Deep Learning with H2O

Arno Candel Jessica Lanford Erin LeDell Viraj Parmar

http://h2o.gitbooks.io/deep-learning/

August 2015: Third Edition

Deep Learning with H2O
by Arno Candel, Jessica Lanford, Erin LeDell, & Viraj Parmar

Published by H2O.ai, Inc.
2307 Leghorn St.
Mountain View, CA 94043

August 2015: Third Edition

Contents

1 Introduction

This document introduces the reader to Deep Learning with H2O. Examples are written in R and Python. The reader is walked through the installation of H2O, basic deep learning concepts, building deep neural nets in H2O, how to interpret model output, how to make predictions, and various implementation details.

2 What is H2O?

H2O is fast, scalable, open-source machine learning and deep learning for Smarter Applications. With H2O, enterprises like PayPal, Nielsen, Cisco, and others can use all their data without sampling to get accurate predictions faster. Advanced algorithms, like Deep Learning, Boosting, and Bagging Ensembles are built-in to help application designers create smarter applications through elegant APIs. Some of our initial customers have built powerful domain-specific predictive engines for Recommendations, Customer Churn, Propensity to Buy, Dynamic Pricing, and Fraud Detection for the Insurance, Healthcare, Telecommunications, AdTech, Retail, and Payment Systems industries.

Using in-memory compression, H2O handles billions of data rows in-memory, even with a small cluster. To make it easier for non-engineers to create complete analytic workflows, H2O's platform includes interfaces for R, Python, Scala, Java, JSON, and Coffeescript/JavaScript, as well as a built-in web interface, Flow. H2O was built alongside (and on top of) Hadoop and Spark Clusters and typically deploys within minutes.

H2O includes many common machine learning algorithms, such as generalized linear modeling (linear regression, logistic regression, etc.), Naïve Bayes, principal components analysis, time series, k-means clustering, and others. H2O also implements best-in-class algorithms at scale, such as Random Forest, Gradient Boosting and Deep Learning. Customers can build thousands of models and compare the results to get the best predictions.

H2O is nurturing a grassroots movement of physicists, mathematicians, and computer scientists to herald the new wave of discovery with data science by collaborating closely with academic researchers and Industrial data scientists. Stanford university giants Stephen Boyd, Trevor Hastie, Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. With hundreds of meetups over the past two years, H2O has become a word-of-mouth phenomenon, growing amongst the data community by a hundred-fold, and is now used by 12,000+ users and is deployed using R, Python, Hadoop, and Spark in 2000+ corporations.

Try it out

H2O's R package can be installed from CRAN at `https://cran.r-project.org/web/packages/h2o/`. A Python package can be installed from PyPI at `https://pypi.python.org/pypi/h2o/`. Download H2O directly from `http://h2o.ai/download`.

Join the community

Visit the open source community forum at `https://groups.google.com/d/forum/h2ostream`. To learn about our meetups, training sessions, hackathons, and product updates, visit `http://h2o.ai`.

3 Installation

The easiest way to directly install H2O is via an R or Python package.

(**Note**: This document was created with H2O version 3.0.1.4.)

3.1 Installation in R

To load a recent H2O package from CRAN, run:

```
install.packages("h2o")
```

Note: The version of H2O in CRAN is often one release behind the current version.

Alternatively, you can (and should for this tutorial) download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in R" tab.
4. Copy and paste the commands into your R session.

After H2O is installed on your system, verify the installation:

```
library(h2o)

#Start H2O on your local machine using all available cores.
#By default, CRAN policies limit use to only 2 cores.
h2o.init(nthreads = -1)

#Get help
?h2o.glm
?h2o.gbm

#Show a demo
demo(h2o.glm)
demo(h2o.gbm)
```

3.2 Installation in Python

To load a recent H2O package from PyPI, run:

```
pip install h2o
```

Alternatively, you can (and should for this tutorial) download the latest stable H2O-3 build from the H2O download page:

1. Go to `http://h2o.ai/download`.
2. Choose the latest stable H2O-3 build.
3. Click the "Install in Python" tab.
4. Copy and paste the commands into your Python session.

After H2O is installed, verify the installation:

```
import h2o

# Start H2O on your local machine
h2o.init()

# Get help
help(h2o.glm)
help(h2o.gbm)

# Show a demo
h2o.demo("glm")
h2o.demo("gbm")
```

3.3 Pointing to a different H2O cluster

Following the instructions in the previous sections create a one-node H2O cluster on your local machine.

To connect to an established H2O cluster (in a multi-node Hadoop environment, for example) specify the IP address and port number for the established cluster using the `ip` and `port` parameters in the `h2o.init()` command:

```
h2o.init(ip="123.45.67.89", port=54321)
```

3.4 Example code

R code for the examples in this document are available here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/booklets/v2_2015/source/DeepLearning_Vignette.R`

Python code for the examples in this document can be found here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/booklets/v2_2015/source/DeepLearning_Vignette.ipynb`

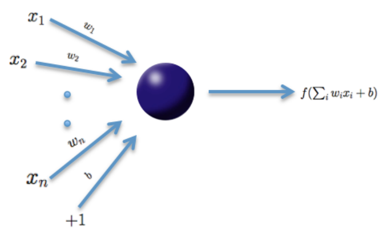
The document source itself can be found here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/booklets/v2_2015/source/DeepLearning_Vignette.tex`

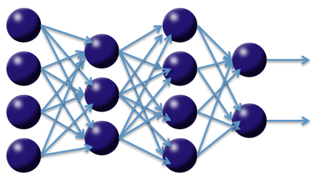
4 Deep Learning Overview

Unlike the neural networks of the past, modern Deep Learning has cracked the code for training stability and generalization and scales on big data. It is often the algorithm of choice for highest predictive accuracy, as deep learning algorithms performs quite well in a number of diverse problems.

First, we present a brief overview of deep neural networks for supervised learning tasks. There are several theoretical frameworks for deep learning, and here we summarize the feedforward architecture used by H2O.



The basic unit in the model (shown above) is the neuron, a biologically inspired model of the human neuron. For humans, varying strengths of neurons' output signals travel along the synaptic junctions and are then aggregated as input for a connected neuron's activation. In the model, the weighted combination $\alpha = \sum_{i=1}^{n} w_i x_i + b$ of input signals is aggregated, and then an output signal $f(\alpha)$ transmitted by the connected neuron. The function f represents the nonlinear activation function used throughout the network, and the bias b accounts for the neuron's activation threshold.



Multi-layer, feedforward neural networks consist of many layers of interconnected neuron units, starting with an input layer to match the feature space, followed by multiple layers of nonlinearity, and ending with a linear regression or classification layer to match the output space. The inputs and outputs of the model's units follow the basic logic of the single neuron described above. Bias units are included in each non-output layer of the network. The weights linking neurons and biases with other neurons fully determine the output of the entire network, and learning occurs when these weights are adapted to minimize the error on labeled training data. More specifically, for each training example j, the objective is to minimize a loss function,

$$L(W, B \mid j).$$

Here, W is the collection $\{W_i\}_{1:N-1}$, where W_i denotes the weight matrix connecting layers i and $i+1$ for a network of N layers. Similarly B is the collection $\{b_i\}_{1:N-1}$, where b_i denotes the column vector of biases for layer $i+1$. This basic framework of multi-layer neural networks can be used to accomplish deep learning tasks. Deep learning architectures are models of hierarchical feature extraction, typically involving multiple levels of nonlinearity. Deep learning models are able to learn useful representations of raw data and have exhibited high performance on complex data such as images, speech, and text (Bengio, 2009).

5 H2O's Deep Learning architecture

As described above, H2O follows the model of multi-layer, feedforward neural networks for predictive modeling. This section provides a more detailed description of H2O's Deep Learning features, parameter configurations, and computational implementation.

5.1 Summary of features

H2O's Deep Learning functionalities include:

- purely supervised training protocol for regression and classification tasks
- fast and memory-efficient Java implementations based on columnar compression and fine-grain Map/Reduce
- multi-threaded and distributed parallel computation to be run on either a single node or a multi-node cluster
- fully automatic per-neuron adaptive learning rate for fast convergence
- optional specification of learning rate, annealing and momentum options
- regularization options include L1, L2, dropout, Hogwild! and model averaging to prevent model overfitting
- elegant web interface or fully scriptable R API from H2O CRAN package
- grid search for hyperparameter optimization and model selection
- model checkpointing for reduced run times and model tuning
- automatic data pre and post-processing for categorical and numerical data
- automatic imputation of missing values
- automatic tuning of communication vs computation for best performance
- model export in plain java code for deployment in production environments
- additional expert parameters for model tuning
- deep autoencoders for unsupervised feature learning and anomaly detection capabilities

5.2 Training protocol

The training protocol described below follows many of the ideas and advances in the recent deep learning literature.

5.2.1 Initialization

Various deep learning architectures employ a combination of unsupervised pretraining followed by supervised training, but H2O uses a purely supervised training protocol. The default initialization scheme is the uniform adaptive option, which is an optimized initialization based on the size of the network. Alternatively, you may select a random initialization to be drawn from either a uniform or normal distribution, for which a scaling parameter may be specified as well.

5.2.2 Activation and loss functions

In the introduction, we described the nonlinear activation function f; the choices are summarized in Table 1. Note here that x_i and w_i denote the firing neuron's input values and their weights, respectively; α denotes the weighted combination $\alpha = \sum_i w_i x_i + b$. The $\tanh$ function is a rescaled and shifted logistic function and its symmetry around 0 allows the training algorithm to converge faster. The rectified linear activation function has demonstrated high performance on image recognition tasks, and is a more biologically accurate model of neuron activations (LeCun et al, 1998). Maxout activation works particularly well with dropout, a regularization method discussed later in this vignette (Goodfellow et al, 2013). It is difficult to determine a "best" activation function to use; each may outperform the others in separate scenarios, but grid search

Table 1: Activation functions

Function	Formula	Range
Tanh	$f(\alpha) = \frac{e^{\alpha}-e^{-\alpha}}{e^{\alpha}+e^{-\alpha}}$	$f(\cdot) \in [-1, 1]$
Rectified Linear	$f(\alpha) = \max(0, \alpha)$	$f(\cdot) \in \mathbb{R}_{+}$
Maxout	$f(\cdot) = \max(w_i x_i + b)$, rescale if $\max f(\cdot) \geq 1$	$f(\cdot) \in [-\infty, 1]$

models (also described later) can help to compare activation functions and other parameters. The default activation function is the Rectifier. Each of these activation functions can be operated with dropout regularization (see below).

The following choices for the loss function $L(W, B \mid j)$ are summarized in Table 2. The system default enforces the table's typical use rule based on whether regression or classification is being performed. Note here that $t^{(j)}$ and $o^{(j)}$ are the predicted (target) output and actual output, respectively, for training example j; further, let y denote the output units and O the output layer.

Table 2: Loss functions

Function	Formula	Typical use
Mean Squared Error	$L(W,B\|j) = \frac{1}{2}\|\|t^{(j)} - o^{(j)}\|\|_2^2$	Regression
Absolute	$L(W,B\|j) =$ TO DO	Regression
Huber	$L(W,B\|j) =$ TO DO	Regression
Cross Entropy	$L(W,B\|j) = -\sum_{y \in O} \left(\ln(o_y^{(j)}) \cdot t_y^{(j)} + \ln(1 - o_y^{(j)}) \cdot (1 - t_y^{(j)}) \right)$	Classification

5.2.3 Parallel distributed network training

The procedure to minimize the loss function $L(W, B \mid j)$ is a parallelized version of stochastic gradient descent (SGD). Standard SGD can be summarized as follows, with the gradient $\nabla L(W, B \mid j)$ computed via backpropagation (LeCun et al, 1998). The constant α indicates the learning rate, which controls the step sizes during gradient descent.

Standard stochastic gradient descent

```
1. Initialize $W, B$
2. Iterate until convergence criterion reached:
        a. Get training example $i$
        b. Update all weights $w_{jk} \in W$, biases $b_{jk} \in B$
            $w_{jk} := w_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial w_{jk}}$
            $b_{jk} := b_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial b_{jk}}$
```

Stochastic gradient descent is known to be fast and memory-efficient, but not easily parallelizable without becoming slow. We utilize Hogwild!, the recently developed lock-free parallelization scheme from Niu et al, 2011, to address this issue. Hogwild! follows a shared memory model where multiple cores (each handling separate subsets or all of the training data) are able to make independent contributions to the gradient updates $\nabla L(W, B \mid j)$ asynchronously. In a multi-node system, this parallelization scheme works on top of H2O's distributed setup where the training data is distributed across the cluster. Each node operates in parallel on its local data until the final parameters W, B are obtained by averaging. Below is a rough summary.

Parallel distributed and multi-threaded training with SGD in H2O Deep Learning

```
1. Initialize global model parameters $W, B$
2. Distribute training data $\mathcal{T}$ across nodes (can be disjoint or replicated)
3. Iterate until convergence criterion reached:
    3.1. For nodes $n$ with training subset $\mathcal{T}_n$, do in parallel:
        a. Obtain copy of the global model parameters $W_n, B_n$
        b. Select active subset $\mathcal{T}_{na} \subset \mathcal{T}_n$ (user-given number of samples per iteration)
        c. Partition $\mathcal{T}_{na}$ into $\mathcal{T}_{nac}$ by cores $n_c$
        d. For cores $n_c$ on node $n$, do in parallel:
            i. Get training example $i \in \mathcal{T}_{nac}$
            ii. Update all weights $w_{jk} \in W_n$, biases $b_{jk} \in B_n$
                $w_{jk} := w_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial w_{jk}}$
                $b_{jk} := b_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial b_{jk}}$
    3.2. Set $W, B :=$ Avg$_n$ $W_n$, Avg$_n$ $B_n$
    3.3. Optionally score the model on (potentially sampled) train/validation scoring sets
```

Here, the weights and bias updates follow the asynchronous HOGWILD! procedure to incrementally adjust each node's parameters W_n, B_n after seeing example i. The Avg$_n$ notation refers to the final averaging of these local parameters across all nodes to obtain the global model parameters and complete training.

5.2.4 Specifying the number of training samples per iteration

H2O Deep Learning is scalable and can take advantage of large clusters of compute nodes. There are three operating modes. The default behavior is to let every node train on the entire (replicated) dataset, but automatically shuffling (and/or using a subset of) the training examples for each iteration locally. For datasets that don't fit into each node's memory (depending on the amount of heap memory specified by the `-XmX` Java option), it might not be possible to replicate the data, and each compute node can be instructed to train only with local data. An experimental single node mode is available for cases where final convergence is slow due to the presence of too many nodes, but this has not been necessary in our testing.

The number of training examples globally presented to the distributed SGD worker nodes between model averaging is defined by the parameter `train_samples_per_iteration`. If the specified value is -1, all nodes process all their local training data per iteration. If `replicate_training_data` is enabled, which is the default setting, this will result in training N epochs (passes over the data) per iteration on N nodes; otherwise, one epoch will be trained per iteration. Another special value is 0, which always results in one epoch per iteration, regardless of the number of compute nodes. In general, any user-specified positive number is permissible for this parameter. For large datasets, we recommend specifying a fraction of the dataset.

For example, if the training data contains 10 million rows, and we specify the number of training samples per iteration as $100,000$ when running on four nodes, then each node will process $25,000$ examples per iteration, and it will take 40 distributed iterations to process one epoch. If the value is too high, it might take too long between synchronization and model convergence may be slow. If the value is too low, network communication overhead will dominate the runtime and computational performance will suffer. A value of -2, which is the default value, enables auto-tuning for this parameter, based on the computational performance of the processors and the network of the system, and attempts to find a good balance between computation and communication. This parameter can affect the convergence rate during training.

5.3 Regularization

H2O's Deep Learning framework supports regularization techniques to prevent overfitting.

ℓ_1 (Lasso) and ℓ_2 (Ridge) regularization enforce the same penalties as they do with other models; that is, modifying the loss function so as to minimize some loss,

$$L'(W,B \mid j) = L(W,B \mid j) + \lambda_1 R_1(W,B \mid j) + \lambda_2 R_2(W,B \mid j).$$

For ℓ_1 regularization, $R_1(W,B \mid j)$ represents of the sum of all ℓ_1 norms of the weights and biases in the network; $R_2(W,B \mid j)$ represents the sum of squares of all the weights and biases in the network. The constants λ_1 and λ_2 are generally specified as very small, for example 10^{-5}.

The second type of regularization available for deep learning is a modern innovation called dropout (Hinton et al., 2012). Dropout constrains the online optimization so that during forward propagation for a given training example, each neuron in the network suppresses its activation with probability P, which is usually less than 0.2 for input neurons and up to 0.5 for hidden neurons. There are two effects: as with ℓ_2 regularization, the network weight values are scaled toward 0. Furthermore, each training example trains a different model, although they share the same global parameters. As a result, dropout allows an exponentially large number of models to be averaged as an ensemble, which can prevent overfitting and improve generalization. Input dropout can be especially useful when the feature space is large and noisy.

5.4 Advanced optimization

H2O features manual and automatic versions of advanced optimization. The manual mode features include momentum training and learning rate annealing, while automatic mode features an adaptive learning rate.

5.4.1 Momentum training

Momentum modifies back-propagation by allowing prior iterations to influence the current version. In particular, a velocity vector, v, is defined to modify the updates as follows: with θ representing the parameters W, B; μ representing the momentum coefficient, and α denoting the learning rate.

$$v_{t+1} = \mu v_t - \alpha \nabla L(\theta_t)$$
$$\theta_{t+1} = \theta_t + v_{t+1}$$

Using the momentum parameter can aid in avoiding local minima and the associated instability (Sutskever et al, 2014). Too much momentum can lead to instabilities, which is why it is best to ramp up the momentum slowly. The parameters that control momentum are `momentum_start`, `momentum_ramp` and `momentum_stable`.

The Nesterov accelerated gradient method, triggered by the `nesterov_accelerated_gradient` parameter, is a recommended improvement when using momentum updates. Using this method, the updates are further modified such that

$$v_{t+1} = \mu v_t - \alpha \nabla L(\theta_t + \mu v_t)$$
$$W_{t+1} = W_t + v_{t+1}$$

5.4.2 Rate annealing

Throughout training, as the model approaches a minimum, the chance of oscillation or "optimum skipping" creates the need for a slower learning rate. Instead of specifying a constant learning rate α, learning rate annealing gradually reduces the learning rate α_t to "freeze" into local minima in the optimization landscape (Zeiler, 2012).

For H2O, the annealing rate (`rate_annealing`) is the inverse of the number of training samples it takes to cut the learning rate in half (e.g., 10^{-6} means that it takes 10^6 training samples to halve the learning rate).

5.4.3 Adaptive learning

The implemented adaptive learning rate algorithm ADADELTA (Zeiler, 2012) automatically combines the benefits of learning rate annealing and momentum training to avoid slow convergence. Specifying only two parameters (ρ and ϵ) simplifies hyper parameter search. In some cases, manually controlled (non-adaptive) learning rate and momentum specifications can lead to better results, but require a hyperparameter search of up to 7 parameters. If the model is built on a topology with many local minima or long plateaus, it is possible for a constant learning rate to produce sub-optimal results. In general, however, we find the adaptive learning rate produces the best results, so this option is used as the default.

The first of two hyper parameters for adaptive learning is ρ (`rho`). It is similar to momentum and relates to the memory to prior weight updates. Typical values are between 0.9 and 0.999. The second of two hyper parameters, ϵ (`epsilon`), for adaptive learning is similar to learning rate annealing during initial training and momentum at later stages where it allows forward progress. Typical values are between 10^{-10} and 10^{-4}.

5.5 Loading data

Loading a dataset in R or Python for use with H2O is slightly different from the usual methodology, as we must convert our datasets into `H2OFrame` objects (distributed data frames), rather than using an R `data.frame` or `data.table` or a Python `pandas.DataFrame` or `numpy.array`.

5.5.1 Standardization

Along with categorical encoding, H2O's Deep Learning preprocesses the data to be standardized for compatibility with the activation functions (recall Table 1's summary of each activation function's target space). Since the activation function does not generally map into the full spectrum of real numbers, $\mathbb{R}$, we first standardize our data to be drawn from $\mathcal{N}(0, 1)$. Standardizing again after network propagation allows us to compute more precise errors in this standardized space, rather than in the raw feature space.

5.6 Additional parameters

This section provided some background on the various parameter configurations in H2O's Deep Learning architecture. Since there are dozens of possible parameter arguments when creating models, H2O Deep Learning models may seem daunting. However, most parameters do not need to be modified; the default settings are recommended as safe. The majority of the parameters that support (and in some cases, require) experimentation are discussed in the previous sections but there are a few more that will be discussed in the following sections.

There is no default setting for the hidden layer size, number, or epochs. Experimenting with building deep learning models using different network topologies and different datasets will lead to intuition for these parameters but two general rules of thumb should be applied. First, choose larger network sizes, as they can perform higher-level feature extraction, and techniques like dropout may train only subsets of the network at once. Second, use more epochs for greater predictive accuracy, but only when the computational cost is affordable. Many example tests can be found in the H2O GitHub repository for pointers on specific values and results for these (and other) parameters.

For a full list of H2O Deep Learning model parameters and default values, see Appendix A.

6 Use case: MNIST digit classification

6.1 MNIST overview

The MNIST database is a well-known academic dataset used to benchmark classification performance. The data consists of 60,000 training images and 10,000 test images, for which each is a standardized 28^2 pixel greyscale image of a single handwritten digit. An example of the scanned handwritten digits is shown in Figure 1.

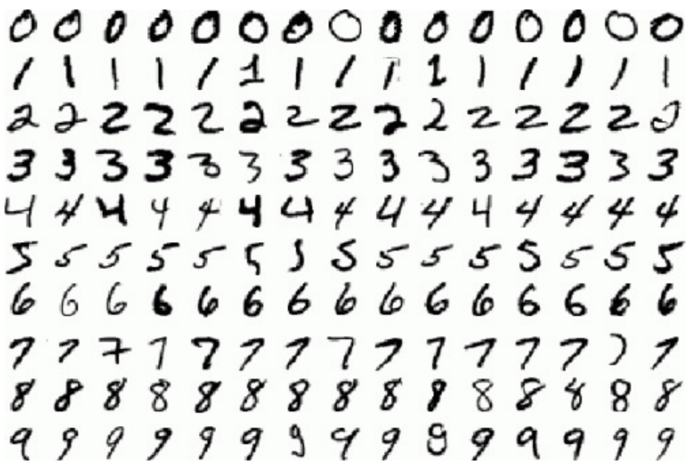


Figure 1: Example MNIST digit images

For this example, we will download and import the train and test datasets from a public Amazon S3 bucket.The train file is 13MB and the test file is 2.1MB, and below we download the data directly so the speed that the data is imported is limited by download speed. Files can be uploaded from a variety of sources, including a remote locations and HDFS.

Example in R

```
library(h2o)
h2o.init()

train_file <- "https://h2o-public-test-data.s3.amazonaws.com/bigdata/
    laptop/mnist/train.csv.gz"
test_file <- "https://h2o-public-test-data.s3.amazonaws.com/bigdata/
    laptop/mnist/test.csv.gz"

train <- h2o.importFile(train_file, header = FALSE, sep = ",")
test <- h2o.importFile(test_file, header = FALSE, sep = ",")

# To see a brief summary of the data, run the following command
summary(train)
summary(test)
```

Example in Python

```
import h2o
h2o.init()

train_file = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/
    laptop/mnist/train.csv.gz"
test_file = "https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop
    /mnist/test.csv.gz"

train = h2o.import_frame(train_file)
test = h2o.import_frame(test_file)

# To see a brief summary of the data, run the following command
train.describe()
test.describe()
```

6.2 Performing a trial run

The example below illustrates the relative simplicity underlying most H2O Deep Learning model parameter configurations, as a result of the default settings. We use the first $28^2 = 784$ values of each row to represent the full image and the final value to denote the digit class. Rectified linear activation is popular with image processing and has performed well on the MNIST database previously and dropout has been known to enhance performance on this dataset as well, so we train our model accordingly.

Example in R

```
# We first encode the response column as categorical for multinomial
    classification
train[,785] <- as.factor(train[,785])

# Train a Deep Learning model and validate on a test set
model <- h2o.deeplearning(x = 1:784, y = 785,
                          training_frame = train,
                          distribution = "multinomial",
                          activation = "RectifierWithDropout",
                          hidden = c(200,200,200),
                          input_dropout_ratio = 0.2,
                          l1 = 1e-5,
                          validation_frame = test,
                          epochs = 10)
```

Example in Python

```
# TO DO: Update this example -- not tested
# We first encode the response column as categorical for multinomial
    classification
#train[,785] = as.factor(train[,785])

# Train a Deep Learning model and validate on a test set
#model = h2o.deeplearning(x = 1:784, y = 785,
#                           training_frame = train,
#                           distribution = "multinomial",
#                           activation = "RectifierWithDropout",
#                           hidden = c(200,200,200),
#                           input_dropout_ratio = 0.2,
#                           l1 = 1e-5,
#                           validation_frame = test,
#                           epochs = 10)
```

The model runs for only 10 epochs since it is just meant as a trial run. In this trial run, we also specified the validation set as the test set. In addition to, or instead of using a validation set, another option is to use k-fold validation by specifying, for example, `nfolds = 5` in addition to / instead of `validation_frame = test`. When `nfolds` is a positive integer, k-fold cross validation will be performed and the cross-validation metrics will be computed. Optionally, the user can save the cross-validation predicted values (generated during cross-validation) by setting `keep_cross_validation_predictions` parameter to true.

6.2.1 Extracting and handling the results

We can extract the parameters of our model, examine the scoring process, and make predictions on new data.

Example in R

```
# View the specified parameters of your deep learning model
model@parameters

# Examine the performance of the trained model
model  # display all performance metrics

h2o.performance(model, valid = FALSE)  # training set metrics
h2o.performance(model, valid = TRUE)   # validation set metrics
```

Example in Python

```
# TO DO: Add a Python version of this code

# # View the specified parameters of your deep learning model
# model@parameters
#
# # Examine the performance of the trained model
# model  # display all performance metrics
#
# h2o.performance(model, valid = FALSE)  # training set metrics
# h2o.performance(model, valid = TRUE)   # validation set metrics
```

The second command returns the training and validation errors for the model. The training error value is based on the parameter `score_training_samples`, which specifies the number of randomly sampled training points to be used for scoring (the default uses 10,000 points). The validation error is based on the parameter `score_validation_samples`, which configures the same value on the validation set; by default, this is the entire validation set.

In general, choosing a greater number of sampled points leads to a better understanding of the model's performance on your dataset; setting either of these parameters to 0 automatically uses the entire corresponding dataset for scoring. However, either method allows you to control the minimum and maximum time spent on scoring with the `score_interval` and `score_duty_cycle` parameters.

These scoring parameters also affect the final model when the parameter `overwrite_with_best_model` is enabled. This option selects the model that achieved the lowest validation error during training (based on the sampled points used for scoring) as the final model after training. If a dataset is not specified as the validation set, the training data is used by default; in this case, either the `score_training_samples` or `score_validation_samples` parameter will control the error computation during training and, in turn, the selected best model.

Once we have a satisfactory model, the `h2o.predict()` command can be used to compute and store predictions on new data, which can then be used for further tasks in the interactive data science process.

Example in R

```
# Perform classification on the test set
# This also returns the probability for each class
prediction <- h2o.predict(model, newdata = test)

# If desired, you can copy predictions from H2O to R
pred  <- as.data.frame(prediction)
```

Example in Python

```
# Perform classification on the held out data
prediction = model.predict(test)

# Copy predictions from H2O to Python
pred = prediction.as_data_frame()

pred.head()
```

6.3 Web interface

H2O R users have access to an intuitive web interface for H2O, Flow, to mirror the model building process in R. After loading data or training a model in R, point your browser to your IP address and port number (e.g., localhost:12345) to launch the web interface. From here, you can click on Admin > Jobs to view specific details about your model. You can also click on Data > List All Frames to view all current H2O frames.

6.3.1 Variable importances

The variable importances feature can be enabled with the argument `variable_importances = TRUE`. This feature allows us to view the absolute and relative predictive strength of each feature in the prediction task. Each H2O algorithm class has it's own methodology for computing variable importance. For H2O's Deep Learning, the Gedeon method is used, which can be slow for large networks, so it is turned off by default. If variable importance is the top priority in your analysis, you may (also) consider training a Random Forest and inspecting the variable importances generated with that method.

The following code demonstrates training using the `variable_importances` option enabled and how to extract the variable importances from the trained model. From the web UI, you can also view a visualization of the variable importances.

Example in R

```r
# Train a Deep Learning model and validate on a test set
# and save the variable importances
model_varimp <- h2o.deeplearning(x = 1:784, y = 785,
                          training_frame = train,
                          distribution = "multinomial",
                          activation = "RectifierWithDropout",
                          hidden = c(200,200,200),
                          input_dropout_ratio = 0.2,
                          l1 = 1e-5,
                          validation_frame = test,
                          epochs = 10,
                          variable_importances = TRUE)  # add

h2o.varimp(model_varimp)
```

Example in Python

```python
# TO DO: Convert to Python

# Train a Deep Learning model and validate on a test set
# and save the variable importances
# model_varimp <- h2o.deeplearning(x = 1:784, y = 785,
#                                  training_frame = train,
#                                  distribution = "multinomial",
#                                  activation = "RectifierWithDropout",
#                                  hidden = c(200,200,200),
#                                  input_dropout_ratio = 0.2,
#                                  l1 = 1e-5,
#                                  validation_frame = test,
#                                  epochs = 10,
#                                  variable_importances = TRUE)  # add
#
# h2o.varimp(model_varimp)
```

6.3.2 Java model

Another important feature of the web interface is the Java (POJO) model, accessible from the PREVIEW POJO button at the bottom of the model results. This button allows access to Java code that builds the model when called from a main method in a Java program. Instructions for downloading and running this Java code are available from the web interface, and example production scoring code is available as well.

6.4 Grid search for model comparison

H2O supports grid search capabilities for model tuning by allowing users to tweak certain parameters and observe changes in model behavior. This is done by specifying sets of values for parameter arguments. The following example represents a sample grid search:

Example in R

```r
activation_opt <- c("Tanh", "Maxout")
hidden_opt <- list(c(100,100), c(100,200,100))
hyper_params <- list(activation = activation_opt,
                     hidden = hidden_opt)

grid <- h2o.grid("deeplearning",
                 hyper_params = hyper_params,
                 x = 1:784,
                 y = 785,
                 distribution = "bernoulli",
                 training_frame = train,
                 validation_frame = test)
```

Example in Python

```python
# TO DO: Add Python example

# activation_opt <- c("Tanh", "Maxout")
# hidden_opt <- list(c(100,100), c(100,200,100))
# hyper_params <- list(activation = activation_opt, hidden = hidden_opt)
#
# grid <- h2o.grid("deeplearning",
#                  hyper_params = hyper_params,
#                  x = 1:784,
#                  y = 785,
#                  distribution = "bernoulli",
#                  training_frame = train,
#                  validation_frame = test)
```

In this example, we specified three different network topologies and two different ℓ_1 norm weights. This grid search model trains six different models using all possible combinations of these parameters; other parameter combinations can be specified for a larger space of models. This provides more subtle insights into the model tuning and selection process by inspecting and comparing our trained models after the grid search process is complete. To learn how and when to select different parameter configurations in a grid search, refer to Appendix A for parameter descriptions and configurable values.

Example in R

```
# TO DO: Check that this works

# print out all prediction errors and run times of the models
grid

# print out the auc for all of the models
grid_models <- lapply(grid@model_ids, function(model_id) { model = h2o.
    getModel(model_id) })
for (i in 1:length(grid_models)) {
  print(sprintf("auc:␣%f", h2o.auc(grid_models[[i]])))
}
```

Example in Python

```
# TO DO: Add Python version

# # print out all prediction errors and run times of the models
# grid
#
# # print out the auc for all of the models
# grid_models <- lapply(grid@model_ids, function(model_id) { model = h2o.
    getModel(model_id) })
# for (i in 1:length(grid_models)) {
#   print(sprintf("auc: %f", h2o.auc(grid_models[[i]])))
# }
```

6.5 Checkpoint model

To resume model training, use checkpoint model keys for incrementally training a particular model with more iterations, more data, different data, and so forth. To train our initial model further, we can use it (or its key) as a checkpoint argument for a new model.

In the command below, `mnist_model_grid@model[[1]]` represents the highest performing model from the grid search used for additional training. For checkpoint restarts, the training and validation datasets, as well as the response column, must match.

```
mnist_checkpoint_model = h2o.deeplearning(x=1:784, y=785, data=train_
    images.hex, checkpoint=mnist_model_grid@model[[1]], validation = test
    _images.hex, epochs=9)
```

Checkpointing can also be used to reload existing models that were saved to disk in a previous session. For example, we can save and reload the best model from the grid search by running the following commands.

```
#Specify a model and the file path where it is to be saved
h2o.saveModel(object = mnist_model_grid@model[[1]], name = "/tmp/mymodel"
    , force = TRUE)

#Alternatively, save the model key in some directory (here we use /tmp)
#h2o.saveModel(object = mnist_model_grid@model[[1]], dir = "/tmp", force
    = TRUE)
```

After restarting H2O, load the saved model by specifying the host and saved model file path. **Note**: The saved model must be reloaded using a compatible H2O version (i.e., the same version used to save the model).

```
best_mnist_grid.load = h2o.loadModel(h2o_server, "/tmp/mymodel")

#Continue training the loaded model
best_mnist_grid.continue = h2o.deeplearning(x=1:784, y=785, data=train_
    images.hex, checkpoint=best_mnist_grid.load, validation = test_images
    .hex, epochs=1)
```

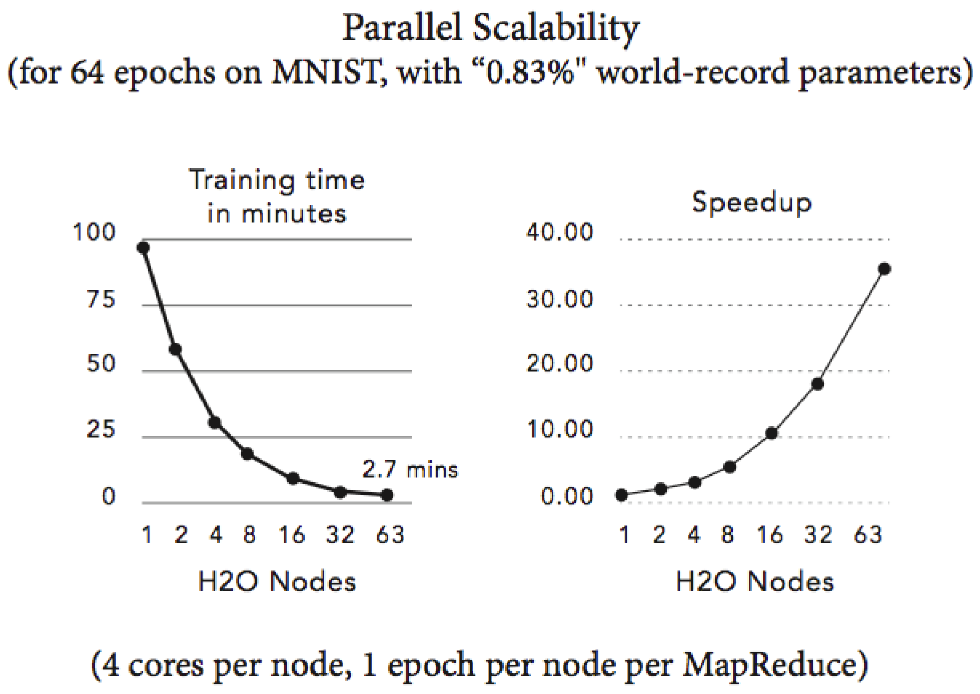
Additionally, you can also use the command

```
model = h2o.getModel(h2o_server, key)
```

to retrieve a model from its H2O key. This command is useful, for example, if you have created an H2O model using the web interface and wish to proceed with the modeling process in R.

6.6 Achieving world-record performance

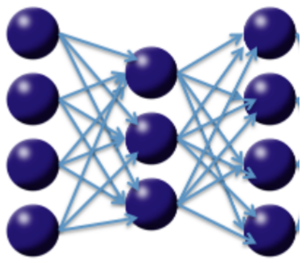
Without distortions, convolutions, or other advanced image processing techniques, the best-ever published test set error for the MNIST dataset is 0.83% by Microsoft. After training for $2,000$ epochs (took about 4 hours) on 4 compute nodes, we obtain 0.87% test set error and after training for $8,000$ epochs (took about 10 hours) on 10 nodes, we obtain 0.83% test set error, which is the current world-record, notably achieved using a distributed configuration and with a simple 1-liner from R. Details can be found in our hands-on tutorial. Accuracies around 1% test set errors are typically achieved within 1 hour when running on 1 node. The parallel scalability of H2O for the MNIST dataset on 1 to 63 compute nodes is shown in the figure below.



7 Deep Autoencoders

7.1 Nonlinear dimensionality reduction

So far, we have discussed purely supervised deep learning tasks. However, deep learning can also be used for unsupervised feature learning or, more specifically, nonlinear dimensionality reduction (Hinton et al, 2006). Consider the diagram of a three-layer neural network with one hidden layer on the following page. If we treat our input data as labeled with the same input values, then the network is forced to learn the identity via a nonlinear, reduced representation of the original data. This type of algorithm is called a deep autoencoder; these models have been used extensively for unsupervised, layer-wise pre-training of supervised deep learning tasks, but here we discuss the autoencoder's ability to discover anomalies in data.



7.2 Use case: anomaly detection

Consider the deep autoencoder model described above. Given enough training data that resembles some underlying pattern, the network will train itself to easily learn the identity when confronted with that pattern. However, if some "anomalous" test point that does not match the learned pattern arrives, the autoencoder will likely have a high error in reconstructing this data, which indicates it is anomalous data.

We use this framework to develop an anomaly detection demonstration using a deep autoencoder. The dataset is an ECG time series of heartbeats and the goal is to determine which heartbeats are outliers. The training data (20 "good" heartbeats) and the test data (training data with 3 "bad" heartbeats appended for simplicity) can be downloaded from the H2O GitHub repository for the H2O Deep Learning documentation at http://bit.ly/1yywZziEach row represents a single heartbeat. The autoencoder is trained as follows:

```
train_ecg.hex = h2o.uploadFile(h2o_server, path="ecg_train.csv", header=F
    , sep=",", key="train_ecg.hex")
test_ecg.hex = h2o.uploadFile(h2o_server, path="ecg_test.csv", header=F,
    sep=",", key="test_ecg.hex")

#Train deep autoencoder learning model on "normal" training data, y
    ignored
anomaly_model = h2o.deeplearning(x=1:210, y=1, train_ecg.hex, activation
    = "Tanh", classification=F, autoencoder=T, hidden = c(50,20,50), l1=1
    E-4,
epochs=100)

#Compute reconstruction error with the Anomaly detection app (MSE between
     output layer and input layer)
recon_error.hex = h2o.anomaly(test_ecg.hex, anomaly_model)

```

```
#Pull reconstruction error data into R and plot to find outliers (last 3
   heartbeats)
recon_error = as.data.frame(recon_error.hex)
recon_error
plot.ts(recon_error)

#Note: Testing = Reconstructing the test dataset
test_recon.hex = h2o.predict(anomaly_model, test_ecg.hex)
head(test_recon.hex)
```

8 Appendix A: Complete parameter list

- x: A vector containing the names of the predictors in the model. No default.
- y: The name of the response variable in the model. No default.
- training_frame: An H2OFrame object containing the variables in the model. No default.
- model_id: (Optional) The unique ID assigned to the resulting model. If not specified, an ID is generated automatically.
- overwrite_with_best_model: Logical. If enabled, overwrite the final model with the best model found during training. Default is true.
- validation_frame: (Optional) An H2OFrame object that represents the validation dataset used to construct the confusion matrix. If blank and nfolds = 0, the training data is used by default.
- checkpoint: (Optional) Model checkpoint (either key or H2ODeepLearningModel) used to resume training.
- autoencoder: Logical. Enable autoencoder; default is false. Refer to the "Deep Autoencoders" section for more details.
- use_all_factor_levels: Logical. Use all factor levels of categorical variance. Otherwise, the first factor level is omitted (without loss of accuracy). Useful for variable importances and auto-enabled for autoencoder. Default is true.
- activation: The choice of nonlinear, differentiable activation function used throughout the network. Options are Tanh, TanhWithDropout, Rectifier, RectifierWithDropout, Maxout, MaxoutWithDropout, and the default is Rectifier. Refer to the "Activation and loss functions" section for more details.
- hidden: The number and size of each hidden layer in the model. For example, if c(100,200,100) is specified, a model with 3 hidden layers will be produced, the middle hidden layer will have 200 neurons and the first and third hidden layers will have 100 neurons. The default is c(200,200). For grid search, use list(c(10,10), c(20,20)) etc. Refer to the section on "Performing a trial run" for more details.
- epochs: The number of passes or over the training dataset (can be fractional). It is recommended to start with lower values for initial grid searches. The value can be modified during checkpoint restarts and allows continuation of selected models. Default is 10.
- train_samples_per_iteration: The number of training samples (globally) per MapReduce iteration. The following special values are also supported: 0 (one epoch); -1 (all available data; e.g., replicated training data); -2 (auto-tuning; default). Refer to the "Specifying the number of training samples per iteration" for more details.
- seed: The random seed controls sampling and initialization. Reproducible results are only expected with single-threaded operation (i.e. when running on one node, turning off load balancing and providing a small dataset that fits in one chunk). In general, the multi-threaded asynchronous updates to the model parameters will result in (intentional) race conditions and non-reproducible results. Default is a randomly generated number.
- adaptive_rate: Logical. Enables adaptive learning rate (ADADELTA). Default is true. Refer to the "Adaptive learning" section for more details.
- rho: Adaptive learning rate time decay factor. This parameter is similar to momentum and relates to the memory to prior weight updates. Typical values are between 0.9 and 0.999. Default value is 0.99. Refer to the section on "Adaptive learning" for more details.
- epsilon: The second of two hyperparameters for adaptive learning rate (when it is enabled). This parameter is similar to learning rate annealing during initial training and momentum at later stages where it allows forward progress. Typical values are between 1e-10 and 1e-4. This parameter is only

active if `adaptive_rate` is enabled. Default is 1e-8. Refer to the "Adaptive learning" section for more details.

- `rate`: The learning rate, α. Higher values lead to less stable models, while lower values lead to slower convergence. Default is 0.005.
- `rate_annealing`: The annealing learning rate is calculated as `(rate)`/(1+`rate_annealing` * N), where N is the number of training samples. It is the inverse of the number of training samples required to cut the learning rate in half. This reduces the learning rate to "freeze" into local minima in the optimization landscape. Default value is 1e-6 (when adaptive learning is disabled). Refer to the "Rate annealing" section for more details.
- `rate_decay`: Learning rate decay factor between layers (L-th layer: `rate` * `rate_decay`^ (L-1)); default is 1.0 (when adaptive learning is disabled). The learning rate decay parameter controls the change of learning rate across layers.
- `momentum_start`: This controls the amount of momentum at the beginning of training (when adaptive learning is disabled). Default is 0. Refer to the "Momentum training" section for more details.
- `momentum_ramp`: This controls the amount of learning for which momentum increases (assuming `momentum_stable` is larger than `momentum_start`). The ramp is measured in the number of training samples and can be enabled when adaptive learning is disabled. Default is 1 million (1e6). Refer to the "Momentum training" section or more details.
- `momentum_stable`: This controls the final momentum value reached after `momentum_ramp` number of training samples (when adaptive learning is disabled). The momentum used for training will remain the same for training beyond reaching that point. Default is 0. Refer to the "Momentum training" for more details.
- `nesterov_accelerated_gradient`: Logical. The Nesterov accelerated gradient descent method is a modification to traditional gradient descent for convex functions. The method relies on gradient information at various points to build a polynomial approximation that minimizes the residuals in fewer iterations of the descent. This parameter is only active if adaptive learning rate is disabled. The default is true (when adaptive learning is disabled). Refer to the "Momentum training" section for more details.
- `input_dropout_ratio`: The fraction of the features for each training row to be omitted from training in order to improve generalization. The default is 0 (always use all features). Refer to the "Regularization" section for more details.
- `hidden_dropout_ratios`: The fraction of the inputs for each hidden layer to be omitted from training in order to improve generalization. The default is 0.5 for each hidden layer. Refer to the "Regularization" section for more details.
- `l1`: L1 regularization, which constrains the absolute value of the weights (can add stability and improve generalization, causes many weights to become 0). The default is 0, for no L1 regularization. Refer to the "Regularization" section for more details.
- `l2`: L2 regularization, which constrains the sum of the squared weights (can add stability and improve generalization, causes many weights to be small). The default is 0, for no L2 regularization. Refer to the "Regularization" section for more details.
- `max_w2`: A maximum on the sum of the squared incoming weights into any one neuron. This tuning parameter is especially useful for unbound activation functions such as Maxout or Rectifier. The default, positive infinity, leaves this maximum unbounded.
- `initial_weight_distribution`: The distribution from which initial weights are to be drawn. Select `Uniform`, `UniformAdaptive` or `Normal`. Default is `UniformAdaptive`. Refer to the "Initialization" for more details.
- `initial_weight_scale`: The scale of the distribution function for Uniform or Normal distributions. For Uniform, the values are drawn uniformly from (-`initial_weight_scale`, `initial_weight_scale`).

For Normal, the values are drawn from a Normal distribution with a standard deviation of `initial_weight_scale`. The default is 1. Refer to the "Initialization" for more details.

- `loss`: Specify one of the loss options: `Automatic`, `CrossEntropy` (for classification only), `MeanSquare`, `Absolute`, or `Huber`. Refer to the "Activation and loss functions" section for more details.
- `distribution`: Specify the distribution function of the response: `AUTO`, `bernouilli`, `multinomial`, `poisson`, `gamma`, `tweedie`, `laplace`, `huber`, or `gaussian`.
- `tweedie_power`: Specify the tweedie power; applicable only if `distribution` is set to `tweedie`. Value must be between 1.0 and 2.0.
- `score_interval`: The minimum time (in seconds) to elapse between model scoring. The actual interval is determined by the number of training samples per iteration and the scoring duty cycle. To use all training set samples, specify `0`. Default is 5.
- `score_training_samples`: The number of training samples to be used for scoring. These samples will be randomly sampled. Use 0 to select the entire training dataset. Default is 10000.
- `score_validation_samples`: The number of validation dataset points to be used for scoring. Can be randomly sampled or stratified (if `balance_classes` is set and `score_validation_sampling` is set to stratify). Use 0 to select the entire validation dataset (default).
- `score_duty_cycle`: Maximum duty cycle fraction spent on model scoring (on both training and validation samples), and on diagnostics such as computation of feature importances (i.e., not on training). Lower values result in more training, while higher values produce more scoring. Default is 0.1.
- `classification_stop`: The stopping criterion for classification error (1 - accuracy) on the training data scoring dataset. When the error is at or below this threshold, the training process stops. Default is 0. To disable, enter `-1`.
- `regression_stop`: The stopping criterion for regression error (MSE) on the training data scoring dataset. When the error is at or below this threshold, the training process stops. Default is 1e-6. To disable, enter `-1`.
- `quiet_mode`: Logical. Enable quiet mode for less output to standard output. Default is false.
- `max_confusion_matrix_size`: For classification models, the maximum size (in terms of classes) of the confusion matrix to display. This option is meant to avoid printing extremely large confusion matrices. Default is 20.
- `max_hit_ratio_k`: The maximum number (top K) of predictions to use for hit ratio computation (for multi-class only, enter 0 to disable). Default is 10.
- `balance_classes`: Logical. For imbalanced data, the training data class counts can be artificially balanced by over-sampling the minority class(es) and under-sampling the majority class(es) so that each class will contain the same number of observations. This can result in improved predictive accuracy. Over-sampling is done with replacement (rather than simulating new observations), and the total number of observations after balancing is controlled by the `max_after_balance_size` parameter. Default is false.
- `class_sampling_factors`: Desired over/under-sampling ratios per class (in lexicographic order). Only applies to classification when `balance_classes` is enabled. If not specified, the ratios will be automatically computed to obtain class balance during training.
- `max_after_balance_size`: When classes are balanced, limit the resulting dataset size to the specified multiple of the original dataset size. This is the maximum relative size of the training data after balancing class counts (can be less than 1.0). Default is 5.
- `score_validation_sampling`: Method used to sample validation dataset for scoring. The possible methods are `Uniform` and `Stratified`. Default is `Uniform`.

- `diagnostics`: (Deprecated) Logical. Gather diagnostics for hidden layers, such as mean and root mean squared (RMS) values of learning rate, momentum, weights and biases. Since deprecation, diagnostics are always enabled (set to true).
- `variable_importances`: Logical. Compute variable importances for input features using the Gedeon method. The implementation considers the weights connecting the input features to the first two hidden layers. Default is false, since this can be slow for large networks.
- `fast_mode`: Logical. Enable fast mode (minor approximation in back-propagation). This should not affect results significantly. Default is true.
- `ignore_const_cols`: Logical. Ignore constant training columns (no information can be gained anyway). Default is true.
- `force_load_balance`: Logical. Force extra load balancing to increase training speed for small datasets to keep all cores busy. Default is true.
- `replicate_training_data`: Logical. Replicate the entire training dataset onto every node for faster training on small datasets. Default is true.
- `single_node_mode`: Logical. Run on a single node for fine-tuning of model parameters. Can be useful for faster convergence during checkpoint resumes after training on a very large count of nodes (for fast initial convergence). Default is false.
- `shuffle_training_data`: Logical. Enable shuffling of training data (on each node). This option is recommended if training data is replicated on N nodes, and the number of training samples per iteration is close to N times the dataset size, where all nodes train with (almost) all of the data. It is automatically enabled if the number of training samples per iteration is set to -1 (or to N times the dataset size or larger). Default is false.
- `sparse`: (Deprecated) Logical. Enable sparse data handling.Default is false.
- `col_major`: (Deprecated) Logical. Use a column major weight matrix for the input layer; can speed up forward propagation, but may slow down backpropagation. Default is false.
- `average_activation`: Specify the average activation for the sparse autoencoder (Experimental). Default is 0.
- `sparsity_beta`: Specify the sparsity-based regularization optimization (Experimental). Default is 0.
- `max_categorical_features`: Maximum number of categorical features allowed in a column, enforced via hashing (Experimental). Default is $2^{31} - 1$ (`Integer.MAX_VALUE` in Java).
- `reproducible`: Logical. Force reproducibility on small data (will be slow – only uses one thread). Default is false.
- `export_weights_and_biases`: Logical. Specify whether to export the neural network weights and biases as an `H2OFrame`. Default is false.
- `offset_column`: Specify the offset column by column name. Regression only. 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 value directly, 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.
- `weights_column`: Specify the weights column by column name. 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.
- `nfolds`: (Optional) Number of folds for cross-validation. Default is 0 (no cross-validation is performed).

- `fold_column`: (Optional) Name of column with cross-validation fold index assignment per observation; the folds are supplied by the user.
- `fold_assignment`: Cross-validation fold assignment scheme, if `nfolds` is greater than zero and `fold_column` is not specified. Options are `AUTO`, `Random`, or `Modulo`.
- `keep_cross_validation_predictions`: Logical. Specify whether to keep the predictions of the cross-validation models. Default is false.

9 Appendix B: References

H2O website: `http://h2o.ai/`

H2O documentation: `http://docs.h2o.ai`

H2O Github Repository: `http://github.com/h2oai/h2o.git`

H2O Training: `http://learn.h2o.ai/`

H2O Training Scripts and Data: `http://data.h2o.ai/`

Code for this Document:
`https://github.com/h2oai/h2o/tree/master/docs/deeplearning/DeepLearningRVignetteDemo`

H2O support: `h2ostream@googlegroups.com`

H2O JIRA: `https://0xdata.atlassian.net/secure/Dashboard.jspa`

YouTube Channel: `https://www.youtube.com/user/0xdata`

Learning Deep Architectures for AI. Bengio, Yoshua, 2009.
`http://www.iro.umontreal.ca/~lisa/pointeurs/TR1312.pdf`

Efficient BackProp. LeCun et al, 1998. `http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf`

Maxout Networks. Goodfellow et al, 2013. `http://arxiv.org/pdf/1302.4389.pdf`

HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent.
Niu et al, 2011. `http://i.stanford.edu/hazy/papers/hogwild-nips.pdf`

Improving neural networks by preventing co-adaptation of feature detectors.
Hinton et al., 2012. http://arxiv.org/pdf/1207.0580.pdf

On the importance of initialization and momentum in deep learning. Sutskever et al, 2014.
http://www.cs.toronto.edu/ fritz/absps/momentum.pdf

ADADELTA: AN ADAPTIVE LEARNING RATE METHOD. Zeiler, 2012.
http://arxiv.org/pdf/1212.5701v1.pdf

H2O GitHub repository for the H2O Deep Learning documentation
https://github.com/h2oai/h2o/tree/master/docs/deeplearning/DeepLearningRVignetteDemo

MNIST database http://yann.lecun.com/exdb/mnist/

Reducing the Dimensionality of Data with Neural Networks. Hinton et al, 2006.
http://www.cs.toronto.edu/ hinton/science.pdf

Definitive Performance Tuning Guide for Deep Learning. http://h2o.ai/blog/2015/02/deep-learning-performance/