5.
- **score_training_samples**: Specify the number of training set samples for scoring. The value must be >= 0. To use all training samples, enter 0. This option defaults to 10000.
- **score_validation_samples**: (Applicable only if a `validation_frame` is specified) Specify the number of validation set samples for scoring. The value must be >= 0. To use all validation samples, enter 0 (default).
- **score_duty_cycle**: Specify the maximum duty cycle fraction forscoring. A lower value results in more training and a higher value results in more scoring. This option defaults to 0.1.
- **classification_stop**: This option specifies the stopping criteria in terms of classification error (1-accuracy) on the training data scoring dataset. When the error is at or below this threshold, training stops. To disable this option, enter -1. This option defaults to 0.
- **regression_stop**: (Regression models only) Specify the stopping criterion for regression error (MSE) on the training data. When the error is at or below this threshold, training stops. To disable this option, enter -1. This option defaults to 1e-06.
- stopping_rounds: Stops training when the option selected for **stopping_metric** doesn't improve for the specified number of training rounds, based on a simple moving average. To disable this feature, specify `0`. The metric is computed on the validation data (if provided); otherwise, training data is used. This option defaults to 5.

  Note: If cross-validation is enabled:

    - All cross-validation models stop training when the validation metric doesn't improve.
    - The main model runs for the mean number of epochs.
    - N+1 models may be off by the number specified for **stopping_rounds** from the best model, but the cross-validation metric estimates the performance of the main model for the resulting number of epochs (which may be fewer than the specified number of epochs).

- stopping_metric: Specify the metric to use for early stopping. The available options are:

    - `AUTO`: This defaults to `logloss` for classification, `deviance` for regression, and `anomaly_score` for Isolation Forest. Note that custom and custom_increasing can only be used in GBM and DRF with the Python client. Must be one of: `AUTO`, `anomaly_score`. Defaults to `AUTO`.
    - `anomaly_score` (Isolation Forest only)
    - `deviance`
    - `logloss`
    - `MSE`
    - `RMSE`
    - `MAE`
    - `RMSLE`
    - `AUC` (area under the ROC curve)
    - `AUCPR` (area under the Precision-Recall curve)
    - `lift_top_group`
    - `misclassification`
    - `mean_per_class_error`
    - `custom` (Python client only)
    - `custom_increasing` (Python client only)

- stopping_tolerance: Specify the relative tolerance for the metric-based stopping to stop training if the improvement is less than this value. This option defaults to 0.
- max_runtime_secs: Maximum allowed runtime in seconds for model training. Use 0 (default) to disable.
- **score_validation_sampling**: Specify the method used to sample validation dataset for scoring. This value can be either “Uniform” (default) or “Stratified”.
- **diagnostics**: Specify whether to compute the variable importances for input features (using the Gedeon method). For large networks, enabling this option can reduce speed. This option is defaults to true (enabled).
- **fast_mode**: Specify whether to enable fast mode, a minor approximation in back-propagation. This option is defaults to true (enabled).
- **force_load_balance**: Specify whether to force extra load balancing to increase training speed for small datasets and use all cores. This option is defaults to true (enabled).
- **variable_importances**: Specify whether to compute variable importance. This option is defaults to true.
- **replicate_training_data**: Specify whether to replicate the entire training dataset onto every node for faster training on small datasets. This option is defaults to true (enabled).
- single_node_mode: Specify whether to run on a single node for fine-tuning of model parameters. This option is defaults to false (not enabled).
- **shuffle_training_data**: Specify whether to shuffle the training data. This option is recommended if the training data is replicated and the value of **train_samples_per_iteration** is close to the number of nodes times the number of rows. This option is defaults to false (not enabled).

- missing_values_handling: Specify how to handle missing values (Skip or MeanImputation). This option defaults to MeanImputation.
- **quiet_mode**: Specify whether to display less output in the standard output. This option is defaults to false (not enabled).
- **autoencoder** : Specify whether to enable the Deep Learning **autoencoder** . This option is defaults to false (not enabled).

  Note: Cross-validation is not supported when **autoencoder** is enabled.

- **sparse**: Specify whether to enable sparse data handling, which is more efficient for data with many zero values. This option is defaults to false (not enabled).
- **col_major**: Specify whether to use a column major weight matrix for the input layer. This option can speed up forward propagation but may reduce the speed of backpropagation. This option is defaults to false (not enabled).
- **average_activation**: Specify the average activation for the sparse **autoencoder** . If **Rectifier** is used, the **average_activation** value must be positive. This option defaults to 0.
- **sparsity_beta**: (Applicable only if **autoencoder** is enabled) Specify the sparsity-based regularization optimization. For more information, refer to the following link. This option defaults to 0.
- **max_categorical_features**: Specify the maximum number of categorical features enforced via hashing. The value must be at least one. This option defaults to 2147483647.
- **reproducible**: Specify whether to force reproducibility on small data. If this option is enabled, the model takes more time to generate because it uses only one thread. This option is defaults to false (not enabled).
- **export_weights_and_biases**: Specify whether to export the neural network weights and biases as H2O frames. This option is defaults to false (not enabled).
- **mini_batch_size**: Specify a value for the mini-batch size. (Smaller values lead to a better fit; larger values can speed up and generalize better.) This option defaults to 1.
- categorical_encoding: Specify one of the following encoding schemes for handling categorical features:

- `auto` or `AUTO`: Allow the algorithm to decide. In Deep Learning, the algorithm will perform `one_hot_internal` encoding if `auto` is specified. Defaults to `AUTO`.
- `one_hot_internal` or `OneHotInternal`: On the fly N+1 new cols for categorical features with N levels (default)
- `binary` or `Binary`: No more than 32 columns per categorical feature
- `eigen` or `Eigen`: *k* columns per categorical feature, keeping projections of one-hot-encoded matrix onto *k*-dim eigen space only
- `label_encoder` or `LabelEncoder`: Convert every enum into the integer of its index (for example, level 0 -> 0, level 1 -> 1, etc.). This is useful for keeping the number of columns small for XGBoost or DeepLearning, where the algorithm otherwise perform ExplicitOneHotEncoding.
- `sort_by_response` or `SortByResponse`: Reorders the levels by the mean response (for example, the level with lowest response -> 0, the level with second-lowest response -> 1, etc.). Note that this requires a specified response column.

Note: This value defaults to `one_hot_internal`. Similarly, if `auto` is specified, then the algorithm performs `one_hot_internal` encoding.

- **elastic_averaging**: Specify whether to enable elastic averaging between computing nodes, which can improve distributed model convergence. This option is defaults to false (not enabled).
- **elastic_averaging_moving_rate**: Specify the moving rate for elastic averaging. This option is only available if `elastic_averaging=True`. This option defaults to 0.9.
- **elastic_averaging_regularization**: Specify the elastic averaging regularization strength. This option is only available if `elastic_averaging=True`. This option defaults to 0.001.
- export_checkpoints_dir: Specify a directory to which generated models will automatically be exported.
- **verbose**: Print scoring history to the console. For Deep Learning, metrics are per epoch. This option is defaults to false (not enabled).

Interpreting a Deep Learning Model

To view the results, click the View button. The output for the Deep Learning model includes the following information for both the training and testing sets:

- Model parameters (hidden)
- A chart of the variable importances
- A graph of the scoring history (training MSE and validation MSE vs epochs)
- Training and validation metrics confusion matrix
- Output (model category, weights, biases)
- Status of neuron layers (layer number, units, type, dropout, L1, L2, mean rate, rate RMS, momentum, mean weight, weight RMS, mean bias, bias RMS)
- Scoring history in tabular format

- Training and validation metrics (model name, model checksum name, frame name, frame checksum name, description, model category, duration in ms, scoring time, predictions, MSE, R2, logloss)
- Top-K Hit Ratios for training and validation (for multi-class classification)

Examples

Below is a simple example showing how to build a Deep Learning model.

R | **Python** | Scala

```
import h2o
from h2o.estimators import H2ODeepLearningEstimator
h2o.init()

# Import the insurance dataset into H2O:
insurance = h2o.import_file("https://s3.amazonaws.com/h2o-public-test-
data/smalldata/glm_test/insurance.csv")

# Set the factors:
 insurance["offset"] = insurance["Holders"].log()
 insurance["Group"] = insurance["Group"].asfactor()
 insurance["Age"] = insurance["Age"].asfactor()
 insurance["District"] = insurance["District"].asfactor()

# Build and train the model:
 dl = H2ODeepLearningEstimator(distribution="tweedie",
                               hidden=[1],
                               epochs=1000,
                               train_samples_per_iteration=-1,
                               reproducible=True,
                               activation="Tanh",
                               single_node_mode=False,
                               balance_classes=False,
                               force_load_balance=False,
                               seed=23123,
                               tweedie_power=1.5,
                               score_training_samples=0,
                               score_validation_samples=0,
                               stopping_rounds=0)
 dl.train(x=list(range(3)),
          y="Claims",
          training_frame=insurance)

# Eval performance:
perf = dl.model_performance()

# Generate predictions on a test set (if necessary):
pred = dl.predict(insurance)
```

FAQ

- **How does the algorithm handle missing values during training?**

  Depending on the selected missing value handling policy, they are either imputed mean or the whole row is skipped. The default behavior is mean imputation. Note that categorical variables are imputed by adding an extra “missing” level. Optionally, Deep Learning can skip all rows with any missing values.

- **How does the algorithm handle missing values during testing?**

  Missing values in the test set will be mean-imputed during scoring.

- **What happens if the response has missing values?**

  No errors will occur, but nothing will be learned from rows containing missing the response.

- **What happens when you try to predict on a categorical level not seen during training?**

  For an unseen categorical level in the test set, Deep Learning makes an extra input neuron that remains untrained and contributes some random amount to the subsequent layer.

- **Does it matter if the data is sorted?**

  Yes, since the training set is processed in order. Depending whether `train_samples_per_iteration` is enabled, some rows will be skipped. If `shuffle_training_data` is enabled, then each thread that is processing a small subset of rows will process rows randomly, but it is not a global shuffle.

- **Should data be shuffled before training?**

  Yes, the data should be shuffled before training, especially if the dataset is sorted.

- **How does the algorithm handle highly imbalanced data in a response column?**

  Specify `balance_classes`, `class_sampling_factors` and `max_after_balance_size` to control over/under-sampling.

- **What if there are a large number of columns?**

  The input neuron layer’s size is scaled to the number of input features, so as the number of columns increases, the model complexity increases as well.

- **What if there are a large number of categorical factor levels?**

  This is something to look out for. Say you have three columns: zip code (70k levels), height, and income. The resulting number of internally one-hot encoded features will be 70,002 and only 3 of them will be activated (non-zero). If the first hidden layer has 200 neurons, then the resulting weight matrix will be of size 70,002 x 200, which can take a long time to train and converge. In this case, we recommend either reducing the number of categorical factor levels upfront (e.g., using `h2o.interaction()` from R), or specifying `max_categorical_features` to use feature hashing to reduce the dimensionality.

- **How does your Deep Learning Autoencoder work? Is it deep or shallow?**

  H2O's DL **autoencoder** is based on the standard deep (multi-layer) neural net architecture, where the entire network is learned together, instead of being stacked layer-by-layer. The only difference is that no response is required in the input and that the output layer has as many neurons as the input layer. If you don't achieve convergence, then try using the *Tanh* activation and fewer layers. We have some example test scripts here, and even some that show how stacked auto-encoders can be implemented in R.

- **When building the model, does Deep Learning use all features or a selection of the best features?**

  For Deep Learning, all features are used, unless you manually specify that columns should be ignored. Adding an L1 penalty can make the model sparse, but it is still the full size.

- **What is the relationship between iterations, epochs, and the ``train_samples_per_iteration`` parameter?**

  Epochs measures the amount of training. An iteration is one MapReduce (MR) step - essentially, one pass over the data. The `train_samples_per_iteration` parameter is the amount of data to use for training for each MR step, which can be more or less than the number of rows.

- **When do ``reduce()`` calls occur, after each iteration or each epoch?**

  Neither; `reduce()` calls occur after every two `map()` calls, between threads and ultimately between nodes. There are many `reduce()` calls, much more than one per MapReduce step (also known as an "iteration"). Epochs are not related to MR iterations, unless you specify `train_samples_per_iteration` as `0` or `-1` (or to number of rows/nodes). Otherwise, one MR iteration can train with an arbitrary number of training samples (as specified by `train_samples_per_iteration`).

- **Does each Mapper task work on a separate neural-net model that is combined during reduction, or is each Mapper manipulating a shared object that's persistent across nodes?**

  Neither; there's one model per compute node, so multiple Mappers/threads share one model, which is why H2O is not reproducible unless a small dataset is used and `force_load_balance=F` or `reproducible=T`, which effectively rebalances to a single chunk and leads to only one thread to launch a `map()`. The current behavior is simple model averaging; between-node model averaging via "Elastic Averaging" is currently in progress.

- **Is the loss function and backpropagation performed after each individual training sample, each iteration, or at the epoch level?**

  Loss function and backpropagation are performed after each training sample (mini-batch size 1 == online stochastic gradient descent).

- **When using Hinton's dropout and specifying an input dropout ratio of ~20% and ``train_samples_per_iteration`` is set to 50, will each of the 50 samples have a different set of the 20% input neurons suppressed?**

  Yes - suppression is not done at the iteration level across as samples in that iteration. The dropout mask is different for each training sample.

- **When using dropout parameters such as ``input_dropout_ratio``, what happens if you use only ``Rectifier`` instead of ``RectifierWithDropout`` in the activation parameter?**

  The amount of dropout on the input layer can be specified for all activation functions, but hidden layer dropout is only supported is set to `WithDropout`. The default hidden dropout is 50%, so you don't need to specify anything but the activation type to get good results, but you can set the hidden dropout values for each layer separately.

- **When using the ``score_validation_sampling`` and ``score_training_samples`` parameters, is scoring done at the end of the Deep Learning run?**

The majority of scoring takes place after each MR iteration. After the iteration is complete, it may or may not be scored, depending on two criteria: the time since the last scoring and the time needed for scoring.

The maximum time between scoring (`score_interval` , default = 5 seconds) and the maximum fraction of time spent scoring (`score_duty_cycle`) independently of loss function, backpropagation, etc.

Of course, using more training or validation samples will increase the time for scoring, as well as scoring more frequently. For more information about how this affects runtime, refer to the Deep Learning Performance Guide.

- **How does the validation frame affect the built neuron network?**

  The validation frame is only used for scoring and does not directly affect the model. However, the validation frame can be used stopping the model early if `overwrite_with_best_model = T` , which is the default. If this parameter is enabled, the model with the lowest validation error is displayed at the end of the training.

  By default, the validation frame is used to tune the model parameters (such as number of epochs) and will return the best model as measured by the validation metrics, depending on how often the validation metrics are computed (`score_duty_cycle`) and whether the validation frame itself was sampled.

  Model-internal sampling of the validation frame (`score_validation_samples` and `score_validation_sampling` for optional stratification) will affect early stopping quality. If you specify a validation frame but set `score_validation_samples` to more than the number of rows in the validation frame (instead of 0, which represents the entire frame), the validation metrics received at the end of training will not be reproducible, since the model does internal sampling.

- **Are there any best practices for building a model using checkpointing?**

In general, to get the best possible model, we recommend building a model with `train_samples_per_iteration = -2` (which is the default value for auto-tuning) and saving it.

To improve the initial model, start from the previous model and add iterations by building another model, setting the checkpoint to the previous model, and changing `train_samples_per_iteration`, `target_ratio_comm_to_comp`, or other parameters.

If you don't know your model ID because it was generated by R, look it up using `h2o.ls()`. By default, Deep Learning model names start with `deeplearning_` To view the model, use `m <- h2o.getModel("my_model_id")` or `summary(m)`.

There are a few ways to manage checkpoint restarts:

Option 1: (Multi-node only) Leave `train_samples_per_iteration = -2`, increase `target_comm_to_comp` from 0.05 to 0.25 or 0.5, which provides more communication. This should result in a better model when using multiple nodes. **Note:** This does not affect single-node performance.

Option 2: (Single or multi-node) Set `train_samples_per_iteration` to (N), where (N) is the number of training samples used for training by the entire cluster for one iteration. Each of the nodes then trains on (N) randomly-chosen rows for every iteration. The number defined as (N) depends on the dataset size and the model complexity.

Option 3: (Single or multi-node) Change regularization parameters such as `l1, l2, max_w2, input_droput_ratio` or `hidden_dropout_ratios`. We recommend build the first mode using `RectifierWithDropout`, `input_dropout_ratio = 0` (if there is suspected noise in the input), and `hidden_dropout_ratios=c(0,0,0)` (for the ability to enable dropout regularization later).

- **How does class balancing work?**

The `max_after_balance_size` parameter defines the maximum size of the over-sampled dataset. For example, if `max_after_balance_size = 3`, the over-sampled dataset will not be greater than three times the size of the original dataset.

For example, if you have five classes with priors of 90%, 2.5%, 2.5%, and 2.5% (out of a total of one million rows) and you oversample to obtain a class balance using `balance_classes = T`, the result is all four minor classes are oversampled by forty times and the total dataset will be 4.5 times as large as the original dataset (900,000 rows of each class). If `max_after_balance_size = 3`, all five balance classes are reduced by 3/5 resulting in 600,000 rows each (three million total).

To specify the per-class over- or under-sampling factors, use `class_sampling_factors`. In the previous example, the default behavior with `balance_classes` is equivalent to `c(1,40,40,40,40)`, while when `max_after_balance\size = 3`, the results would be `c(3/5,40*3/5,40*3/5,40*3/5)`.

In all cases, the probabilities are adjusted to the pre-sampled space, so the minority classes will have lower average final probabilities than the majority class, even if they were sampled to reach class balance.

- **How is variable importance calculated for Deep Learning?**

  For Deep Learning, variable importance is calculated using the Gedeon method.

- **How is deviance computed for a Deep Learning regression model?**

  The following formula is used to compute deviance for a Deep Learning regression model:

  > Loss = Quadratic -> MSE==Deviance For Absolute/Laplace or Huber -> MSE != Deviance

---

Deep Learning Tuning Guide

The Definitive Performance Tuning Guide for H2O Deep Learning

- R
- Blog

References

"Deep Learning." *Wikipedia: The free encyclopedia*. Wikimedia Foundation, Inc. 1 May 2015. Web. 4 May 2015.

"Artificial Neural Network." *Wikipedia: The free encyclopedia*. Wikimedia Foundation, Inc. 22 April 2015. Web. 4 May 2015.

Zeiler, Matthew D. 'ADADELTA: An Adaptive Learning Rate Method'. Arxiv.org. N.p., 2012. Web. 4 May 2015.

Sutskever, Ilya et al. "On the importance of initialization and momementum in deep learning." JMLR:W&CP vol. 28. (2013).

Hinton, G.E. et. al. "Improving neural networks by preventing co-adaptation of feature detectors." University of Toronto. (2012).

Wager, Stefan et. al. "Dropout Training as Adaptive Regularization." Advances in Neural Information Processing Systems. (2013).

Gedeon, TD. "Data mining of inputs: analysing magnitude and functional measures." University of New South Wales. (1997).

Candel, Arno and Parmar, Viraj. "Deep Learning with H2O." H2O.ai, Inc. (2015).

Deep Learning Training

Slideshare slide decks

Youtube channel

Candel, Arno. "The Definitive Performance Tuning Guide for H2O Deep Learning." H2O.ai, Inc. (2015).

Niu, Feng, et al. "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent." Advances in Neural Information Processing Systems 24 (2011): 693-701. (algorithm implemented is on p.5)

Hawkins, Simon et al. "Outlier Detection Using Replicator Neural Networks." CSIRO Mathematical and Information Sciences