Deep Learning with H2O

Arno Candel Erin LeDell

Edited by: Angela Bartz

---

http://h2o.ai/resources/

April 2019: Sixth Edition

Deep Learning with H2O
by Arno Candel & Erin LeDell
with assistance from Viraj Parmar & Anisha Arora
Edited by: Angela Bartz

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

April 2019: Sixth Edition

Photos by ©H2O.ai, Inc.

Printed in the United States of America.

Contents

1 Introduction **5**

2 What is H2O? **5**

3 Installation **6**
- 3.1 Installation in R . . . 7
- 3.2 Installation in Python . . . 7
- 3.3 Pointing to a Different H2O Cluster . . . 8
- 3.4 Example Code . . . 8
- 3.5 Citation . . . 9

4 Deep Learning Overview **9**

5 H2O's Deep Learning Architecture **10**
- 5.1 Summary of Features . . . 11
- 5.2 Training Protocol . . . 12
  - 5.2.1 Initialization . . . 12
  - 5.2.2 Activation and Loss Functions . . . 12
  - 5.2.3 Parallel Distributed Network Training . . . 15
  - 5.2.4 Specifying the Number of Training Samples . . . 17
- 5.3 Regularization . . . 18
- 5.4 Advanced Optimization . . . 18
  - 5.4.1 Momentum Training . . . 19
  - 5.4.2 Rate Annealing . . . 19
  - 5.4.3 Adaptive Learning . . . 20
- 5.5 Loading Data . . . 20
  - 5.5.1 Data Standardization/Normalization . . . 20
  - 5.5.2 Convergence-based Early Stopping . . . 21
  - 5.5.3 Time-based Early Stopping . . . 21
- 5.6 Additional Parameters . . . 21

6 Use Case: MNIST Digit Classification **22**
- 6.1 MNIST Overview . . . 22
- 6.2 Performing a Trial Run . . . 25
  - 6.2.1 N-fold Cross-Validation . . . 27
  - 6.2.2 Extracting and Handling the Results . . . 28
- 6.3 Web Interface . . . 31
  - 6.3.1 Variable Importances . . . 31
  - 6.3.2 Java Model . . . 33
- 6.4 Grid Search for Model Comparison . . . 33

6.4.1 Cartesian Grid Search 34
6.4.2 Random Grid Search 35
6.5 Checkpoint Models 37
6.6 Achieving World-Record Performance 41
6.7 Computational Performance 41

7 Deep Autoencoders **42**
7.1 Nonlinear Dimensionality Reduction 42
7.2 Use Case: Anomaly Detection 43
7.2.1 Stacked Autoencoder 46
7.2.2 Unsupervised Pretraining with Supervised Fine-Tuning 46

8 Parameters **46**

9 Common R Commands **53**

10 Common Python Commands **53**

11 Acknowledgments **53**

12 References **54**

13 Authors **55**

Introduction

This document introduces the reader to Deep Learning with H2O. Examples are written in R and Python. Topics include:

- installation of H2O
- basic Deep Learning concepts
- building deep neural nets in H2O
- how to interpret model output
- how to make predictions

as well as various implementation details.

What is H2O?

H2O.ai is focused on bringing AI to businesses through software. Its flagship product is H2O, the leading open source platform that makes it easy for financial services, insurance companies, and healthcare companies to deploy AI and deep learning to solve complex problems. More than 9,000 organizations and 80,000+ data scientists depend on H2O for critical applications like predictive maintenance and operational intelligence. The company – which was recently named to the CB Insights AI 100 – is used by 169 Fortune 500 enterprises, including 8 of the world's 10 largest banks, 7 of the 10 largest insurance companies, and 4 of the top 10 healthcare companies. Notable customers include Capital One, Progressive Insurance, Transamerica, Comcast, Nielsen Catalina Solutions, Macy's, Walgreens, and Kaiser Permanente.

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 is designed to run in standalone mode, on Hadoop, or within a Spark Cluster, 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, k-means clustering, and word2vec. H2O implements best-in-class algorithms at scale, such as distributed random forest, gradient boosting, and deep learning. H2O also includes a Stacked Ensembles method, which finds the optimal combination of a collection of prediction algorithms using a process

known as "stacking." With H2O, 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, and Rob Tibshirani advise the H2O team on building scalable machine learning algorithms. And with hundreds of meetups over the past several years, H2O continues to remain a word-of-mouth phenomenon.

Try it out

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

Join the community

- To learn about our training sessions, hackathons, and product updates, visit `http://h2o.ai`.
- To learn about our meetups, visit `https://www.meetup.com/topics/h2o/all/`.
- Have questions? Post them on Stack Overflow using the **h2o** tag at `http://stackoverflow.com/questions/tagged/h2o`.
- Have a Google account (such as Gmail or Google+)? Join the open source community forum at `https://groups.google.com/d/forum/h2ostream`.
- Join the chat at `https://gitter.im/h2oai/h2o-3`.

Installation

H2O requires Java; if you do not already have Java installed, install it from `https://java.com/en/download/` before installing H2O.

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

Installation in R

To load a recent H2O package from CRAN, run:

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

Note: The version of H2O in CRAN may be one release behind the current version.

For the latest recommended version, 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
?h2o.deeplearning

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

Installation in Python

To load a recent H2O package from PyPI, run:

```
pip install h2o
```

To 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.estimators.glm.H2OGeneralizedLinearEstimator)
help(h2o.estimators.gbm.H2OGradientBoostingEstimator)
help(h2o.estimators.deeplearning.
    H2ODeepLearningEstimator)

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

Pointing to a Different H2O Cluster

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. The syntax for this function is identical for R and Python:

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

Example Code

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

`https://github.com/h2oai/h2o-3/tree/master/h2o-docs/src/booklets/v2_2015/source/DeepLearning_Vignette_code_examples`

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`

Citation

To cite this booklet, use the following:

Candel, A., Parmar, V., LeDell, E., and Arora, A. (Apr 2019). *Deep Learning with H2O*. `http://h2o.ai/resources`.

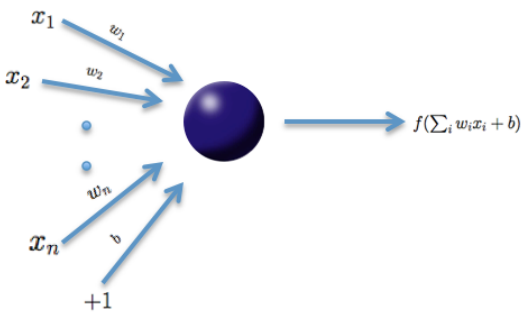
Deep Learning Overview

Unlike the neural networks of the past, modern Deep Learning provides training stability, generalization, and scalability with big data. Since it performs quite well in a number of diverse problems, Deep Learning is quickly becoming the algorithm of choice for the highest predictive accuracy.

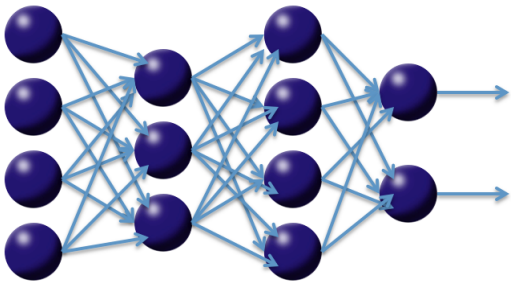
The first section is a brief overview of deep neural networks for supervised learning tasks. There are several theoretical frameworks for Deep Learning, but this document focuses primarily on the feedforward architecture used by H2O.

The basic unit in the model (shown in the image below) is the neuron, a biologically inspired model of the human neuron. In humans, the varying strengths of the 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 represents the neuron's activation threshold.



Multi-layer, feedforward neural networks consist of many layers of interconnected neuron units (as shown in the following image), 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. Learning occurs when these weights are adapted to minimize the error on the 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).

H2O's Deep Learning Architecture

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.

Summary of Features

H2O's Deep Learning functionalities include:

- supervised training protocol for regression and classification tasks
- fast and memory-efficient Java implementations based on columnar compression and fine-grain MapReduce
- multi-threaded and distributed parallel computation that can be run on a single or a multi-node cluster
- automatic, per-neuron, adaptive learning rate for fast convergence
- optional specification of learning rate, annealing, and momentum options
- regularization options such as L1, L2, dropout, HOGWILD!, and model averaging to prevent model overfitting
- elegant and intuitive web interface (Flow)
- fully scriptable R API from H2O's CRAN package
- fully scriptable Python API
- grid search for hyperparameter optimization and model selection
- automatic early stopping based on convergence of user-specified metrics to user-specified tolerance
- model checkpointing for reduced run times and model tuning
- automatic pre- and post-processing for categorical and numerical data
- automatic imputation of missing values (optional)
- 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

Training Protocol

The training protocol described below follows many of the ideas and advances discussed in recent Deep Learning literature.

Initialization

Various Deep Learning architectures employ a combination of unsupervised pre-training 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. Deep Learning can also be started using a random initialization drawn from either a uniform or normal distribution, optionally specifying a scaling parameter.

Activation and Loss Functions

The choices for the nonlinear activation function f described in the introduction are summarized in Table 1 below. x_i and w_i represent the firing neuron's input values and their weights, respectively; α denotes the weighted combination $\alpha = \sum_i w_i x_i + b$.

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	$f(\alpha_1, \alpha_2) = \max(\alpha_1, \alpha_2)$	$f(\cdot) \in \mathbb{R}$

The $\tanh$ function is a rescaled and shifted logistic function; 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 is a generalization of the `Rectifiied Linear` activation, where each neuron picks the largest output of k separate channels, where each channel has its own weights and bias values. The current implementation supports only $k = 2$. Maxout activation works particularly well with dropout (Goodfellow et al, 2013). For more information, refer to **Regularization**.

The Rectifier is the special case of Maxout where the output of one channel is always 0. It is difficult to determine a "best" activation function to use; each may outperform the others in separate scenarios, but grid search models can help to compare activation functions and other parameters. For more information, refer to **Grid Search for Model Comparison**. The default activation function is the Rectifier. Each of these activation functions can be operated with dropout regularization. For more information, refer to **Regularization**.

Specify the one of the following distribution functions for the response variable using the `distribution` argument:

- AUTO
- Bernoulli
- Multinomial
- Poisson
- Gamma
- Tweedie
- Laplace
- Quantile
- Huber
- Gaussian

Each distribution has a primary association with a particular loss function, but some distributions allow users to specify a non-default loss function from the group of loss functions specified in Table 2. Bernoulli and multinomial are primarily associated with cross-entropy (also known as log-loss), Gaussian with Mean Squared Error, Laplace with Absolute loss (a special case of Quantile with `quantile_alpha=0.5`) and Huber with Huber loss. For Poisson, Gamma, and Tweedie distributions, the loss function cannot be changed, so `loss` must be set to AUTO.

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 (also known as target) output and actual output, respectively, for training example j; further, let y represent 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	$L(W,B\|j) = \|t^{(j)} - o^{(j)}\|_1$	Regression
Huber	$L(W,B\|j) = \begin{cases} \frac{1}{2}\|t^{(j)} - o^{(j)}\|_2^2 & \text{for } \|t^{(j)} - o^{(j)}\|_1 \leq 1, \\ \|t^{(j)} - o^{(j)}\|_1 - \frac{1}{2} & \text{otherwise.} \end{cases}$	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

To predict the 80-th percentile of the petal length of the Iris dataset in R, use the following:

Example in R

```
library(h2o)
h2o.init(nthreads = -1)
train.hex <- h2o.importFile("https://h2o-public-test-
    data.s3.amazonaws.com/smalldata/iris/iris_wheader.
    csv")
splits <- h2o.splitFrame(train.hex, 0.75, seed=1234)
dl <- h2o.deeplearning(x=1:3, y="petal_len",
        training_frame=splits[[1]],
        distribution="quantile", quantile_alpha=0.8)
h2o.predict(dl, splits[[2]])
```

To predict the 80-th percentile of the petal length of the Iris dataset in Python, use the following:

Example in Python

```
import h2o
from h2o.estimators.deeplearning import
    H2ODeepLearningEstimator
h2o.init()
train = h2o.import_file("https://h2o-public-test-data.
    s3.amazonaws.com/smalldata/iris/iris_wheader.csv")
splits = train.split_frame(ratios=[0.75], seed=1234)
dl = H2ODeepLearningEstimator(distribution="quantile",
     quantile_alpha=0.8)
dl.train(x=range(0,2), y="petal_len", training_frame=
    splits[0])
print(dl.predict(splits[1]))
```

Parallel Distributed Network Training

The process of minimizing the loss function $L(W, B \mid j)$ is a parallelized version of stochastic gradient descent (SGD). A summary of standard SGD provided below, with the gradient $\nabla L(W, B \mid j)$ computed via backpropagation (LeCun et al, 1998). The constant α is 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 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 (where each core handles 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 that distributes the training data across the cluster. Each node operates in parallel on its local data until the final parameters W, B are obtained by averaging.

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 the example i. The Avg$_n$ notation represents the final averaging of these local parameters across all nodes to obtain the global model parameters and complete training.

Specifying the Number of Training Samples

H2O Deep Learning is scalable and can take advantage of large clusters of compute nodes. There are three operating modes. The default behavior allows every node to 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, so each compute node can be specified 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.

To specify the global number of training examples shared with the distributed SGD worker nodes between model averaging, use the `train_samples_per_iteration` parameter. If the specified value is -1, all nodes process all their local training data on each 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. Specifying 0 always results in one epoch per iteration regardless of the number of compute nodes. In general, this parameter supports any positive number. For large datasets, we recommend specifying a fraction of the dataset.

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.

For example, if the training data contains 10 million rows, and the number of training samples per iteration is specified 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.

Regularization

H2O's Deep Learning framework supports regularization techniques to prevent overfitting. ℓ_1(L1: Lasso) and ℓ_2 (L2: Ridge) regularization enforce the same penalties as they do with other models: modifying the loss function so as to minimize 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, $R_1(W, B \mid j)$ is the sum of all ℓ_1 norms for the weights and biases in the network; ℓ_2 regularization via $R_2(W, B \mid j)$ 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. Although they share the same global parameters, each training example trains a different model. As a result, dropout allows an exponentially large number of models to be averaged as an ensemble to help prevent overfitting and improve generalization.

If the feature space is large and noisy, specifying an input dropout using the `input_dropout_ratio` parameter can be especially useful. Note that input dropout can be specified independently of the dropout specification in the hidden layers (which requires `activation` to be `TanhWithDropout`, `MaxoutWithDropout`, or `RectifierWithDropout`). Specify the amount of hidden dropout per hidden layer using the `hidden_dropout_ratios` parameter, which is set to 0.5 by default.

Advanced Optimization

H2O features manual and automatic advanced optimization modes. The manual mode features include momentum training and learning rate annealing and the automatic mode features an adaptive learning rate.

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:

- θ represents the parameters W, B
- μ represents the momentum coefficient
- α represents 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 any associated instability (Sutskever et al, 2014). Too much momentum can lead to instability, so we recommend incrementing the momentum slowly. The parameters that control momentum are `momentum_start`, `momentum_ramp`, and `momentum_stable`.

When using momentum updates, we recommend using the Nesterov accelerated gradient method, which uses the `nesterov_accelerated_gradient` parameter. This method modifies the updates as follows:

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

Rate Annealing

During training, the chance of oscillation or "optimum skipping" creates the need for a slower learning rate as the model approaches a minimum. As opposed to 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 required to divide the learning rate in half (e.g., 10^{-6} means that it takes 10^6 training samples to halve the learning rate).

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. To simplify hyper parameter search, specify only ρ and ϵ.

In some cases, a manually controlled (non-adaptive) learning rate and momentum specifications can lead to better results but require a hyperparameter search of up to seven parameters. If the model is built on a topology with many local minima or long plateaus, a constant learning rate may produce sub-optimal results. However, the adaptive learning rate generally produces the best results during our testing, so this option is the default.

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

Loading Data

Loading a dataset in R or Python for use with H2O is slightly different than the usual methodology. Instead of using `data.frame` or `data.table` in R, or `pandas.DataFrame` or `numpy.array` in Python, datasets must be converted into `H2OFrame` objects (distributed data frames).

Data Standardization/Normalization

Along with categorical encoding, H2O's Deep Learning preprocesses the data to standardize it for compatibility with the activation functions (refer to to the summary of each activation function's target space in **Activation Functions**).

Since the activation function generally does not 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.

For autoencoding, the data is normalized (instead of standardized) to the compact interval of $\mathcal{U}(-0.5, 0.5)$ to allow bounded activation functions like $\tanh$ to better reconstruct the data.

Convergence-based Early Stopping

Early stopping based on convergence of a user-specified metric is an especially helpful feature for finding the optimal number of epochs. By default, it uses the metrics on the validation dataset, if provided. Otherwise, training metrics are used.

- To stop model building if misclassification improves (is reduced) by less than one percent between individual scoring epochs, specify `stopping_rounds=1`, `stopping_tolerance=0.01` and `stopping_metric="misclassification"`.
- To stop model building if the simple moving average (window length 5) if the AUC improves (increases) by less than 0.1 percent for 5 consecutive scoring epochs, use `stopping_rounds=5`, `stopping_metric="AUC"`, and `stopping_tolerance=0.001`.
- To stop model building if the logloss on the validation set does not improve at all for 3 consecutive scoring epochs, specify a `validation_frame`, `stopping_rounds=3`, `stopping_tolerance=0` and `stopping_metric="logloss"`.
- To continue model building even after metrics have converged, disable this feature using `stopping_rounds=0`.
- To compute the best number of epochs with cross-validation, simply specify `stopping_rounds>0` as in the examples above, in combination with `nfolds>1`, and the main model will pick the ideal number of epochs from the convergence behavior of the `nfolds` cross-validation models.

Time-based Early Stopping

To stop model training after a given amount of seconds, specify `max_runtime_secs > 0`. This option is also available for grid searches and models with cross-validation. Note: The model(s) will likely end up with fewer epochs than specified by `epochs`.

Additional Parameters

Since there are dozens of possible parameter arguments when creating models, configuring H2O Deep Learning models may seem daunting. However, most parameters do not need to be modified; the default settings are recommended for most use cases.

There is no general rule for setting the number of hidden layers, their sizes or the number of epochs. Experimenting by building Deep Learning models using different network topologies and different datasets will lead to insights about these parameters.

For pointers on specific values and results for these (and other) parameters, many example tests are available in the H2O GitHub repository. For a full list of H2O Deep Learning model parameters and default values, refer to **Parameters**.

Use Case: MNIST Digit Classification

The following use case describes how to use H2O's Deep Learning for classification of handwritten numerals.

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. Each image 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

This example downloads and imports the training and testing datasets from a public Amazon S3 bucket (available at `https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/train.csv.gz` and same for `test.csv.gz`). The training file is 13MB, and the testing file is 2.1MB. The data is downloaded directly, so the data import speed is dependent on download speed. Files can be imported from a variety of sources, including local file systems or shared file systems such as NFS, S3 and HDFS.

Example in R

To run the MNIST example in R, use the following:

```
library(h2o)

# Sets number of threads to number of available cores
h2o.init(nthreads = -1)

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)
test <- h2o.importFile(test_file)

# Get a brief summary of the data
summary(train)
summary(test)
```

Example in Python

To run the MNIST example in Python, use the following:

```
import h2o

# Start H2O cluster with all available cores (default)
h2o.init()

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

# Get a brief summary of the data
train.describe()
test.describe()
```

Performing a Trial Run

The example below illustrates how the default settings provide the relative simplicity underlying most H2O Deep Learning model parameter configurations. The first $28^2 = 784$ values of each row represent the full image and the final value denotes the digit class.

Rectified linear activation is popular with image processing and has previously performed well on the MNIST database. Dropout has been known to enhance performance on this dataset as well, so we train our model accordingly.

To perform the trial run in R, use the following:

Example in R

```
# Specify the response and predictor columns
y <- "C785"
x <- setdiff(names(train), y)

# Encode the response column as categorical for
   multinomial classification
train[,y] <- as.factor(train[,y])
test[,y] <- as.factor(test[,y])

# Train Deep Learning model and validate on test set
model <- h2o.deeplearning(
        x = x,
        y = y,
        training_frame = train,
        validation_frame = test,
        distribution = "multinomial",
        activation = "RectifierWithDropout",
        hidden = c(32,32,32),
        input_dropout_ratio = 0.2,
        sparse = TRUE,
        l1 = 1e-5,
        epochs = 10)
```

To perform the trial run in Python, use the following:

Example in Python

```
from h2o.estimators.deeplearning import
    H2ODeepLearningEstimator

# Specify the response and predictor columns
y = "C785"
x = train.names[0:784]

# Encode the response column as categorical for
    multinomial classification
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()

# Train Deep Learning model and validate on test set
model = H2ODeepLearningEstimator(
        distribution="multinomial",
        activation="RectifierWithDropout",
        hidden=[32,32,32],
        input_dropout_ratio=0.2,
        sparse=True,
        l1=1e-5,
        epochs=10)
model.train(
        x=x,
        y=y,
        training_frame=train,
        validation_frame=test)
```

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 to estimate generalization error is N-fold cross-validation.

N-fold Cross-Validation

If the value specified for `nfolds` is a positive integer, N-fold cross-validation is performed on the `training_frame` and the cross-validation metrics are computed and stored as model output. To disable cross-validation, use `nfolds=0`, which is the default value.

To save the predicted values generated during cross-validation, set the `keep_cross_validation_predictions` parameter to true. This enables calculation of custom cross-validated performance metrics for the R or Python model.

Advanced users can also specify a `fold_column` that specifies the holdout fold associated with each row. By default, the holdout `fold_assignment` is random, but other schemes such as round-robin assignment using the modulo operator are also supported.

To run the cross-validation example in R, use the following:

Example in R

```
# Perform 5-fold cross-validation on training_frame
model_cv <- h2o.deeplearning(
        x = x,
        y = y,
        training_frame = train,
        distribution = "multinomial",
        activation = "RectifierWithDropout",
        hidden = c(32,32,32),
        input_dropout_ratio = 0.2,
        sparse = TRUE,
        l1 = 1e-5,
        epochs = 10,
        nfolds = 5)
```

To run the cross-validation example in Python, use the following:

Example in Python

```
# Perform 5-fold cross-validation on training_frame
model_cv = H2ODeepLearningEstimator(
        distribution="multinomial",
        activation="RectifierWithDropout",
        hidden=[32,32,32],
        input_dropout_ratio=0.2,
        sparse=True,
        l1=1e-5,
        epochs=10,
        nfolds=5)
model_cv.train(
        x=x,
        y=y,
        training_frame=train)
```

Extracting and Handling the Results

We can now extract the parameters of our model, examine the scoring process, and make predictions on new data. The `h2o.performance` function in R returns all pre-computed performance metrics for the training/validation set or returns cross-validated metrics for the training set, depending on model configuration.

An equivalent `model_performance` method is available in Python, as well as utility functions that return specific metrics, such as mean square error (MSE) or area under curve (AUC). Examples shown below use the previously-trained `model` and `model_cv` objects.

To view the model results in R, use the following:

Example in R

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

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

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

# Get MSE only
h2o.mse(model, valid = TRUE)

# Cross-validated MSE
h2o.mse(model_cv, xval = TRUE)
```

To view the model results in Python, use the following:

Example in Python

```
# View specified parameters of the Deep Learning model
model.params

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

model.model_performance(train=True)  # training
    metrics
model.model_performance(valid=True)  # validation
    metrics

# Get MSE only
model.mse(valid=True)

# Cross-validated MSE
model_cv.mse(xval=True)
```

The training error value is based on the parameter `score_training_samples`, which specifies the number of randomly sampled training points used for scoring (the default value is 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.

If the parameter `overwrite_with_best_model` is enabled, these scoring parameters also affect the final model. 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 consequently, which model is selected as the best model.

Once we have a satisfactory model (as determined by the validation or cross-validation metrics), use the `h2o.predict()` command to compute and store predictions on new data for additional refinements in the interactive data science process.

To view the predictions in R, use the following:

Example in R

```
# Classify the test set (predict class labels)
# This also returns the probability for each class
pred <- h2o.predict(model, newdata = test)

# Take a look at the predictions
head(pred)
```

To view the predictions in Python, use the following:

Example in Python

```
# Classify the test set (predict class labels)
# This also returns the probability for each class
pred = model.predict(test)

# Take a look at the predictions
pred.head()
```

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:54321) 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.

H2O Python users can connect to Flow the same way as R users: launch an instance of H2O, then launch your browser and enter `localhost:54321` (or your custom IP address) in the address bar. Python users can also use iPython notebook to run examples. Launch iPython notebook in Terminal using `ipython notebook`, and a browser window should automatically launch the iPython interface. Otherwise, launch your browser and enter `localhost:8888` in the address bar.

Variable Importances

To enable variable importances, setting the `variable_importances` to 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 its own methodology for computing variable importance.

H2O's Deep Learning uses the Gedeon method (Gedeon, 1997), which is disabled by default since it can be slow for large networks. If variable importance is a top priority in your analysis, consider training a Distributed Random Forest (DRF) model and comparing the generated variable importances.

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

Example in R
To generate variable importances in R, use the following:

```
# Train Deep Learning model and validate on test set
# and save the variable importances
model_vi <- h2o.deeplearning(
        x = x,
        y = y,
        training_frame = train,
        distribution = "multinomial",
        activation = "RectifierWithDropout",
        hidden = c(32,32,32),
        input_dropout_ratio = 0.2,
        sparse = TRUE,
        l1 = 1e-5,
        validation_frame = test,
        epochs = 10,
        variable_importances = TRUE)

# Retrieve the variable importance
h2o.varimp(model_vi)
```

Example in Python
To generate variable importances in Python, use the following:

```
# Train Deep Learning model and validate on test set
# and save the variable importances
model_vi = H2ODeepLearningEstimator(
        distribution="multinomial",
        activation="RectifierWithDropout",
        hidden=[32,32,32],
        input_dropout_ratio=0.2,
        sparse=True,
        l1=1e-5,
        epochs=10,
        variable_importances=True)

model_vi.train(
        x=x,
```

```
        y=y,
        training_frame=train,
        validation_frame=test)

# Retrieve the variable importance
model_vi.varimp()
```

Java Model

To access Java code to use to build the current model in Java, click the PREVIEW POJO button at the bottom of the model results. This button generates a POJO model that can be used in a Java application independently of H2O.

To download the model:

1. Open the terminal window.
2. Create a directory where the model will be saved.
3. Set the new directory as the working directory.
4. Follow the curl and java compile commands displayed in the instructions at the top of the Java model.

For more information on how to use an H2O POJO, refer to the **POJO Quick Start Guide** at `https://github.com/h2oai/h2o-3/blob/master/h2o-docs/src/product/howto/POJO_QuickStart.md`.

Grid Search for Model Comparison

H2O supports model tuning in grid search by allowing users to specify sets of values for parameter arguments and observe changes in model behavior.

In the R and Python examples below, three different network topologies and two different ℓ_1 norm weights are specified. 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. Note that the models will most likely converge before the default value of `epochs`, since early stopping is enabled.

Grid search 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 **Parameters** for parameter descriptions and configurable values.

Cartesian Grid Search

To run a Cartesian hyper-parameter grid search in R, use the following:

Example in R

```
hidden_opt <- list(c(32,32), c(32,16,8), c(100))
l1_opt <- c(1e-4,1e-3)
hyper_params <- list(hidden = hidden_opt, l1 = l1_opt)

model_grid <- h2o.grid("deeplearning",
    grid_id = "mygrid",
    hyper_params = hyper_params,
    x = x,
    y = y,
    distribution = "multinomial",
    training_frame = train,
    validation_frame = test,
    score_interval = 2,
    epochs = 1000,
    stopping_rounds = 3,
    stopping_tolerance = 0.05,
    stopping_metric = "misclassification")
```

To run a Cartesian hyper-parameter grid search in Python, use the following:

Example in Python

```
hidden_opt = [[32,32],[32,16,8],[100]]
l1_opt = [1e-4,1e-3]
hyper_parameters = {"hidden":hidden_opt, "l1":l1_opt}

from h2o.grid.grid_search import H2OGridSearch
model_grid = H2OGridSearch(H2ODeepLearningEstimator,
    hyper_params=hyper_parameters)
model_grid.train(x=x, y=y,
    distribution="multinomial", epochs=1000,
    training_frame=train, validation_frame=test,
```

```
    score_interval=2, stopping_rounds=3,
    stopping_tolerance=0.05,
    stopping_metric="misclassification")
```

To view the results of the grid search, use the following:

Example in R

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

# print out the Test MSE for all of the models
for (model_id in model_grid@model_ids) {
  mse <- h2o.mse(h2o.getModel(model_id), valid = TRUE)
  print(sprintf("Test set MSE: %f", mse))
}
```

Example in Python

```
# print model grid search results
model_grid

for model in model_grid:
    print model.model_id + " mse: " + str(model.mse())
```

Random Grid Search

If the search space is too large (i.e., you don't want to restrict the parameters too much), you can also let the Grid Search make random model selections for you. Just specify how many models (and/or how much training time) you want, and a seed to make the random selection deterministic:

Example in R

```
hidden_opt = lapply(1:100, function(x)10+sample(50,
    sample(4), replace=TRUE))
l1_opt = seq(1e-6,1e-3,1e-6)
hyper_params <- list(hidden = hidden_opt, l1 = l1_opt)
```

```
search_criteria = list(strategy = "RandomDiscrete",
    max_models = 10, max_runtime_secs = 100,
    seed=123456)

model_grid <- h2o.grid("deeplearning",
    grid_id = "mygrid",
    hyper_params = hyper_params,
    search_criteria = search_criteria,
    x = x,
    y = y,
    distribution = "multinomial",
    training_frame = train,
    validation_frame = test,
    score_interval = 2,
    epochs = 1000,
    stopping_rounds = 3,
    stopping_tolerance = 0.05,
    stopping_metric = "misclassification")
```

Example in Python

```

hidden_opt =
    [[17,32],[8,19],[32,16,8],[100],[10,10,10,10]]
l1_opt = [s/1e6 for s in range(1,1001)]
hyper_parameters = {"hidden":hidden_opt, "l1":l1_opt}
search_criteria = {"strategy":"RandomDiscrete",
    "max_models":10, "max_runtime_secs":100,
    "seed":123456}

from h2o.grid.grid_search import H2OGridSearch
model_grid = H2OGridSearch(H2ODeepLearningEstimator,
    hyper_params=hyper_parameters,
    search_criteria=search_criteria)
model_grid.train(x=x, y=y,
    distribution="multinomial", epochs=1000,
    training_frame=train, validation_frame=test,
    score_interval=2, stopping_rounds=3,
    stopping_tolerance=0.05,
    stopping_metric="misclassification")
```

Checkpoint Models

To resume model training, use checkpoint model keys (`model_id`) to incrementally train a specific model using more iterations, more data, different data, and so forth. To further train the initial model, use it (or its key) as a checkpoint argument for a new model.

To get the best possible model in a general multi-node setup, we recommend building a model with `train_samples_per_iteration=-2` (default, auto-tuning) and saving it to disk so that youll have at least one saved model.

To improve this initial model, start from the previous model and add iterations by building another model, specifying `checkpoint=previous_model_id`, and changing `train_samples_per_iteration`, `target_ratio_comm_to_comp`, or other parameters. Many parameters can be changed between checkpoints, especially those that affect regularization or performance tuning.

Checkpoint restart suggestions:

1. For multi-node only: Leave `train_samples_per_iteration=-2`, increase `target_ratio_comm_to_comp` from 0.05 to 0.25 or 0.5 (more communication). This should lead to a better model when using multiple nodes. Note: No affect on single-node performance at all, since there is no actual communication needed.

2. For both single and multi-node (bagging-like): Explicitly set `train_samples_per_iteration=N`, where N is the number of training samples for the whole cluster to train with for one iteration. Each of the n nodes will then train on N/n randomly chosen rows for each iteration. Obviously, a good choice for N depends on the dataset size and the model complexity. Refer to the logs to see what values of N are used in option 1 (when auto-tuning is enabled). Typically, option 1 is sufficient.

3. For both single and multi-node: Change regularization parameters such as `l1`, `l2`, `max_w2`, `input_dropout_ratio`, `hidden_dropout_ratios`. For best results, build the first model with `RectifierWithDropout` and `input_dropout_ratio=0` and `hidden_dropout_ratios` of all `0`s, just to be able to enable dropout regularization later. Hidden dropout is often used for initial models, since it often improves generalization. Input dropout is especially useful if there is some noise in the input.

Options 1 and 3 should result in a good model. Of course, grid search can be used with checkpoint restarts to scan a broad range of good continuation models.

In the R example below, `model_grid@model_ids[[1]]` represents the highest-performing model from the grid search used for additional training. For checkpoint restarts, the response column, training, and validation datasets must match. In addition, the model architecture, such as `hidden = c(32,32,32)` in the example below, must match.

To restart training in R using a checkpoint model, use the following:

Example in R

```
# Re-start the training process on a saved DL model
# using the `checkpoint` argument
model_chkp <- h2o.deeplearning(
        x = x,
        y = y,
        training_frame = train,
        validation_frame = test,
        distribution = "multinomial",
        checkpoint = model@model_id,
        activation = "RectifierWithDropout",
        hidden = c(32,32,32),
        input_dropout_ratio = 0.2,
        sparse = TRUE,
        l1 = 1e-5,
        epochs = 20)
```

The Python example uses the "trial run" model as the checkpoint model.

Example in Python

```
# Re-start the training process on a saved DL model
# using the `checkpoint` argument
model_chkp = H2ODeepLearningEstimator(
        checkpoint=model,
        distribution="multinomial",
        activation="RectifierWithDropout",
        hidden=[32,32,32],
        input_dropout_ratio=0.2,
        sparse=True,
```

```
        l1=1e-5,
        epochs=20)

model_chkp.train(
        x=x,
        y=y,
        training_frame=train,
        validation_frame=test)
```

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 a model by running the following commands.

To save a model in R, use the following:

Example in R

```
# Specify a model and the file path where it is to be
    saved. If no path is specified, the model will be
    saved to the current working directory
model_path <- h2o.saveModel(object = model,
                            path=getwd(), force = TRUE
                                )

print(model_path)
# /tmp/mymodel/DeepLearning_model_R_1441838096933
```

To save a model in Python, use the following:

Example in Python

```
# Specify a model and the file path where it is to be
    saved. If no path is specified, the model will be
    saved to the current working directory
model_path = h2o.save_model(
        model = model,
        #path = "/tmp/mymodel",
        force = True)

print model_path
# /tmp/mymodel/DeepLearning_model_python_1441838096933
```

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).

To load a saved model in R, use the following:

Example in R

```
# Load model from disk
saved_model <- h2o.loadModel(model_path)
```

To load a saved model in Python, use the following:

Example in Python

```
# Load model from disk
saved_model = h2o.load_model(model_path)
```

You can also use the following commands to retrieve a model from its H2O key. This is useful if you have created an H2O model using the web interface and want to continue the modeling process in R.

To retrieve a model in R using its key, use the following:

Example in R

```
# Retrieve model by H2O key
model <- h2o.getModel(model_id = model_chkp@model_id)
```

To retrieve a model in Python using its key, use the following:

Example in Python

```
# Retrieve model by H2O key
model = h2o.get_model(model_id=model_chkp._id)
```

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 (which took about four hours) on four compute nodes, we obtain a test set error of 0.87%.

After training for $8,000$ epochs (which took about ten hours) on ten nodes, we obtain a test set error of 0.83%, which is the current world-record, notably achieved using a distributed configuration and with a simple 1-liner from R:

Example in R

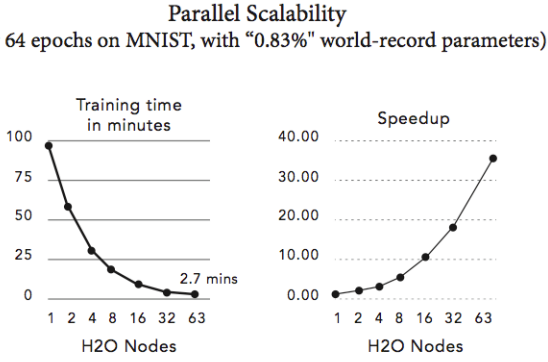
```
#World Record run used epochs=8000

model <- h2o.deeplearning(x=x, y=y,
        training_frame=train_hex, validation_frame=
           test_hex,
        activation="RectifierWithDropout",
        hidden=c(1024,1024,2048), epochs=10,
        input_dropout_ratio=0.2, l1=1e-5, max_w2=10,
        train_samples_per_iteration=-1,
        classification_stop=-1, stopping_rounds=0)
```

Computational Performance

There are many parameters that affect the computational performance of H2O Deep Learning, but the default values should result in good performance for most problems.

An in-depth study of the computational performance characteristics of H2O Deep Learning with complete code examples and results can be found in our blog post, **Definitive Performance Tuning Guide for Deep Learning**, available at `http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/deep-learning.html#deep-learning-tuning-guide`.

The parallel scalability of H2O for the MNIST dataset on 1 to 63 compute nodes is shown in the figure below.



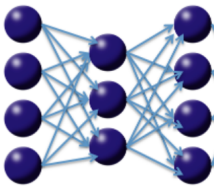
Deep Autoencoders

This section describes the use of deep autoencoders for Deep Learning.

Nonlinear Dimensionality Reduction

Previous sections 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).

Based on the diagram of a three-layer neural network with one hidden layer below, if our input data is treated 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, called a deep autoencoder, has been used extensively for unsupervised, layer-wise pre-training of supervised Deep Learning tasks. Here we discuss the autoencoder's ability to discover anomalies in data.

Use Case: Anomaly Detection

For the deep autoencoder model described above, if enough training data resembling some underlying pattern is provided, the network will train itself to easily learn the identity when confronted with that pattern. However, if an anomalous test point does not match the learned pattern, the autoencoder will likely have a high error rate in reconstructing this data, indicating anomalous data.

This framework is used 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 directly into the H2O cluster, as shown below. Each row represents a single heartbeat.

Example in R

```
# Import ECG train and test data into the H2O cluster
train_ecg <- h2o.importFile(
        path = "http://h2o-public-test-data.s3.amazonaws.com
            /smalldata/anomaly/ecg_discord_train.csv",
        header = FALSE,
        sep = ",")
test_ecg <- h2o.importFile(
        path = "http://h2o-public-test-data.s3.amazonaws.com
            /smalldata/anomaly/ecg_discord_test.csv",
        header = FALSE,
        sep = ",")

# Train deep autoencoder learning model on "normal"
# training data, y ignored
anomaly_model <- h2o.deeplearning(
        x = names(train_ecg),
        training_frame = train_ecg,
        activation = "Tanh",
        autoencoder = TRUE,
        hidden = c(50,20,50),
        sparse = TRUE,
        l1 = 1e-4,
        epochs = 100)

# Compute reconstruction error with the Anomaly
# detection app (MSE between output and input layers)
recon_error <- h2o.anomaly(anomaly_model, test_ecg)

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

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

To run the anomaly detection example in Python, use the following:

Example in Python

```
# Import ECG train and test data into the H2O cluster
from h2o.estimators.deeplearning import
    H2OAutoEncoderEstimator

train_ecg = h2o.import_file("http://h2o-public-test-data.s3.
    amazonaws.com/smalldata/anomaly/ecg_discord_train.csv")
test_ecg = h2o.import_file("http://h2o-public-test-data.s3.
    amazonaws.com/smalldata/anomaly/ecg_discord_test.csv")


# Train deep autoencoder learning model on "normal"
# training data, y ignored
anomaly_model = H2OAutoEncoderEstimator(
        activation="Tanh",
        hidden=[50,50,50],
        sparse=True,
        l1=1e-4,
        epochs=100)
anomaly_model.train(
        x=train_ecg.names,
        training_frame=train_ecg)

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


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

Stacked Autoencoder

It can be difficult to obtain convergence for deep autoencoders, especially since H2O attempts to train all layers at once without imposing symmetry conditions on the network topology (arbitrary configuration of layers is allowed). To train a deep autoencoder layer by layer, follow the `R` code example here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-r/tests/testdir_algos/deeplearning/runit_deeplearning_stacked_autoencoder_large.R`

Unsupervised Pretraining with Supervised Fine-Tuning

Sometimes, there's much more unlabeled data than labeled data. It this case, it might make sense to train an autoencoder model on the unlabeled data and then fine-tune the learned model with the available labels. In H2O, you would train an autoencoder model with `autoencoder` enabled, and then you can transfer its state to a supervised regular Deep Learning model by specifying `pretrained_autoencoder`. You can seen an `R` example here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-r/tests/testdir_algos/deeplearning/runit_deeplearning_autoencoder_large.R`,

and the corresponding `Python` example here:

`https://github.com/h2oai/h2o-3/blob/master/h2o-py/tests/testdir_algos/deeplearning/pyunit_autoencoderDeepLearning_large.py`

Parameters

Logical indicates the parameter requires a value of either `TRUE` or `FALSE`.

- `x`: Specifies the vector containing the names of the predictors in the model. No default.
- `y`: Specifies the name of the response variable in the model. No default.
- `training_frame`: Specifies an `H2OFrame` object containing the variables in the model. No default.
- `model_id`: (Optional) Specifies the unique ID associated with the model. If a value is not specified, an ID is generated automatically.
- `overwrite_with_best_model`: Logical. If enabled, overwrites the final model with the best model scored during training. The default is true.
- `validation_frame`: (Optional) Specifies an `H2OFrame` object representing the validation dataset used for the confusion matrix. If a value is not specified and `nfolds = 0`, the training data is used by default.

- `checkpoint`: (Optional) Specifies the model checkpoint (either an `H2ODeepLearningModel` or a key) from which to resume training.
- `autoencoder`: Logical. Enables autoencoder. The default is false. Refer to the **Deep Autoencoders** section for more details.
- `pretrained_autoencoder`: (Optional) Pretrained autoencoder model (either an `H2ODeepLearningModel` or a key) to initialize the model state of a supervised DL model with.
- `use_all_factor_levels`: Logical. Uses all factor levels of categorical variance. Otherwise, omits the first factor level without loss of accuracy. Useful for variable importances and auto-enabled for autoencoder. The default is true. Refer to the **Deep Autoencoders** section for more details.
- `activation`: Specifies the nonlinear, differentiable activation function used in the network. The options are `Tanh, TanhWithDropout, Rectifier, RectifierWithDropout, Maxout,` or `MaxoutWithDropout`. The default is `Rectifier`. Refer to the **Activation and Loss Functions** and **Regularization** sections for more details.
- `hidden`: Specifies 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 is generated. The middle hidden layer will have 200 neurons and the first and third hidden layers will have 100 neurons each. The default is `c(200,200)`. For grid search, use the following format: `list(c(10,10), c(20,20))`. Refer to the section on **Performing a Trial Run** for more details.
- `epochs`: Specifies the number of iterations or passes over the training dataset (can be fractional). For initial grid searches, we recommend starting with lower values. The value allows continuation of selected models and can be modified during checkpoint restarts. The default is 10.
- `train_samples_per_iteration`: Specifies the number of training samples (globally) per MapReduce iteration. The following special values are also supported:
  - `0` (one epoch)
  - `-1` (all available data including replicated training data);
  - `-2` (auto-tuning; default)

  Refer to **Specifying the Number of Training Samples** for more details.
- `seed`: Specifies the random seed controls sampling and initialization. Reproducible results are only expected with single-threaded operations (i.e. 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. The default is a randomly generated number.
- `adaptive_rate`: Logical. Enables adaptive learning rate (ADADELTA). The default is true. Refer to **Adaptive Learning** for more details.

- `rho`: Specifies the adaptive learning rate time decay factor. This parameter is similar to momentum and relates to the memory for prior weight updates. Typical values are between 0.9 and 0.999. The default value is 0.99. Refer to **Adaptive Learning** for more details.
- `epsilon`: When enabled, specifies the second of two hyperparameters for the adaptive learning rate. This parameter is similar to learning rate annealing during initial training and momentum at later stages where it assists progress. Typical values are between 1e-10 and 1e-4. This parameter is only active if `adaptive_rate` is enabled. The default is 1e-8. Refer to **Adaptive Learning** for more details.
- `rate`: Specifies the learning rate, α. Higher values lead to less stable models, while lower values result in slower convergence. The default is 0.005.
- `rate_annealing`: Reduces the learning rate to "freeze" into local minima in the optimization landscape. 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. If adaptive learning is disabled, the default value is 1e-6. Refer to **Rate Annealing** for more details.
- `rate_decay`: Controls the change of learning rate across layers. The learning rate decay factor between layers is calculated as (L-th layer: `rate` * `rate_decay`^ (L-1)). If adaptive learning is disabled, the default is 1.0.
- `momentum_start`: Controls the amount of momentum at the beginning of training when adaptive learning is disabled. The default is 0. Refer to **Momentum Training** for more details.
- `momentum_ramp`: If the value for `momentum_stable` is greater than `momentum_start`, increases momentum for the duration of the learning. The ramp is measured in the number of training samples and can be enabled when adaptive learning is disabled. The default is 1 million (1e6). Refer to **Momentum Training** for more details.
- `momentum_stable`: Specifies the final momentum value after the number of training samples specified for `momentum_ramp` when adaptive learning is disabled. The training momentum is applied for any additional training. The default is 0. Refer to **Momentum Training** for more details.
- `nesterov_accelerated_gradient`: Logical. Enables the Nesterov accelerated gradient descent method, which is a modification to the 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 the adaptive learning rate is disabled. When adaptive learning is disabled, the default is true. Refer to **Momentum Training** for more details.
- `input_dropout_ratio`: Specifies the fraction of the features for each training row to omit from training to improve generalization. The default is 0, which always uses all features. Refer to **Regularization** for more details.

- `hidden_dropout_ratios`: Specifies the fraction of the inputs for each hidden layer to omit from training to improve generalization. The default is 0.5 for each hidden layer. Refer to **Regularization** for more details.
- `categorical_encoding`: Specify one of the following encoding schemes for handling categorical features:
  - `auto`: Allow the algorithm to decide
  - `one_hot_internal`: On the fly N+1 new cols for categorical features with N levels (default)
  - `binary`: No more than 32 columns per categorical feature
  - `eigen`: k columns per categorical feature, keeping projections of one-hot-encoded matrix onto k-dim eigen space only
- `l1`: Specifies the ℓ_1 (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 **Regularization** for more details.
- `l2`: L2 regularization, which constrains the sum of the squared weights and can add stability and improve generalization by reducing many weights). The default is 0, which disables L2 regularization. Refer to **Regularization** for more details.
- `max_w2`: Specifies the maximum for the sum of the squared incoming weights for a neuron. This tuning parameter is especially useful for unbound activation functions such as Maxout or Rectifier. The default, which is positive infinity, leaves this maximum unbounded.
- `initial_weight_distribution`: Specifies the distribution from which to draw the initial weights. Select `Uniform`, `UniformAdaptive` or `Normal`. The default is `UniformAdaptive`. Refer to **Initialization** for more details.
- `initial_weight_scale`: Specifies the scale of the distribution function for uniform or normal distributions. For uniform distributions, the values are drawn uniformly from `initial_weight_scale`, `initial_weight_scale`. For normal distributions, the values are drawn from a normal distribution with a standard deviation of `initial_weight_scale`. The default is 1. Refer to **Initialization** for more details.
- `loss`: Specifies the loss option: `Automatic`, `CrossEntropy` (classification only), `Quadratic`, `Absolute`, or `Huber`. The default is `Automatic`. Refer to **Activation and Loss Functions** for more details.
- `distribution`: Specifies the distribution function of the response: `AUTO`, `bernoulli`, `multinomial`, `poisson`, `gamma`, `tweedie`, `laplace`, `quantile`, `huber`, or `gaussian`.
- `quantile_alpha`: Desired quantile for quantile regression (from 0.0 to 1.0) when `distribution = "quantile"`. The default is 0.5 (median, same as `distribution = "laplace"`).

- `tweedie_power`: Specifies the Tweedie power when `distribution` is `tweedie`. The range is from 1.0 to 2.0.
- `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.
- `score_interval`: Specifies the minimum time (in seconds) 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`. The default is 5.
- `score_training_samples`: Specifies the number of training samples to randomly sample for scoring. To select the entire training dataset, specify `0`. The default is 10000.
- `score_validation_samples`: Specifies the number of validation dataset points for scoring. Can be randomly sampled or stratified if `balance_classes` is enabled and `score_validation_sampling` is `Stratified`. To select the entire validation dataset, specify `0`, which is the default.
- `score_duty_cycle`: Specifies the maximum duty cycle fraction for model scoring on both training and validation samples and diagnostics such as computation of feature importances. Lower values result in more training, while higher values produce more scoring. The default is 0.1.
- `classification_stop`: Specifies the stopping criterion for classification error (1 - accuracy) on the training data scoring dataset. When the error is at or below this threshold, training stops. The default is 0. To disable, specify `-1`.
- `regression_stop`: Specifies the stopping criterion for regression error (MSE) on the training data scoring dataset. When the error is at or below this threshold, training stops. The default is 1e-6. To disable, specify `-1`.
- `stopping_rounds`: Early stopping based on convergence of `stopping_metric`. Stop if simple moving average of length k of the `stopping_metric` does not improve for k:=`stopping_rounds` scoring events. Can only trigger after at least 2k scoring events. To disable, specify `0`.
- `stopping_metric`: Metric to use for early stopping (`AUTO`: `logloss` for classification, `deviance` for regression). Can be any of `AUTO`, `deviance`, `logloss`, `misclassification`, `lift_top_gain`, `MSE`, `AUC`, and `mean_per_class_error`.
- `stopping_tolerance`: Relative tolerance for metric-based stopping criterion Relative tolerance for metric-based stopping criterion (stop if relative improvement is not at least this much).
- `max_runtime_secs`: Maximum allowed runtime in seconds for model training. Use 0 to disable.
- `missing_values_handling`: Handling of missing values. Either `Skip` or `MeanImputation` (default).

- `quiet_mode`: Logical. Enables quiet mode for less output to standard output. The default is false.
- `max_hit_ratio_k`: For multi-class only. Specifies the maximum number (top K) of predictions to use for hit ratio computation. To disable, specify `0`. The default is 10.
- `balance_classes`: Logical. For imbalanced data, the training data class counts can be artificially balanced by over-sampling the minority classes and under-sampling the majority classes so that each class contains the same number of observations. This can result in improved predictive accuracy. Over-sampling uses replacement, rather than simulating new observations. The `max_after_balance_size` parameter specifies the total number of observations after balancing. The default is false.
- `class_sampling_factors`: Specifies the desired over/under-sampling ratios per class (in lexicographic order). Only applies to classification when `balance_classes` is enabled. If not specified, the ratios are automatically computed to obtain class balancing during training.
- `max_after_balance_size`: When classes are balanced, limits 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 and can be less than 1.0. The default is 5.
- `score_validation_sampling`: Specifies the method used to sample the validation dataset for scoring. The options are `Uniform` and `Stratified`. The default is `Uniform`.
- `variable_importances`: Logical. Computes variable importances for input features using the Gedeon method. Uses the weights connecting the input features to the first two hidden layers. The default is false, since this can be slow for large networks.
- `fast_mode`: Logical. Enables fast mode, a minor approximation in back-propagation that should not significantly affect results. The default is true.
- `ignore_const_cols`: Logical. Ignores constant training columns, since no information can be gained anyway. The default is true.
- `force_load_balance`: Logical. Forces extra load balancing to increase training speed for small datasets to keep all cores busy. The default is true.
- `replicate_training_data`: Logical. Replicates the entire training dataset on every node for faster training on small datasets. The default is true.
- `single_node_mode`: Logical. Runs Deep Learning on a single node for fine-tuning model parameters. Can be useful for faster convergence during checkpoint restarts after training on a very large number of nodes (for fast initial convergence). The default is false.
- `shuffle_training_data`: Logical. Shuffles 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). The default is false.

- `sparse`: Logical. Enables sparse data handling (more efficient for data with lots of 0 values). The default is false.
- `col_major`: (Deprecated) Logical. Uses a column major weight matrix for the input layer; can speed up forward propagation, but may slow down backpropagation. The default is false.
- `average_activation`: Specifies the average activation for the sparse autoencoder (Experimental). The default is 0.
- `sparsity_beta`: Specify the sparsity-based regularization optimization (Experimental). The default is 0.
- `max_categorical_features`: Specifies the maximum number of categorical features in a column, enforced via hashing (Experimental). The default is $2^{31}-1$ (`Integer.MAX_VALUE` in Java).
- `reproducible`: Logical. Forces reproducibility on small data; slow, since it only uses one thread. The default is false.
- `export_weights_and_biases`: Logical. Exports the neural network weights and biases as an `H2OFrame`. The default is false.
- `offset_column`: Specifies 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`: Specifies the weights column by column name, which 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`. 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) Specifies the number of folds for cross-validation. The default is 0, which disables cross-validation.
- `fold_column`: (Optional) Specifies the name of the column with the cross-validation fold index assignment per observation; the folds are supplied by the user.
- `fold_assignment`: Specifies the cross-validation fold assignment scheme if `nfolds` is greater than zero and `fold_column` is not specified. The options are `AUTO`, `Random`, `Stratified` or `Modulo`.
- `keep_cross_validation_predictions`: Logical. Specify whether to keep the predictions of the cross-validation models. The default is false.

Common R Commands

- `library(h2o)`: Imports the H2O R package.
- `h2o.init()`: Connects to (or starts) an H2O cluster.
- `h2o.shutdown()`: Shuts down the H2O cluster.
- `h2o.importFile(path)`: Imports a file into H2O.
- `h2o.deeplearning(x,y,training_frame,hidden,epochs)`: Creates a Deep Learning model.
- `h2o.grid(algorithm,grid_id,...,hyper_params = list())`: Starts H2O grid support and gives results.
- `h2o.predict(model, newdata)`: Generate predictions from an H2O model on a test set.

Common Python Commands

- `import h2o`: Imports the H2O Python package.
- `h2o.init()`: Connects to (or starts) an H2O cluster.
- `h2o.shutdown()`: Shuts down the H2O cluster.
- `h2o.import_file(path)`: Imports a file into H2O.
- `model = H2ODeepLearningEstimator(hidden, epochs)`: Creates a Deep Learning model.
- `model.train(x, y, training_frame)`: Trains our Deep Learning model.
- `h2o.predict(model, newdata)`: Generate predictions from an H2O model on a test set.

Acknowledgments

We would like to acknowledge the following individuals for their contributions to this booklet: Viraj Parmar and Anisha Arora.

References

Feng Niu, Benjamin Recht, Christopher R, and Stephen J. Wright. **Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent**. In *In NIPS*, 2011

G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. **Improving neural networks by preventing co-adaptation of feature detectors**. 2012. URL http://arxiv.org/pdf/1207.0580.pdf

Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. **On the importance of initialization and momentum in deep learning**. 2014. URL http://www.cs.toronto.edu/~fritz/absps/momentum.pdf

Matthew D. Zeiler. **ADADELTA: An Adaptive Learning Rate Method**. 2012. URL http://arxiv.org/pdf/1212.5701v1.pdf

Yann LeCun, Corinna Cortes, and Christopher J.C Burgres. **The MNIST Database**. URL http://yann.lecun.com/exdb/mnist/

G.E. Hinton and R.R. Salakhutdinov. **Reducing the Dimensionality of Data with Neural Networks**. *Science*, 313:504–507, 2006. URL http://www.cs.toronto.edu/~hinton/science.pdf

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

H2O.ai Team. **H2O website**, 2019. URL http://h2o.ai

H2O.ai Team. **H2O documentation**, 2019. URL http://docs.h2o.ai

H2O.ai Team. **H2O GitHub Repository**, 2019. URL https://github.com/h2oai

H2O.ai Team. **H2O Datasets**, 2019. URL http://data.h2o.ai

H2O.ai Team. **H2O JIRA**, 2019. URL https://jira.h2o.ai

H2O.ai Team. **H2Ostream**, 2019. URL https://groups.google.com/d/forum/h2ostream

H2O.ai Team. **H2O R Package Documentation**, 2019. URL http://h2o-release.s3.amazonaws.com/h2o/latest_stable_Rdoc.html

Authors

Arno Candel

Arno is the Chief Architect of H2O, a distributed and scalable open-source machine learning platform. He is also the main author of H2O Deep Learning. Arno holds a PhD and Masters summa cum laude in Physics from ETH Zurich, Switzerland. He has authored dozens of scientific papers and is a sought-after conference speaker. Arno was named 2014 Big Data All-Star by Fortune Magazine. Follow him on Twitter: @ArnoCandel.

Erin LeDell

Erin is a Statistician and Machine Learning Scientist at H2O.ai. She is the main author of H2O Ensemble. Erin received her Ph.D. in Biostatistics with a Designated Emphasis in Computational Science and Engineering from University of California, Berkeley. Her research focuses on ensemble machine learning, learning from imbalanced binary-outcome data, influence curve based variance estimation and statistical computing. She also holds a B.S. and M.A. in Mathematics. Follow her on Twitter: @ledell.

Angela Bartz

Angela is the doc whisperer at H2O.ai. With extensive experience in technical communication, she brings our products to life by documenting the many features and functionality of H2O. Having worked for companies both large and small, she is an expert at understanding her audience and translating complex ideas to user-friendly content. Follow her on Twitter: @abartztweet.