Deep Learning with H2O's R Package

Arno Candel Viraj Parmar

September 2014

Contents

1 Introduction

This documentation presents the Deep Learning framework in H2O, as presented through the H2O R interface. Further documentation on H2O's system and algorithms can be found at the 0xdata website at http://docs.0xdata.com, especially the "R User documentation". The datasets, R code and instructions for this document can be found at the H2O Github repository at
https://github.com/0xdata/h2o/tree/master/docs/deeplearning/DeepLearningRVignetteDemo/.
This introductory section provides instructions on getting H2O started from R, followed by a brief overview of deep learning.

1.1 Installation

To install H2O, follow the "Download" link on H2O's website at http://h2o.ai/. For multi-node operation, download the H2O zip file and deploy H2O on your cluster, following instructions from the "Full Documentation". For single-node operation, follow the instructions in the "Install in R" tab. Open your R Console and run the following to install and start H2O directly from R:

```
# The following two commands remove any previously installed H2O packages for R.
if ("package:h2o" %in% search()) { detach("package:h2o", unload=TRUE) }

if ("h2o" %in% rownames(installed.packages())) { remove.packages("h2o") }

# Next, we download, install and initialize the H2O package for R (filling in the
*'s with the latest version number obtained from the H2O download page)
install.packages("h2o", repos=(c("http://s3.amazonaws.com/h2o-release/h2o/master/
****/R", getOption("repos"))))

library(h2o)
```

Initialize H2O with

```
h2o_server = h2o.init()
```

With this command, the H2O R module will start an instance of H2O automatically at localhost:54321. Alternatively, to specify a connection with an existing H2O cluster node (other than localhost at port 54321) you must explicitly state the IP address and port number in the `h2o.init()` call. An example is given below, but do not directly paste; you should specify the IP and port number appropriate to your specific environment.

```
h2o_cluster = h2o.init(ip = "192.555.1.123", port = 12345, startH2O = FALSE)
```

An automatic demo is available to see h2o.deeplearning at work. Run the following command to observe an example binary classification model built through H2O's Deep Learning.

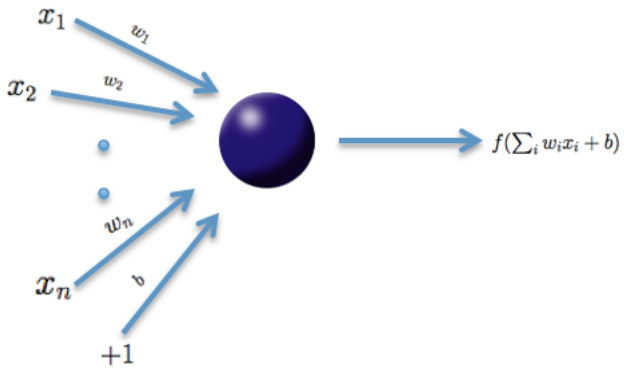
```
demo(h2o.deeplearning)
```

1.2 Support

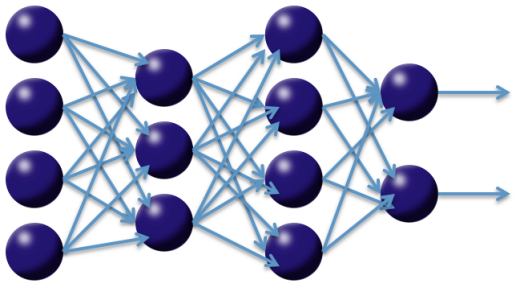
Users of the H2O package may submit general enquiries and bug reports to the 0xdata support address. Alternatively, specific bugs or issues may be filed to the 0xdata JIRA.

1.3 Deep learning overview

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 $\alpha = \sum_{j=1}^{n} w_i x_i + b$ 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: beginning with an input layer to match the feature space; followed by multiple layers of nonlinearity; and terminating 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 specificially, 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. Such 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).

2 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.

2.1 Summary of features

H2O's Deep Learning functionalities include:

- purely supervised training protocol for regression and classification tasks
- multi-threaded parallel computation to be run on either a single node or a multi-node cluster
- advanced training options including adaptive learning, momentum training, rate annealing, and dropout
- regularization options to prevent model overfitting
- fast and memory-efficient Java implementations of the underlying algorithms
- elegant web interface to mirror the model building and scoring process running in R
- grid search for hyperparameter optimization and model selection
- model checkpointing
- model export in plain java code for deployment in production environments
- additional parameters for model tuning
- deep autoencoders for unsupervised feature learning and anomaly detection capabilities

2.2 Training protocol

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

2.2.1 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.

2.2.2 Activation and loss functions

In the introduction we introduced the nonlinear activation function f, for which the choices are summarized in Table 1. Note here that x_i and w_i denonote 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	$f(\cdot) = \frac{e^{\alpha} - e^{-\alpha}}{e^{\alpha} + e^{-\alpha}}$	$f(\cdot) \in [-1, 1]$
Rectified Linear	$f(\cdot) = \max(0, \alpha)$	$f(\cdot) \in \mathbb{R}_+$
Maxout	$f(\cdot) = \max(w_i x_i + b)$, rescale if $\max f(\cdot) \geq 1$	$f(\cdot) \in [-\infty, 1]$

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 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
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

2.2.3 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

Initialize W, B
Iterate until convergence criterion reached
  Get training example i
  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. 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 summmary.

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

---

Initialize global model parameters W, B
Distribute training data $\mathcal{T}$ across nodes (can be disjoint or replicated)
Iterate until convergence criterion reached
  For nodes n with training subset $\mathcal{T}_n$, do in parallel:
    Obtain copy of the global model parameters W_n, B_n
    Select active subset $\mathcal{T}_{na} \subset \mathcal{T}_n$ (user-given number of samples per iteration)
    Partition $\mathcal{T}_{na}$ into $\mathcal{T}_{nac}$ by cores n_c
    For cores n_c on node n, do in parallel:
      Get training example $i \in \mathcal{T}_{nac}$
      Update all weights $w_{jk} \in W_n$, biases $b_{jk} \in B_n$

$w_{jk} := w_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial w_{jk}}$

$b_{jk} := b_{jk} - \alpha \frac{\partial L(W,B|j)}{\partial b_{jk}}$

Set $W, B := \text{Avg}_n\ W_n$, $\text{Avg}_n\ B_n$

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.

2.2.4 Specifying the number of training samples per iteration

H2O Deep Learning is scalable and can take advantage of a large cluster of compute nodes. There are three modes in which to operate. The default behavior is to let every node train on the entire (replicated) dataset, but automatically locally shuffling (and/or using a subset of) the training examples for each iteration. For datasets that don't fit into each node's memory (also depending on the heap memory specified by the -Xmx 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 the case where slow final convergence is observed due to the presence of too many nodes, but we've never seen this become necessary.
The number of training examples (globally) presented to the distributed SGD worker nodes between model averaging is controlled by the important parameter `train_samples_per_iteration`. One special value is -1, which results in all nodes processing all their local training data per iteration. Note that if `replicate_training_data` is enabled (true by default), this will result in training N epochs per iteration on N nodes, otherwise 1 epoch will be trained per iteration. Another special value is 0, which always results in 1 epoch per iteration, independent of the number of compute nodes. In general, any user-given positive number is permissible for this parameter. For large datasets, it might make sense to specify 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 4 nodes, then each node will process $25,000$ examples per iteration, and it will take 40 such distributed iterations to process one epoch. If the value is set too high, it might take too long between synchronization and model convergence can be slow. If the value is set too low, network communication overhead will dominate the runtime, and computational performance will suffer. The special value of -2 (the default) enables auto-tuning of 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. Note that this parameter can affect the convergence rate during training.

2.3 Regularization

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

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

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

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 chosen to be very small, for example 10^{-5}. The second type of regularization available for deep learning is a recent innovation called dropout (Hinton et al., 2012). Dropout constrains the online optimization such that during forward propagation for a given training example, each neuron in the network suppresses its activation with probability P, generally taken to be less than 0.2 for input neurons and up to 0.5 for hidden neurons. The effect is twofold: as with ℓ_2 regularization, the network weight values are scaled toward 0; furthermore, each training example trains a different model, albeit sharing the same global parameters. Thus dropout allows an exponentially large number of models to be averaged as an ensemble, which can prevent overfitting and improve generalization. Note that input dropout can be especially useful when the feature space is large and noisy.

2.4 Advanced optimization

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

2.4.1 Momentum training

Momentum modifies back propagation by allowing prior iterations to influence the current update. 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 the momentum is best ramped up slowly.

A recommended improvement when using momentum updates is the Nesterov accelerated gradient method. Under this method the updates are further modified such that

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

2.4.2 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 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).

2.4.3 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. Specification of 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 the 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 adaptive learning rate to produce the best results, and this option is kept as the default.

The first of two hyper parameters for adaptive learning is ρ. 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 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}.

2.5 Loading data

Loading a dataset in R for use with H2O is slightly different from the usual methodology, as we must convert our datasets into `H2OParsedData` objects. For an example, we use a toy weather dataset included in the H2O GitHub repository for the H2O Deep Learning documentation at https://github.com/0xdata/h2o/tree/master/docs/deeplearning/DeepLearningRVignetteDemo/. First load the data to your current working directory in your R Console (do this henceforth for dataset downloads), and then run the following command.

```
weather.hex = h2o.uploadFile(h2o_server, path = "weather.csv", header = TRUE, sep
= ",", key = "weather.hex")
```

To see a quick summary of the data, run the following command.

```
summary(weather.hex)
```

2.5.1 Standardization

Along with categorical encoding, H2O preprocesses data to be standardized for compatibility with the activation functions. Recall Table 1's summary of each activation function's target space. Since in general the activation function does not map into $\mathbb{R}$, 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.

2.6 Additional parameters

This section has reviewed some background on the various parameter configurations in H2O's Deep Learning architecture. H2O Deep Learning models may seem daunting since there are dozens of possible parameter arguments when creating models. However, most parameters do not need to be tuned or experimented with; the default settings are safe and recommended. Those parameters for which experimentation is possible and perhaps necessary have mostly been discussed here but there a couple more which deserve mention.

There is no default for both hidden layer size/number as well as epochs. Practice building deep learning models with 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 able to afford the computational cost. 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.

3 Use case: MNIST digit classification

3.1 MNIST overview

The MNIST database is a famous academic dataset used to benchmark classification performance. The data consists of 60,000 training images and 10,000 test images, each a standardized 28^2 pixel greyscale image of a single handwritten digit. You can download the datasets from the H2O GitHub repository for the H2O Deep Learning documentation at https://github.com/0xdata/h2o/tree/master/docs/deeplearning/DeepLearningRVignetteDemo/. Remember to save these .csv files to your working directory. Following the weather data example, we begin by loading these datasets into R as `H2OParsedData` objects.

```
train_images.hex = h2o.uploadFile(h2o_server, path = "mnist_train.csv", header =
FALSE, sep = ",", key = "train_images.hex")
test_images.hex = h2o.uploadFile(h2o_server, path = "mnist_test.csv", header = FALSE,
sep = ",", key = "test_images.hex")
```

3.2 Performing a trial run

The trial run below is illustrative of the relative simplicity that underlies most H2O Deep Learning model parameter configurations, thanks to the defaults. 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. As mentioned before, 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.

```
#Train the model for digit classification
```

```
mnist_model = h2o.deeplearning(x = 1:784, y = 785, data = train_images.hex, activation
= "RectifierWithDropout", hidden = c(200,200,200), input_dropout_ratio = 0.2, l1
= 1e-5, validation = test_images.hex, epochs = 10)
```

The model is run for only 10 epochs since it is meant just as a trial run. In this trial run we also specified the validation set to be the test set, but another option is to use n-fold validation by specifying, for example, `nfolds=5` instead of `validation=test_images`.

3.2.1 Extracting and handling the results

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

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

#Examine the performance of the trained model
mnist_model
```

The latter command returns the trained model's training and validation error. 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 controls the same value on the validation set and is set by default to be the entire validation set. In general choosing more sampled points leads to a better idea of the model's performance on your dataset; setting either of these parameters to 0 automatically uses the entire corresponding dataset for scoring. Either way, however, you can 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 `override_with_best_model` is turned on. This override sets the final model after training to be the model which achieved the lowest validation error during training, based on the sampled points used for scoring. Since the validation set is automatically set to be the training data if no other dataset is specified, either the `score_training_samples` or `score_validation_samples` parameter will control the error computation during training and, in turn, the chosen 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.

```
#Perform classification on the test set
prediction = h2o.predict(mnist_model, newdata=test_images.hex)

#Copy predictions from H2O to R
pred = as.data.frame(prediction)
```

3.3 Web interface

H2O R users have access to a slick web interface to mirror the model building process in R. After loading data or training a model in R, point your browser to your IP address+port number (e.g., localhost:12345) to launch the web interface. From here you can click on ADMIN > JOBS to view your specific model details. You can also click on DATA > VIEW ALL to view and keep track of your datasets in current use.

3.3.1 Variable importances

One useful feature is the variable importances option, which can be enabled with the additional argument `variable_importances=TRUE`. This features allows us to view the absolute and relative predictive strength of each feature in the prediction task. From R, you can access these strengths with the command `mnist_model@model$varimp`. You can also view a visualization of the variable importances on the web interface.

3.3.2 Java model

Another important feature of the web interface is the Java (POJO) model, accessible from the JAVA MODEL button in the top right of a model summary page. This button allows access to Java code which, when called from a main method in a Java program, builds the model at hand. Instructions for downloading and running this Java code are available from the web interface, and example production scoring code is available as well.

3.4 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. For example, below is an example of a grid search:

```
#Create a set of network topologies
hidden_layers = list(c(200,200), c(100,300,100),c(500,500,500))

mnist_model_grid = h2o.deeplearning(x = 1:784, y = 785, data = train_images.hex,
activation = "RectifierWithDropout", hidden = hidden_layers, validation = test_images.hex,
epochs = 1, l1 = c(1e-5,1e-7), input_dropout_ratio = 0.2)
```

Here we specified three different network topologies and two different ℓ_1 norm weights. This grid search model effectively trains six different models, over the possible combinations of these parameters. Of course, sets of other parameters can be specified for a larger space of models. This allows for more subtle insights in the model tuning and selection process, as we inspect and compare our trained models after the grid search process is complete. To decide how and when to choose different parameter configurations in a grid search, see Appendix A for parameter descriptions and possible values.

```
#print out all prediction errors and run times of the models
mnist_model_grid
mnist_model_grid@model
```

```
#print out a *short* summary of each of the models (indexed by parameter)
mnist_model_grid@sumtable

#print out *full* summary of each of the models
all_params = lapply(mnist_model_grid@model, function(x) { x@model$params })
all_params

#access a particular parameter across all models
l1_params = lapply(mnist_model_grid@model, function(x) { x@model$params$l1 })
l1_params
```

3.5 Checkpoint model

Checkpoint model keys can be used to start off where you left off, if you feel that you want to further train a particular model with more iterations, more data, different data, and so forth. If we felt that our initial model should be trained further, we can use it (or its key) as a checkpoint argument in a new model. In the command below, `mnist_model_grid@model[[1]]` indicates the highest performance model from the grid search that we wish to train further. Note that the training and validation datasets and the response column etc. have to match for checkpoint restarts.

```
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)
```

Checkpoint models are also applicable for the case when we wish to reload existing models that were saved to disk in a previous session. For example, we can save and later load 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)
```

Later (e.g., after restarting H2O) we can load the saved model by indicating the host and saved model file path. This assumes the saved model was saved with a compatible H2O version (no changes to the H2O model implementation).

```
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)
```

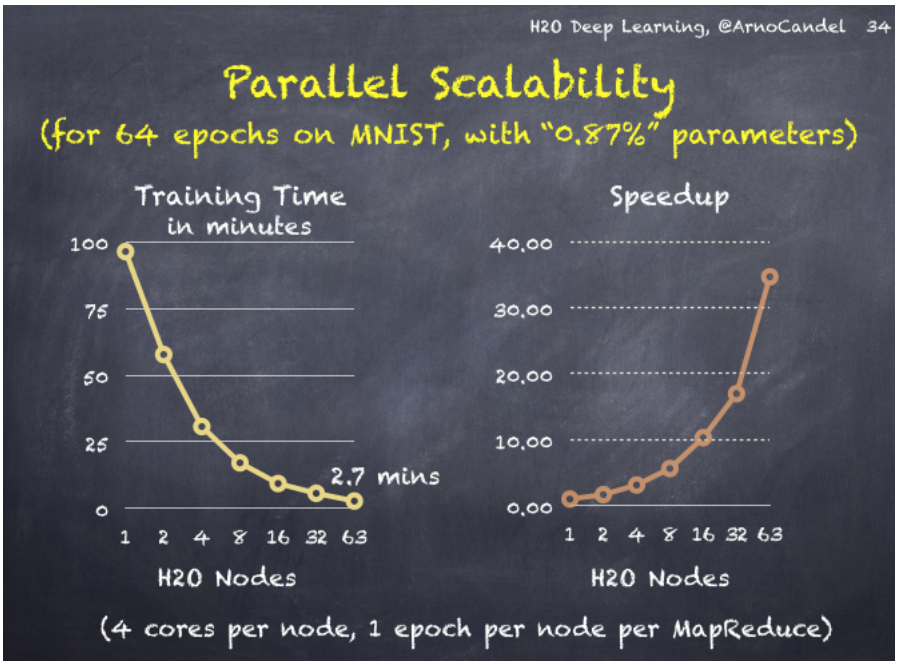
Additionally, you can also use the command

```
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.

3.6 Achieving state-of-the-art 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 (with `l1`=1e-5 and `input_dropout=0.2`) we obtain 0.87% test set error, which is a world-record-level result (within the statistical noise of $\approx 0.1\%$ with 10,000 test points), notably achieved using a distributed configuration. 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.

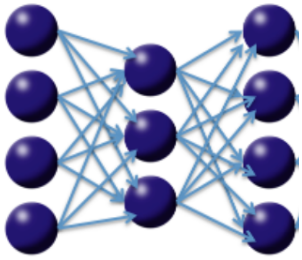


4 Deep autoencoders

4.1 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 specficially, nonlinear dimensionality reduction (Hinton et al, 2006). Consider the diagram on the following page of a three-layer neural network with one hidden layer. 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. Such an algorithm is called a deep autoencoder; these models have been used extensively for unsupervised, layer-wise pretraining of supervised deep learning tasks, but here we consider the autoencoder's application for discovering anomalies in data.



4.2 Use case: anomaly detection

Consider the deep autoencoder model described above. Given enough training data resembling some underlying pattern, the network will train itself to easily learn the identity when confronted with that pattern. However, if some "anomalous" test point not matching 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) is available here, and the test data (training data with 3 "bad" heartbeats appended for simplicity) here. Each 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=1E-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)
```

5 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.
- `data`: An `H2OParsedData` object containing the training data. No default.
- `key`: The unique hex key assigned to the resulting model. If none is given, a key will automatically be generated.
- `override_with_best_model`: If enabled, override the final model with the best model found during training. Default is true.
- `checkpoint`: Model checkpoint (either key or H2ODeepLearningModel) to resume training with.
- `classification`: A logical value indicating whether the algorithm should conduct classification. Otherwise, regression is performed on a numeric response variable.
- `nfolds`: Number of folds for cross-validation. If the number of folds is more than 1, then validation must remain empty. Default is false.
- `validation`: An `H2OParsedData` object indicating the validation dataset used to construct confusion matrix. If left blank, default is the training data.
- `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`. See section 2.2.2 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, and the middle hidden layer wil have 200 neurons. The default is c(200,200). For grid search, use list(c(10,10), c(20,20)) etc. See section 3.2 for more details. .
- `autoencoder`: Default is false. See section 4 for more details.
- `use_all_factor_levels`: Use all factor levels of categorical variables. Otherwise, the first factor level is omitted (without loss of accuracy). Useful for variable importances and auto-enabled for autoencoder.
- `epochs`: The number of passes over the training dataset to be carried out. 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`: Default is -1, but performance might depend greatly on this parameter. See section 2.2.4 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. Note that deterministic sampling and initialization might still lead to some weak sense of determinism in the model. Default is a random real number.

- `adaptive_rate`: The default enables this feature for adaptive learning rate. See section 2.4.3 for more details.

- `rho`: The first of two hyperparameters for adaptive learning rate (when it is enabled). 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 0.95. See section 2.4.3 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 learning rate is enabled. Default is 1e-6. See section 2.4.3 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`: Default value is 1e-6 (when adaptive learning is disabled). See section 2.4.2 for more details.

- `rate_decay`: Default is 1.0 (when adaptive learning is disabled). The learning rate decay parameter controls the change of learning rate across layers.

- `momentum_start`: The momentum_start parameter controls the amount of momentum at the beginning of training (when adaptive learning is disabled). Default is 0. 2.4.1 for more details.

- `momentum_ramp`: The momentum_ramp parameter controls the amount of learning for which momentum increases assuming momentum_stable is larger than momentum_start. It can be enabled when adaptive learning is disabled. The ramp is measured in the number of training samples. Default is 1e-6. See section 2.4.1 for more details.

- `momentum_stable`: The momentum_stable parameter controls the final momentum value reached after momentum_ramp 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. See section 2.4.1 for more details.

- `nesterov_accelerated_gradient`: The default is true (when adaptive learning is disabled). See Section 2.4.1 for more details.

- `input_dropout_ratio`: The default is 0. See Section 2.3 for more details.

- `hidden_dropout_ratio`: The default is 0. See Section 2.3 for more details.
- `l1`: The default is 0. See section 2.3 for more details.
- `l2`: The default is 0. See section 2.3 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 leaves this maximum unbounded.
- `initial_weight_distribution`:The distribution from which initial weights are to be drawn. The default is the uniform adaptive option. Other options are Uniform and Normal distributions. See section 2.2.1 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 1.0. See section 2.2.1 for more details.
- `loss`: The default is automatic based on the particular learning problem. See section 2.2.2 for more details.
- `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. Default is 5.
- `score_training_samples`: The number of training dataset points to be used for scoring. Will be randomly sampled. Use 0 for selecting 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 for selecting the entire training dataset (this is also the default).
- `score_duty_cycle`: Maximum fraction of wall clock time spent on model scoring on training and validation samples, and on diagnostics such as computation of feature importances (i.e., not on training). Default is 0.1.
- `classification_stop`: The stopping criteria in terms of classification error (1-accuracy) on the training data scoring dataset. When the error is at or below this threshold, training stops. Default is 0.
- `regression_stop`: The stopping criteria in terms of regression error (MSE) on the training data scoring dataset. When the error is at or below this threshold, training stops. Default is 1e-6.
- `quiet_mode`: 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 for it to be printed. 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, 0 to disable). Default is 10.

- `balance_classes`: For imbalanced data, balance training data class counts via over/under-sampling. This can result in improved predictive accuracy. Default is false.

- `class_sampling_factors`: Desired over/under-sampling ratios per class (lexicographic order). Only when balance_classes is enabled. If not specified, they 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 5.0.

- `score_validation_sampling`: Method used to sample validation dataset for scoring. The possible methods are Uniform and Stratified. Default is Uniform.

- `diagnostics`: Gather diagnostics for hidden layers, such as mean and RMS values of learning rate, momentum, weights and biases. Default is true.

- `variable_importances`: Whether to compute variable importances for input features. The implementation considers the weights connecting the input features to the first two hidden layers. Default is false.

- `fast_mode`: Enable fast mode (minor approximation in back-propagation), should not affect results significantly. Default is true.

- `ignore_const_cols`: Ignore constant training columns (no information can be gained anyway). Default is true.

- `force_load_balance`: Increase training speed on small datasets by splitting it into many chunks to allow utilization of all cores. Default is true.

- `replicate_training_data`: Replicate the entire training dataset onto every node for faster training on small datasets. Default is true.

- `single_node_mode`: 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`: 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 will (almost) all 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), otherwise it is disabled by default.